

CiS - Projekt

Micha von der Osten-Sacken

Timo Wacke

Numerische Berechnung von Planetenbahnen

Solarsystem

Um ein Solarsystem nach dem Vorbild des Unseren zu simulieren haben wir in der Datei Solar.py die Massen, Distanzen zur Sonne und Bahngeschwindigkeiten von den Planeten und unserem Mond definiert.

```
In [ ]: #verify that Solar.py and progressBar.py are in the same directory as this jupyter Notebook
import Solar as sol
import numpy as np
import matplotlib.animation as animation
import matplotlib.pyplot as plt
import progressBar
```

Verfahren

Die Bewegungsgleichung für 3 oder mehr Körper lässt sich im Allgemeinen nicht analytisch lösen.

Numerisch gibt es verschiedene Anstöße:

Trivial Euler Verfahren:

Zu einer Startposition $\vec{r}(0)$ mit gegebenem $\vec{v}(0)$ und einem gewählten Zeitschritt dt , berechnet man iterativ die Koordinaten für die folgenden Koordinaten nach der Vorschrift:

$$\vec{r}(t + dt) = \vec{r}(t) + dt \cdot \vec{v}(t)$$

wobei sich jeweils das neue $\vec{v}(t)$ für einen nächsten Schritt aus der Kraft $\vec{F}(r)$ welche bei $\vec{r}(t)$ auf den Körper wirkt berechnet.

$$\vec{a} = \vec{F}/m$$
$$\vec{v}(t + dt) = \vec{v}(t) + \vec{a}(t + dt) \cdot dt$$

Um die weiteren Zeitschritte zu berechnen, ist das erfahren zu Wiederholen bzw. $t = t + dt$ zu setzen.

Dieses Verfahren ist aber sehr instabil, denn selbst bei einer konstanten Beschleunigung ist die Geschwindigkeit mit der bei einem Zeitschritt gerechnet wird immer nur für den Anfangszeitpunkt des Zeitschrittes exakt und danach immer zu wenig in die beschleunigte Richtung. Für eine konstante Kraft kommt so in jedem Zeitschritt der gleiche Fehler hinzu und der summiert sich auf und wird immer größer. Beim Beispiel der Planetenbewegung ist

die Kraft zwar nicht konstant, aber immer nach innen gerichtet, sodass der Planet bei dem Verfahren immer nicht weit genug nach innen beschleunigt wird, und immer weiter nach außen wegdriftet. Berechnet man zu erst $\vec{v}(t + dt)$ und mit dieser Geschwindigkeit dann $\vec{r}(t + dt)$ so wird die Beschleunigung zu weit nach innen gerichtet sein, und der Planet immer weiter nach innen driften.

Durch möglichst kleine Zeitschritte lässt sich der Fehler und damit der Drift zwar minimieren aber nicht beseitigen. Vergleichbar mit der Unter und Obersumme beim Integrieren.

Besser Verlet Verfahren:

Vor dem ersten Schritt berechnet man die Kraft $\vec{F}(r)$ und die daraus resultierende Beschleunigung $\vec{a} = \vec{F}/m$ die auf den Körper an den entsprechenden Startkoordinaten wirkt.

Dann passt man die Geschwindigkeit nur um die Beschleunigung für einen halben Zeitschritt an:

$$\vec{v}(t + \frac{1}{2}dt) = \vec{v}(t) + \frac{1}{2}dt \cdot \vec{a}(t).$$

Wenn wir von nun an $\vec{r}(t + dt)$ berechnen machen wir das auf der Grundlage von $\vec{v}(t + \frac{1}{2}dt)$

Und wenn wir $\vec{v}(t + dt + \frac{1}{2}dt)$ berechnen machen wir das auf der Grundlage von $\vec{F}(t + dt)$. Dann werden in jedem Zeitschritt die Koordinaten

$$\vec{r}(t + dt) = \vec{r}(t) + \vec{v}(t + \frac{1}{2} \cdot dt) \cdot dt$$

anhand der Geschwindigkeit zum Zeitpunkt in der Mitte des Zeitschrittes berechnet. Und die Geschwindigkeit

$$\vec{v}(t + 3/2 \cdot dt) = \vec{v}(t + \frac{1}{2} \cdot dt) + \vec{a}(t + dt) \cdot dt$$

wird anhand der Beschleunigung aus der Kraft zu den Koordinaten von ganzzahligen Zeitschritten berechnet, wobei das wieder die Beschleunigung zu dem Zeitpunkt ist, der genau in der Mitte zwischen den Zeitpunkten liegt, für die die Geschwindigkeiten berechnet wurden. Also wäre die Rechnung schonmal für sich linear mit der Zeit ändernde Beschleunigungen und Geschwindigkeiten exakt und der Rechenaufwand ist kaum gestiegen. Das Verlet verfahren lässt sich auch über eine Taylorentwicklung zur 3. Ordnung von der Mitte in beide Richtungen herleiten, der Fehler liegt damit erst in 4. Ordnung. Die antisymmetrische Anordnung des 1. und 3. Terms lässt sich diese wegkürzen, sodass der Rechenaufwand niedrig bleibt. Da das Verlet-Verfahren ein symplektisches ist bleibt beim Verlet-Verfahren das Volumen des Phasenraums erhalten. Dies haben wir versucht zu demonstrieren im Laufe des Projekts.

Zeitschritte, Einheiten und Particle Class

Beim Verlet-Algorithmus werden für n Iterationen mit dem Zeitschritt h die Positionen \vec{r} und Geschwindigkeiten \vec{v} der Massenpunkte ("Particle") mit folgenden Zeitschritten ausgerechnet:

$$\vec{r}(t_0 + h \cdot i)$$

$$\vec{v}(t_0 - \frac{h}{2} + h \cdot i)$$

wobei

$$i \in 1, \dots, n$$

Die Werte für $\vec{r}(t)$ und $\vec{v}(t)$ eines Particle werden dabei in folgendermaßen berechnet:

$$\vec{r}(t) = \vec{r}(t - h) + h \cdot \vec{v}(t - \frac{h}{2})$$

$$\vec{v}(t) = \vec{v}(t - h) + h \cdot \vec{a}(t - \frac{h}{2})$$

Dabei ist $\vec{a}(t)$ die Beschleunigung des Particle, die aus $\vec{r}(t)$ folgendermaßen berechnet wird:

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

$$\vec{F} = G \cdot m \cdot \sum_{j=1}^p \frac{m_j}{|\vec{r}_j - r|^2}$$

$$\Rightarrow \vec{a} = G \cdot \sum_{j=1}^p \frac{m_j}{|\vec{r}_j - r|^2}$$

Dabei sind m_j die Massen und \vec{r}_j die Ortsvektoren der p anderen Körper im System. Um das ständige Multiplizieren mit der Gravitationskonstante G während der Simulation zu vermeiden, können wir stattdessen einfach zu Beginn alle Massen jeweils mit ihr multiplizieren und diese Größe λ statt der Masse unter `particle.mass` speichern. Dann berechnet sich die Beschleunigung folgendermaßen:

$$\vec{a} = \sum_{j=1}^p \frac{\lambda_j}{|\vec{r}_j - r|^2}$$

mit

$$\lambda_j = m_j \cdot G$$

Dabei sind λ_j die mit der Gravitationskonstante multiplizierten massen `particle.mass`. Die Massen werden im Code aber auch für die Berechnung der kinetischen und potentiellen Energie verwendet, also muss bei diesen Berechnungen die Masse durch $\frac{\lambda}{G}$ ersetzt werden:

$$W_{pot} = G \frac{-m_1 \cdot m_2}{|\vec{r}_1 - \vec{r}_2|} = \frac{1}{G} \cdot \frac{-\lambda_1 \cdot \lambda_2}{|\vec{r}_1 - \vec{r}_2|}$$

$$W_{kin} = \frac{1}{2} \cdot m \cdot v^2 = \frac{1}{G} \cdot \lambda \cdot v^2$$

Bei diesen Formeln sieht man dann, dass man die Gravitationskonstante einfach weglassen kann, und alle Energien werden mit dem gleichen Faktor skaliert, sodass alle Verhältnisse noch stimmen.

```
In [ ]: class Particle:

    """
    Physics for a Particle and it's interaction with another
    """

    @params
        name      - Required : name of the particle set
        mass      - Required : mass
        coord     - Required : coordinates in dict {x: , y: , z: }
        velocity  - Required : velocity in dict {x: , y: , z: }
        color     - Required : HEX code, or Color name
    """

    def __init__(self, name, mass, coord, velocity, color):
        self.name = name
        self.mass = mass
        self.coord = np.array([coord["x"], coord["y"], coord["z"]])
        self.velocity = np.array([velocity["x"], velocity["y"], velocity["z"]])
        self.color = color

    def move(self, dt):
        """
        Applies a movement with the particles currently set velocity
        for the timestep of dt to it.

        @params
            dt          - Required : time for which the velocity should be applied to t
        """

        delta = self.velocity * dt
        self.coord = self.coord + delta

    def kineticEnergy(self):
        """
        returns the particles current kinetic energy
        """
        return self.mass / 2 * np.linalg.norm(self.velocity)**2

    def momentum(self):
        """
        returns the particles current momentum vector
        """
        return self.mass * self.velocity

    def phi(self):
        """
        returns the particles current angle in radian relative to the coordinate system
        """
        radianHalf = np.arccos(self.coord[0] / np.linalg.norm(np.array([self.coord[0], self.coord[1]])))
        if self.coord[1] < 0:
            return 2 * np.pi - radianHalf
        return radianHalf

    def potEnergy(self, otherParticle):
        """
        returns the particles current potential energy in the gravity field of the other
        """

        @params
            otherParticle : the particle with the gravity field
```

```

"""
diff = np.subtract(self.coord, otherParticle.coord)
radius = np.linalg.norm(diff)
return - self.mass * otherParticle.mass / radius

def accelerate(self, other, dt):
    """
    adjusts the velocity of two particles, based on the gravity force they apply to

    @params
        otherParticle : the particle that's responsible for the acceleration
    """
    diff = np.subtract(other.coord, self.coord)
    radius = np.linalg.norm(diff)
    self.velocity = self.velocity + diff * other.mass * dt / (radius**3)
    other.velocity = other.velocity - diff * self.mass * dt / (radius**3)

```

Particle Factory

Im folgenden definieren wir einige Sets an Himmelskörpern -> "Particles" für die folgenden Simulationen und eine Methode um die Geschwindigkeiten von Particles in einer Liste so zu "shiften", dass der Schwerpunkt des Systems konstant ist.

```
In [ ]: class ParticleFactory:

    """
    Class to handle a set of particle and use it in a simulation

    @params
        name      - Required  : name of the predefined particle set in: \n{"Solar",
    """

    def __init__(self, name: str):
        self.name = name
        self.particleList = []
        self.setParticles()
        self.setSizeAndColor()

    def setSizeAndColor(self):
        try:
            self.colors = [p.color for p in self.particleList]
            self.masses = [p.mass for p in self.particleList]
            maxsize = max(self.masses) ** (1/8)
            self.sizes = list(map(lambda x: x**(1/8) / maxsize * 69, self.masses))
        except:
            self.sizes = [1] * len(self.particleList)
            self.colors = ["black"] * len(self.particleList)

    def importParticles(self, particles: list[Particle]):
        self.particleList = particles
        self.setSizeAndColor()

    def setParticles(self):
        if self.name == "Solar":
            star = Particle("Sun", sol.mass_sun, {"x": 0, "y": 0, "z": 0}, {
                "x": 0, "y": 0, "z": 0}, "yellow")
            mercury = Particle("Mercury", sol.mass_mercury, {"x": 0, "y": sol.distance_m

```

```

        "x": sol.velocity_mercury, "y": 0, "z": 0}, "#DDCC44")
venus = Particle("Venus", sol.mass_venus, {"x": 0, "y": sol.distance_venus,
        "x": sol.velocity_venus, "y": 0, "z": 0}, "#884400")
earth = Particle("Earth", sol.mass_earth, {"x": 0, "y": sol.distance_earth,
        "x": sol.velocity_earth, "y": 0, "z": 0}, "steelblue")
mars = Particle("Mars", sol.mass_mars, {"x": 0, "y": sol.distance_mars, "z":
        "x": sol.velocity_mars, "y": 0, "z": 0}, "#EE1111")
jupiter = Particle("Jupiter", sol.mass_jupiter, {"x": 0, "y": sol.distance_ju
        "x": sol.velocity_jupiter, "y": 0, "z": 0}, "chocolate")
saturn = Particle("Saturn", sol.mass_saturn, {"x": 0, "y": sol.distance_satu
        "x": sol.velocity_saturn, "y": 0, "z": 0}, "goldenrod")
uranus = Particle("Uranus", sol.mass_uranus, {"x": 0, "y": sol.distance_urar
        "x": sol.velocity_uranus, "y": 0, "z": 0}, "lightseagreen")
neptune = Particle("Neptune", sol.mass_neptune, {"x": 0, "y": sol.distance_r
        "x": sol.velocity_neptune, "y": 0, "z": 0}, "cornflowerblue")
moon = Particle("Moon", sol.mass_moon, {"x": sol.distance_moon, "y": sol.dis
        "x": sol.velocity_earth, "y": sol.velocity_moon, "z": 0}, "#00FFFF")
self.particleList = [star, mercury, venus, earth, mars, jupiter, saturn, ura

if self.name == "Tatooine":
    sun1 = Particle("Tatoo I", 50, {"x": 0, "y": 10, "z": 0}, {
        "x": -1, "y": 0, "z": 0}, "peachpuff")
    sun2 = Particle(
        "Tatoo II", 50, {"x": 0, "y": -10, "z": 0}, {"x": 1, "y": 0, "z": 0}, "c
    planet = Particle("Planet", 1, {"x": 100, "y": 0, "z": 0}, {
        "x": 0, "y": 1, "z": 0}, "cadetblue")

    self.particleList =[sun1, sun2, planet]

if self.name == "Tatoo":
    sun1 = Particle("Tatoo I", 100, {"x": 0, "y": 15, "z": 0}, {
        "x": -1, "y": 0, "z": 0}, "peachpuff")
    sun2 = Particle(
        "Tatoo II", 100, {"x": 0, "y": -15, "z": 0}, {"x": 1, "y": 0, "z": 0}, 'c
    planet = Particle("Planet", 1, {"x": 0, "y": 0, "z": 0}, {
        "x": 3, "y": 0.5, "z": 0}, "cadetblue")

    self.particleList =[sun1, sun2, planet]

if self.name == "Elipse":
    star = Particle("Star", 1000, {"x": 0, "y": 0, "z": 0}, {
        "x": 0, "y": 0, "z": 0}, "orange")
    planet = Particle("Pluto", 1, {"x": 100, "y": 0, "z": 0}, {
        "x": 0, "y": 1.3, "z": 0}, "cadetblue")

    self.particleList = [star, planet]

if self.name == "Moon System":
    star = Particle("Star", 1000, {"x": 0, "y": 0, "z": 0}, {
        "x": 0, "y": 0, "z": 0}, "orange")
    planet = Particle("Planet", 10, {"x": 100, "y": 0, "z": 0}, {
        "x": 0, "y": 3, "z": 0}, "cadetblue")
    moon = Particle("Moon", 1, {"x": 100, "y": 4, "z": 0}, {
        "x": 1, "y": 3, "z": 0}, "black")
    self.particleList = [star, planet, moon]

if self.name == "Lagrangepoints":
    star = Particle("Star", 1000, {"x": 0, "y": 0, "z": 0}, {
        "x": 0, "y": 0, "z": 0}, "yellow")
    planet = Particle("Planet", 7, {"x": 100, "y": 0, "z": 0}, {
        "x": 0, "y": 3, "z": 0}, "green")

```

```

# L1 = Particle("L1", 0.01, {"x": 100/(sqrt(50)+1)*sqrt(50), "y": 0, "z": 0}
#                 "x": 0, "y": 3/(sqrt(50)+1)*sqrt(50), "z": 0}, "black")
# L2 = Particle("L2", 0.01, {"x": 100, "y": 0, "z": 0}, {
#                 "x": 0, "y": 3, "z": 0}, "black")
l3 = Particle("L3", 0.01, {"x": -100, "y": 0, "z": 0}, {
    "x": 0, "y": -3, "z": 0}, "black")
l4 = Particle("L4", 0.01, {"x": 100*np.cos(np.pi/3), "y": -100*np.sin(np.pi/
    "x": 3*np.sin(np.pi/3), "y": 3*np.cos(np.pi/3), "z": 0}, "black")
l5 = Particle("L5", 0.01, {"x": 100*np.cos(np.pi/3), "y": 100*np.sin(np.pi/3),
    "x": -3*np.sin(np.pi/3), "y": 3*np.cos(np.pi/3), "z": 0}, "black")
self.particleList = [star, planet, l3, l4, l5]

if self.name == "Phase Room":
    parts = [Particle("Star", 100000, {"x": 0, "y": 0, "z": 0}, {
        "x": 0, "y": 0, "z": 0}, "yellow")]

    for i in range(10):
        parts.append(Particle("Planet {}".format(i), 1, {"x": 0, "y": (i+1)*500, "z": 0}, {"x": 10-i, "y": 0, "z": 0}, '#' + str(i)*6))
    self.particleList = parts

def removeOffsetSpeed(self):
    """
    Determines the velocity of the center of mass of the particle set.
    Offsets all particles velocity by the negative of this velocity.
    This sets the total momentum of the particle set to zero.
    """
    totalMomentum = np.array([float(0), float(0), float(0)])
    totalMass = 0
    for p in self.particleList:
        totalMomentum = np.add(totalMomentum, p.momentum())
        totalMass += p.mass
    totalVelocity = totalMomentum / totalMass
    for p in self.particleList:
        p.velocity = np.subtract(p.velocity, totalVelocity)

    def __repr__(self):
        pretty = "-----\n"
        pretty += "Particle Factory - " + self.name + "\n"
        for p in self.particleList:
            pretty += "\t" + str(p) + "\n"
        pretty += "-----\n"
        return pretty

```

Exkurs Phasenraum

Um zu demonstrieren, dass das Verlet Verfahren wie oben angegeben symplektisch ist und damit das Volumen erhalten bleibt, simulieren wir ein einfaches Beispiel mit 10 Pendeln und zeichnen diese in einen zwei dimensionalen Phasenraum.

Beispiel Pendel:

Wir wählen Erdbeschleunigungskonstante = 1 Und als Energie Einheit wählen wir J / kg

```
In [ ]: #set True to show at the bottom of the file on animate
showPendulums = False
```

```

class Pendulum:

    def __init__(self, name, length, phi0, color):
        self.name = name
        self.length = length
        self.phi = phi0
        self.v = 0
        self.color = color

    def move(self, dt):
        dphi = self.v / self.length
        self.phi += dphi * dt

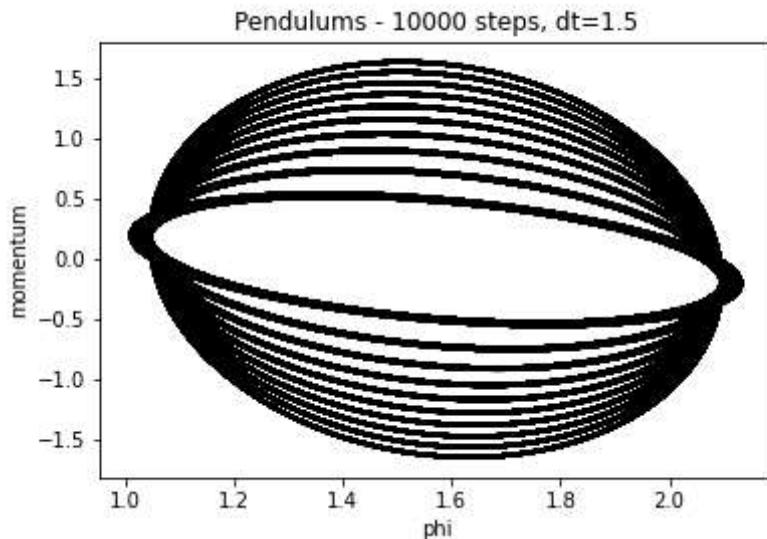
    def accelerate(self, dt):
        a = np.cos(self.phi) # * self.mass / self.mass
        self.v += a * dt

    def energy(self):
        return self.v**2 - np.sin(self.phi)

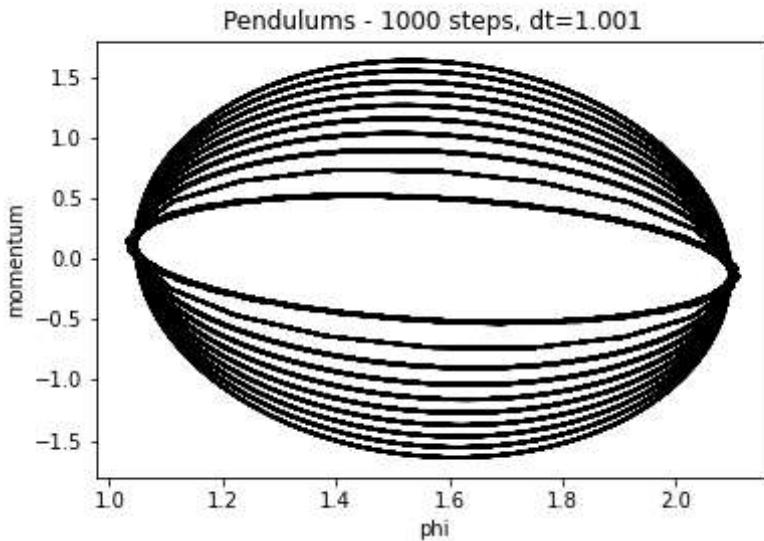
```

Es ist gut zu erkennen wie das Volumen des Phasenraums erhalten bleibt und er lediglich verzehrt wird. Dies ist leider bei den Planetenbahnen sehr schwer darzustellen, da es mind. 4 generalisierte Koordinaten gibt.

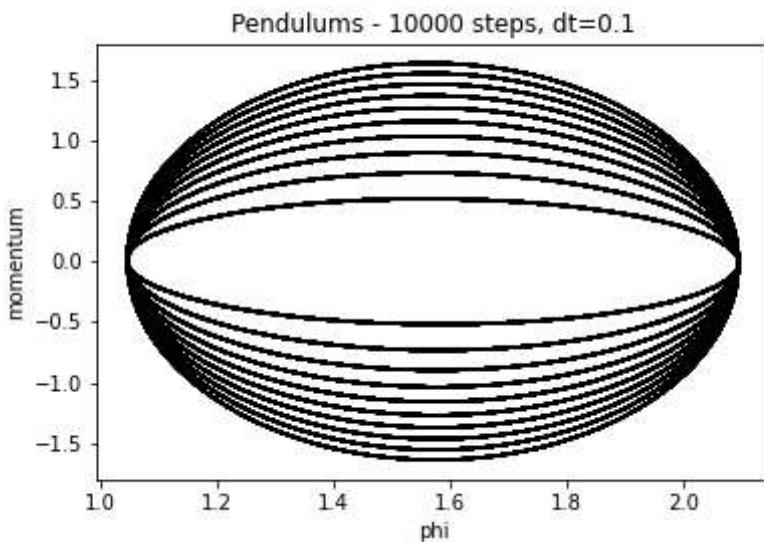
Sehr ungenau berechnet mit $h=1.5$



auch ungenau berechnet mit $h=1$



Genauer berechnet mit h=0.1



Simulate

Wie oben erklärt berechnen wir zuerst die Geschwindigkeiten aller Partikel bei $t_0 + \frac{1}{2}dt$. Dafür wenden wir auf jeden Partikel die Beschleunigung die er durch jeweils jeden anderen Partikel erfährt an.

Für die Wechselwirkung zwischen einem Partikel `pti` mit einem Partikel `ptj` können wir dafür `pti.accelerate(ptj, self.dt/2)` nutzen.

Bei den Partikeln a, b, c, d müssen wir also folgende Methoden Aufrufe tätigen:

oder anders herum:

- `b.accelerate(a, self.dt/2)`
- `c.accelerate(a, self.dt/2)`
 - `c.accelerate(b, self.dt/2)`
- `d.accelerate(a, self.dt/2)`
 - `d.accelerate(b, self.dt/2)`

- `d.accelerate(c, self.dt/2)`

dies wird mit der Schleife

```
`for pti in particles:
    for ptj in particles:
        if ptj == pti:
            break
        pti.accelerate(ptj, self.dt/2)`
```

erreicht.

Darauf folgend wird bei jeder iteration der neue Ort mit der zuvor gesetzten Geschwindigkeit berechnet und danach auf Basis diesen neuen Ortes auf die neue Geschwindigkeit beschleunigt.

```
In [ ]: from DataResults import Data

class Simulate:
    """
    Class for Simulations

    @params
        h           - Required : size of timestep (float)
        n           - Required : number of timesteps (int)
    """

    def __init__(self, h: float, n: int ):
        self.h = h
        self.n = n

    def verlet(self, particles: list[Particle]) -> Data:
        """
        Calculates the movement of particles with Verlet algorithm
        Returns positions in multidimensional array [particle p][dimension 0-2][timestep]
        Returns maximum and minimum of the coordinates too

        @params
            particles      - Required : list of Particle Objects
        """
        t_axis = np.linspace(0, self.n*self.h, num=self.n)
        energy = np.zeros(self.n)
        p_axes = []
        p_momentum = []
        p_radius = []
        p_phi = []

        center = np.array([float(0), float(0), float(0)])
        masssum = 0
        for p in particles:
            p_axes.append([np.zeros(self.n), np.zeros(self.n), np.zeros(self.n)])
            masssum += p.mass
            center += p.coord * p.mass
            p_momentum.append(np.zeros(self.n))
            p_phi.append(np.zeros(self.n))
```

```

        p_radius.append(np.zeros(self.n))
        center *= (1/masssum)

        for pti in particles:
            for ptj in particles:
                if ptj == pti:
                    break
                # Vor dem Ersten Schritt die Beschleunigung für einen halben Zeitschritt
                pti.accelerate(ptj, self.h/2)
        for i in range(self.n):
            for pt in particles:
                pt.move(self.h)
            for pti in particles:
                for ptj in particles:
                    if pti == ptj:
                        break
                    energy[i] += pti.potEnergy(ptj)
                    #force = pti.gravityForce(ptj)
                    #pti.accelerate(force, self.h)
                    #ptj.accelerate(force, self.h)
                    pti.accelerate(ptj, self.h)
            for p, pt in enumerate(particles):
                energy[i] += pt.kineticEnergy()
                p_momentum[p][i] = np.linalg.norm(pt.momentum())
                p_radius[p][i] = np.linalg.norm(np.subtract(pt.coord, center) )
                p_phi[p][i] = pt.phi()
                p_axes[p][0][i] = pt.coord[0]
                p_axes[p][1][i] = pt.coord[1]
                p_axes[p][2][i] = pt.coord[2]
            if (i+1)/self.n*100%5 == 0:
                progressBar.draw(i, self.n, "Verlet", "Complete", length=50)

        return Data(p_axes, p_momentum, p_radius, p_phi, energy, t_axis)

# useful if you create a Pendulum class to experiment
def verletPendulum(self, pendulumList: list) -> Data:

    """
    Calculates the movement of pendulums with Verlet algorithm
    Returns positions in multidimensional array [particle p][dimension 0-2][timestep]
    Returns maximum and minimum of the coordinates too
    """

    t_axis = np.linspace(0, self.n*self.dt, num=self.n) # a list for each timestep to fill in with the total time
    energy = np.zeros(self.n) # a list for each timestep to fill in with the total energy
    p_axes = [] # for every particle a list of dimensions. And for each dimension a list of timesteps
    p_momentum = [] # for every particle a list with all timesteps to fill in with the total momentum
    p_radius = [] # for every particle a list with all timesteps to fill in with the total radius
    p_phi = [] # for every particle a list with all timesteps to fill in with the total phi

    for p in pendulumList:
        p_axes.append(np.zeros(self.n))
        p_momentum.append(np.zeros(self.n))
        p_phi.append(np.zeros(self.n))
        p_radius.append(np.zeros(self.n))

    for p in pendulumList:

```

```

        p.accelerate(self.dt/2)

    # Iterate for each timestep
    for i in range(self.n):

        for p in pendulumList:
            p.move(self.dt)
            p.accelerate(self.dt/2)

    # put the new data on the record
    for p, pl in enumerate(pendulumList):
        energy[i] += pl.energy()
        p_momentum[p][i] = pl.v
        p_radius[p][i] = pl.length
        p_phi[p][i] = pl.phi
        p_axes[p][i] = pl.phi

    # update the progress bar after every 5% progress
    if (i+1)/self.n*100%5 == 0:
        progressBar.draw(i, self.n, "Pendulum", "Complete", length=50)
    return Data(p_axes, p_momentum, p_radius, p_phi, energy, t_axis)

```

Animate Class

Diese Klasse bekommt eine particleFactory übergeben, aus dieser werden die Particle, deren Farben und Größen gewonnen.

Bekommt eine simulator Funktion wie etwa verlet() aus der Simulate class übergeben. Das zu ihr gehörige simulator Objekt, kennt die Anzahl und Größe der Zeitschritte n und h, so dass die simulator Funktion nur die particeList übergeben bekommen muss.

Die Simulator Methode Verlet in der Simulat class gibt wie oben gesehen allerlei Werte zurück, die für das Plotten relevant sind.

```

In [ ]: from collections.abc import Callable

class Animate:
    """Animated scatter plot and charts using matplotlib.animations.FuncAnimation.

    @params
        particleFactory - Required : array of Objects from Particle Class (Particle)
        simulator       - Required : method like Simulate.verlet(), that returns data
    """
    def __init__(self, particleFactory: ParticleFactory, simulator: Callable[[list[Particle]], np.ndarray]):
        self.particleFactory = particleFactory
        self.data = simulator(particleFactory.particleList)
        self.n = len(self.data.t_axis)
        self.h = np.round(self.data.t_axis[1]-self.data.t_axis[0], 3)
        # Setup the figure and axes...
    def create_video(self):
        self.fig, self.ax = plt.subplots()
        self.fig.set_figheight(10)
        self.fig.set_figwidth(10)
        plt.xlim([self.data.qxmin, self.data.qxmax])

```

```

plt.ylim([self.data.qymin, self.data.qymax])
# Then setup FuncAnimation.
self.frames = 500
self.fps = 25
self.skip = int(self.n / self.frames)

self.ani = animation.FuncAnimation(self.fig, self.update, interval=int(self.frames),
                                   init_func=self.setup_plot, blit=True)

def setup_plot(self):
    """Initial drawing of the scatter plot."""
    xs = []
    ys = []
    for p in self.data.p_axes:
        xs.append(p[0][0])
        ys.append(p[1][0])
    self.scat = self.ax.scatter(
        xs, ys, s=self.particleFactory.sizes, c=self.particleFactory.colors)
    return self.scat,

def update(self, i):
    """Update the scatter plot."""
    xy = []
    progressBar.draw(
        i+1, self.frames, prefix='Rendering', suffix='Complete', length=50)
    for p in self.data.p_axes:
        xy.append([p[0][i*self.skip], p[1][i*self.skip]])
    # Set x and y data...
    self.scat.set_offsets(xy)

    return self.scat,

def pathPlot(self):
    fig1, ax1 = plt.subplots()

    for p, pt in enumerate(self.particleFactory.particleList):
        ax1.plot(self.data.p_axes[p][0], self.data.p_axes[p][1], c=pt.color, linewidth=2, label=pt.name)
    fig1.set_figwidth(10)
    fig1.set_figheight(10)
    plt.xlim([self.data.qxmin, self.data.qxmax])
    plt.ylim([self.data.qymin, self.data.qymax])
    plt.legend()

    plt.title(
        '{} - {} steps, dt={}'.format(self.particleFactory.name, self.n, self.h))
    plt.show()
    plt.savefig(
        'images/{} - {} steps, dt={}.png'.format(self.particleFactory.name, self.n, self.h))

    plt.close()

def energyPlot(self):
    fig2, ax2 = plt.subplots()
    ax2.plot(self.data.t_axis, self.data.energy, 'r-', label='energy')
    plt.xlabel("time")
    plt.ylabel("total energy in system")
    plt.title(
        '{} - {} steps, dt={}'.format(self.particleFactory.name, self.n, self.h))
    plt.show()
    plt.savefig(
        'images/{} - {} steps, dt={}.png'.format(self.particleFactory.name, self.n, self.h))

```

```

        'images/{} - {} steps, dt={}_energy.png'.format(self.particleFactory.name, s
        plt.close()

    def save_video(self):
        print("\n")
        print("Saving Video")
        self.ani.save(self.particleFactory.name + '.mp4', writer='ffmpeg', fps=self.fps,
                     dpi=100, metadata={'title': 'test'})

    def phaseSpace(self):
        fig3 = plt.figure()
        ax3 = fig3.add_subplot(projection='3d')

        for p, pt in enumerate(self.particleFactory.particleList):
            ax3.scatter(self.data.p_radius[p], self.data.p_phi[p],
                        self.data.p_momentum[p], c=pt.color, s=4, label=pt.name)

        ax3.set_xlabel("radius")
        ax3.set_ylabel("phi")
        # ax3.set_zLabel("phi")
        plt.title(
            '{} - {} steps, dt={}'.format(self.particleFactory.name, self.n, self.h))
        plt.legend()
        plt.show()
        plt.savefig(
            'images/{} - {} steps, dt={}_phase.png'.format(self.particleFactory.name, se
        plt.close()

        return (self.data.p_axes)

    def closePlot(self):
        plt.show()
        plt.close()

```

Execute

1. Erstelle eine ParticleFactory mit einer Particle List
2. Stelle sicher, dass diese keinen gesamt Impuls hat
3. Erstelle eine Simulate umgebung und definiere hierbei die Schrittgrößen und Anzahl an Schritten
4. Übergebe die ParticleFactory und Simulate umgebung an animate, so dass die Berechnung und danach das Plotten gestartet wird

zum Pendel Exkurs

```
In [ ]: if(showPendulums):
    myPendulums = []

    for i in range(10):
        myPendulums.append(Pendulum("Pendulum {}".format(i+1), i*2+2, np.pi / 3, "black"))
p = ParticleFactory("Pendulums")
p.importParticles(myPendulums)
s = Simulate(h=1.5, n=10000)
a = Animate(p, s.verletPendulum)
a.energyPlot()
a.phaseSpace(False)
```

einfache Ellipse

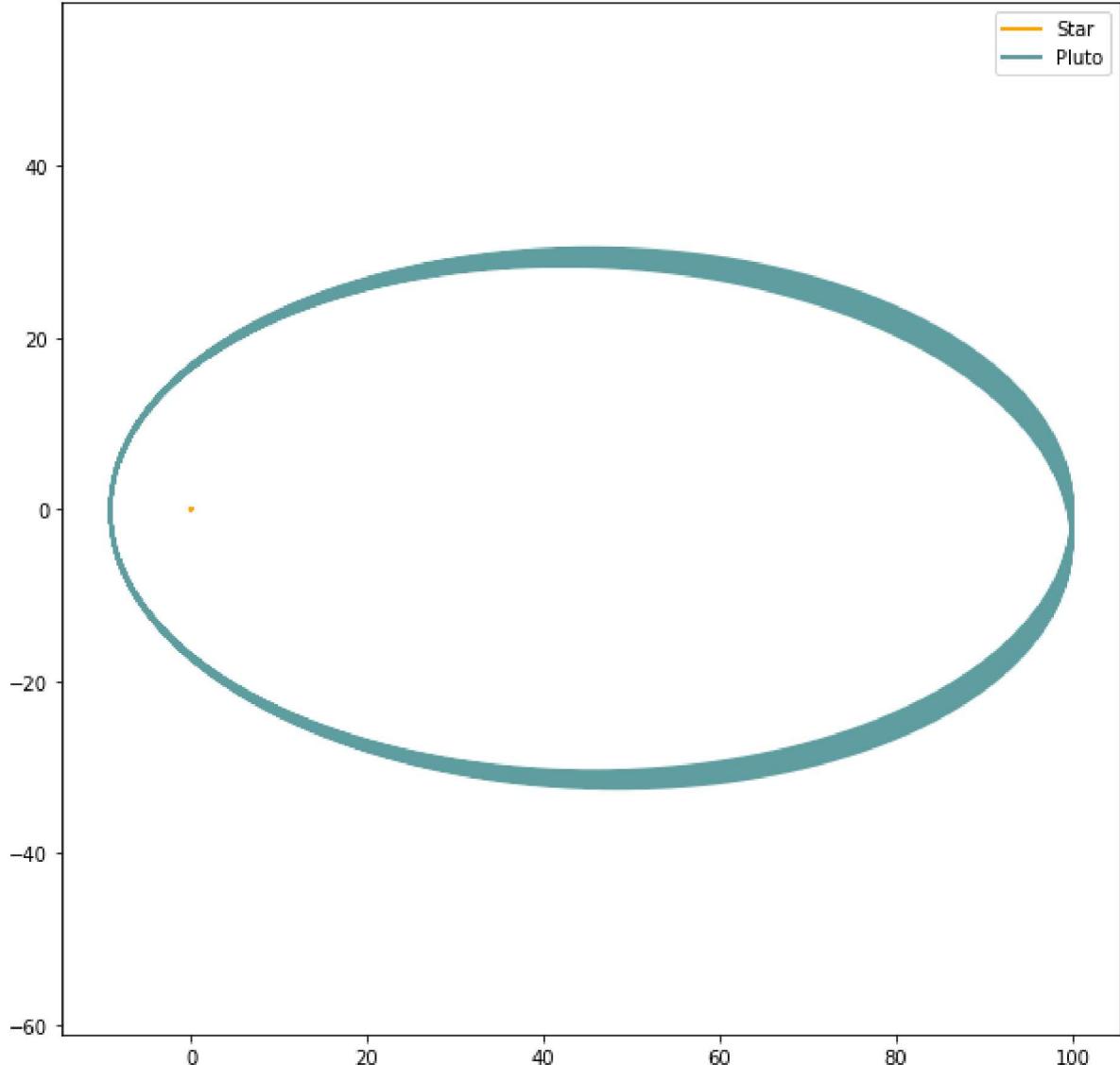
Nach den Keplerschen Gesetzen und der Newtonschen Mechanik bewegen sich alle Planeten, die an einen Stern gebunden sind, auf Elliptischen Bahnen um diesen. Eine Kreisbahn ist dabei natürlich auch eine Ellipse, aber mit Hinblick auf die Stabilität des Verlet-Algorithmus ist auch die betrachtung besonders "elliptischer" Bahnen interessant.

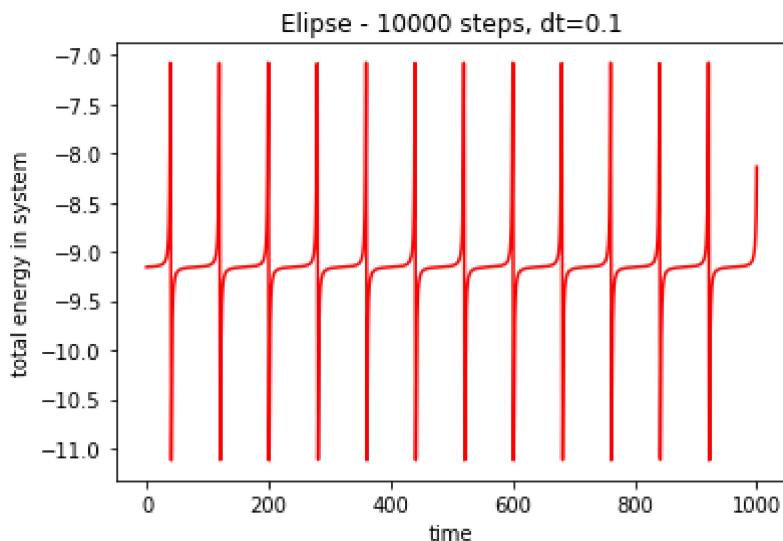
```
In [ ]: p = ParticleFactory("Ellipse")
p.removeOffsetSpeed()
s = Simulate(h=0.1, n=10000)
a = Animate(p, s.verlet)

a.pathPlot()
a.energyPlot()
a.phaseSpace()
a.closePlot()
```

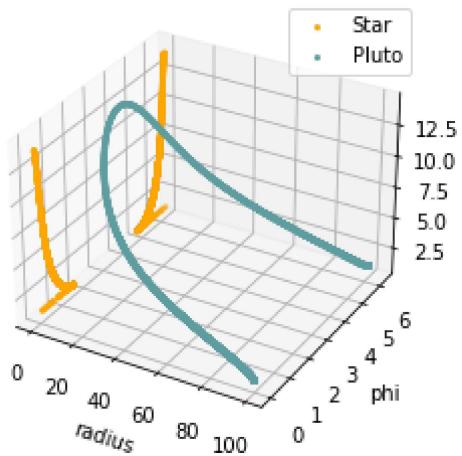
Verlet |  - | 100.0% Complete

Ellipse - 10000 steps, dt=0.1





Elipse - 10000 steps, dt=0.1



Das Ergebnis des Programms lässt schon eindeutig eine Ellipse erkennen, jedoch wird die Ellipse nicht wie in den Keplerschen Gesetzen vorausgesagt genau erhalten, sondern der Planet nimmt bei jeder Umrundung eine leicht gedrehte Bahn, als würde die Elliptische Bahn auch um den Stern kreisen. Dieser Effekt verstärkt sich wenn man die Genauigkeit verringert, also den Verlet-Algorithmus mit weniger, größeren Iterationsschritten ausführt. Interessant ist dabei, dass bei echten Planetenbahnen ähnliche Effekte auftreten, wenn man relativistische Effekte beachtet. Am Energiegraphen sieht man, dass die Energie bei sehr elliptischen Bahnen bei Berechnung mit dem Verlet-Algorithmus stark schwankt aber nach einer vollen Bahn auf der Ellipse erhalten ist. Die Amplitude dieser Schwankung verringert sich mit kleineren Iterationsschritten. So ist der Ausschlag bei:

- **dt = 0.1 noch $2/9 = 22\%$**
- **dt = 0.01 schon $0.2/9 = 2.2\%$**

2 Sonnen System

In Star Wars gibt es den Planeten Tatooine, der Teil eines Doppelsternsystems ist. Wie sich so ein System verhalten könnte, lässt sich natürlich mit unserem Programm herausfinden.

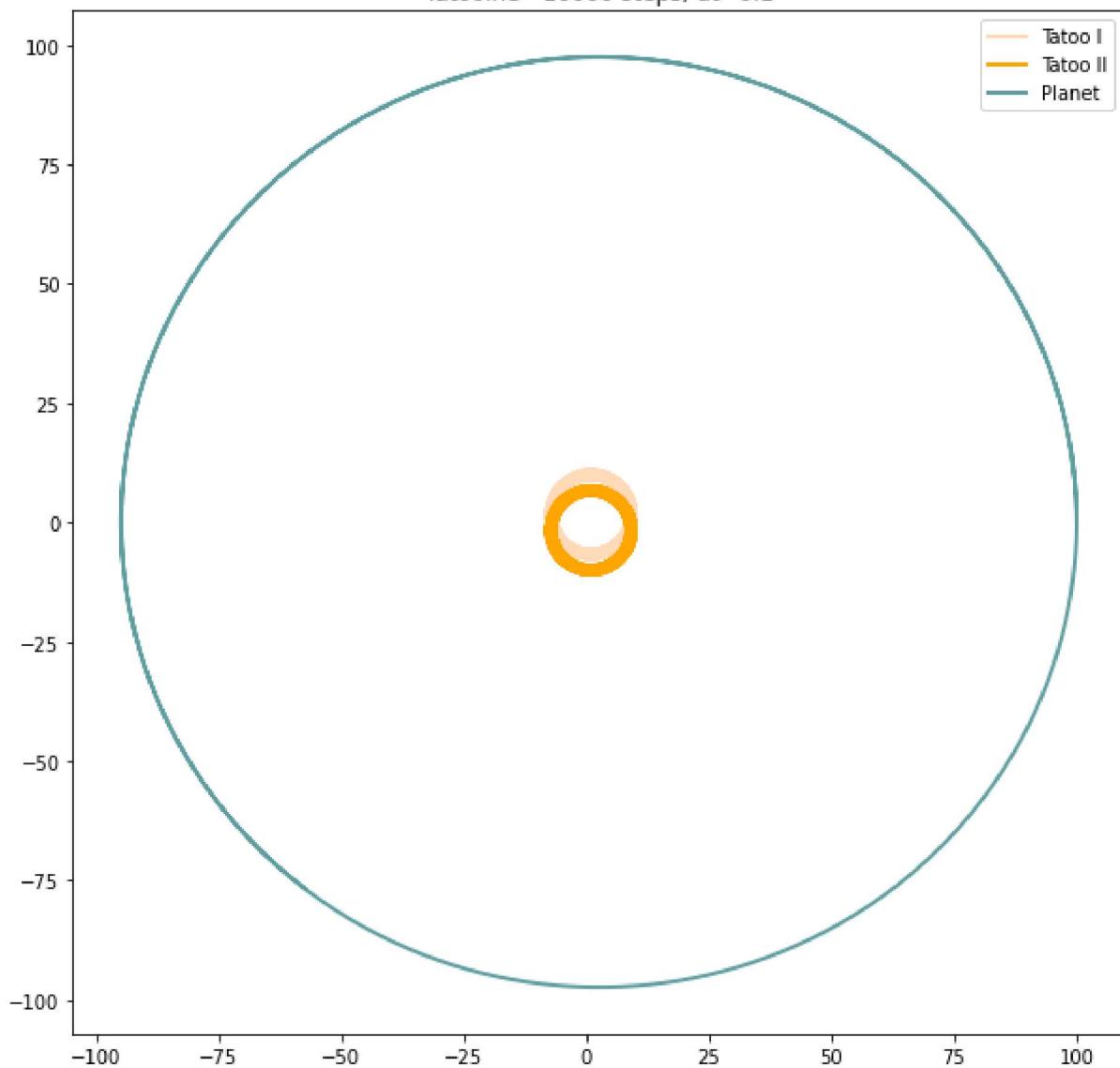
In []:

```
p = ParticleFactory("Tatooine")
p.removeOffsetSpeed()
s = Simulate(h=0.1, n=10000)
a = Animate(p, s.verlet)
```

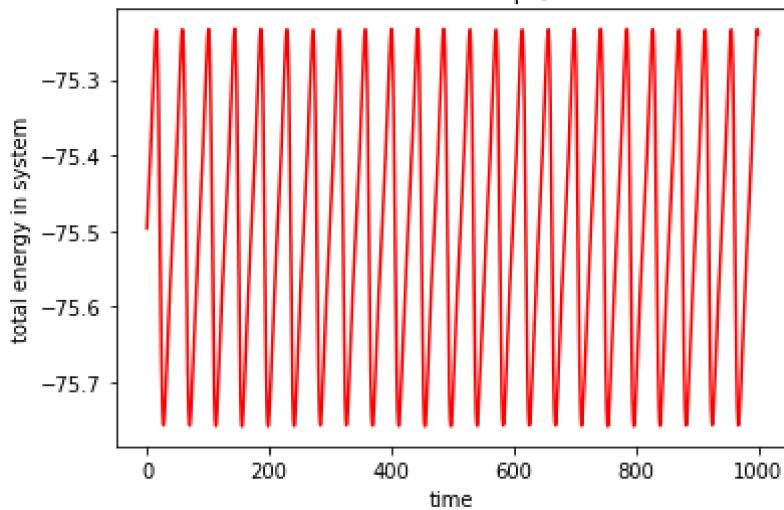
```
a.pathPlot()  
a.energyPlot()  
a.phaseSpace()  
a.closePlot()
```

Verlet |  -| 100.0% Complete

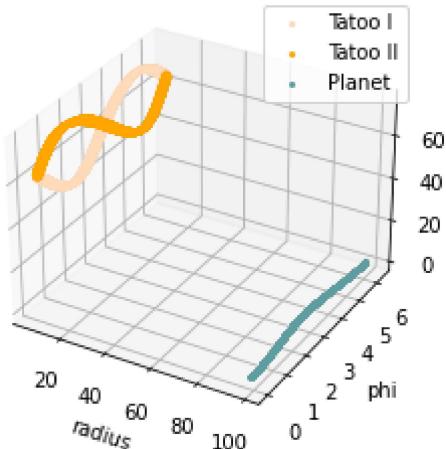
Tatooine - 10000 steps, dt=0.1



Tatooine - 10000 steps, dt=0.1



Tatooine - 10000 steps, dt=0.1



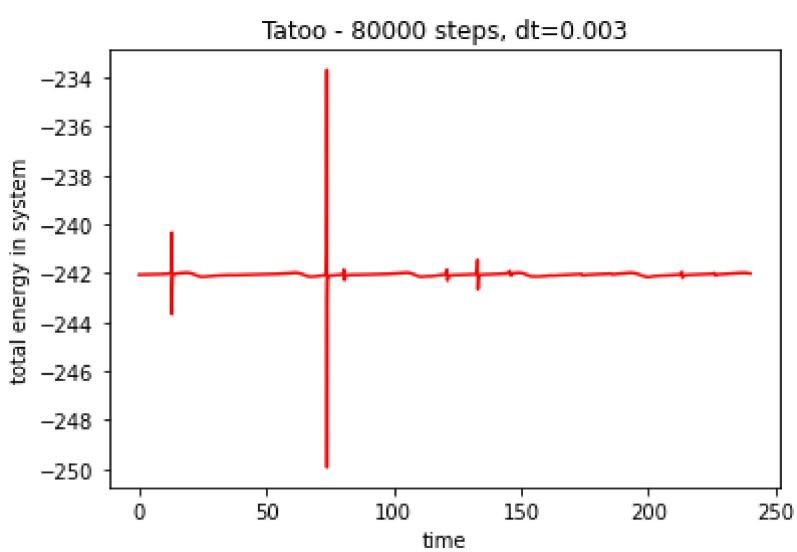
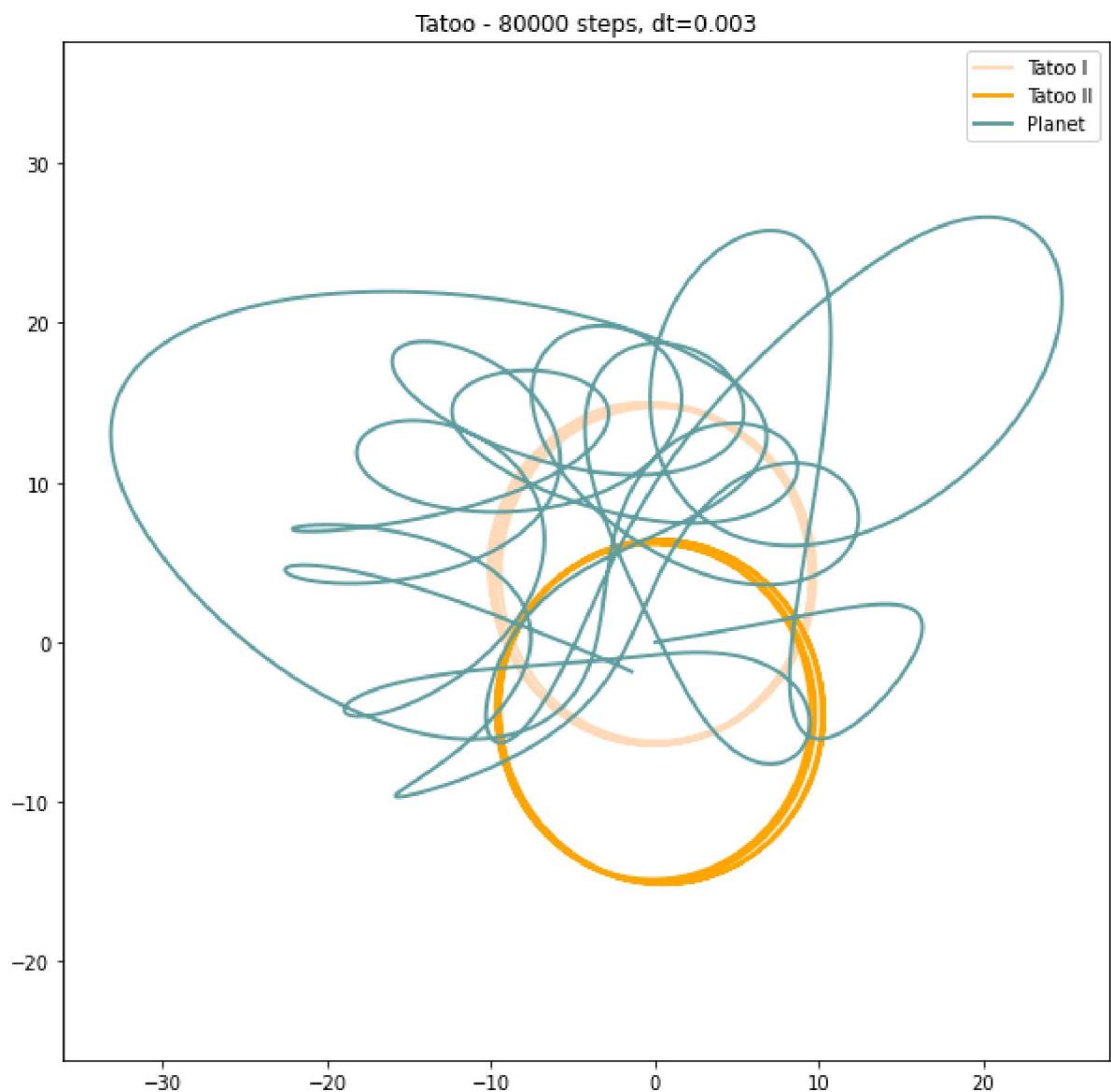
Eine Implementation nahe an der Situation wie sie in Star Wars erscheint, nämlich, dass der Planet die beiden Sterne mit relativ großem Abstand umkreist, während die beiden Sterne relativ eng umeinander kreisen, ist vielleicht etwas entäuschend, denn der Planet umkreist einfach den Schwerpunkt der beiden Sterne und man erkennt keinen großen Unterschied zu der Situation, dass man die beiden Sterne durch einen Stern mit der Summe der Massen ersetzen würde. Ein weniger triviales System erhält man wie folgt:

Variation mit Startpunkt des Planeten zwischen den Sternen

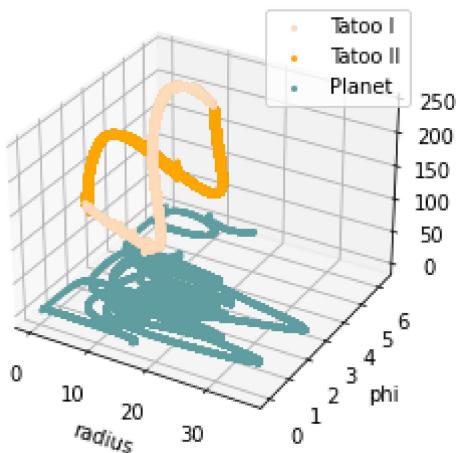
```
In [ ]: p = ParticleFactory("Tatoo")
p.removeOffsetSpeed()
s = Simulate(h=0.003, n=80000)
a = Animate(p, s.verlet)

a.pathPlot()
a.energyPlot()
a.phaseSpace()
a.closePlot()

Verlet |███████████| 100.0% Complete
```



Tatoo - 80000 steps, dt=0.003



Das System ist im Bezug auf Realismus und numerische Instabilität schwierig, wir haben für diese Simulation von allen den kleinsten Zeitschritt benötigt um die Planetenbahn zu berechnen. Der Planet fällt häufiger auf einen der Sterne oder zumindest nur sehr knapp an ihm vorbei und wird dabei sehr schnell.

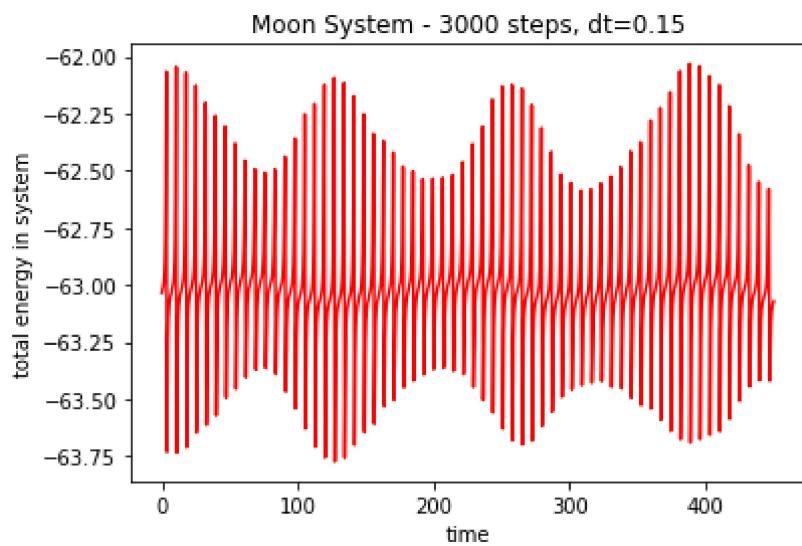
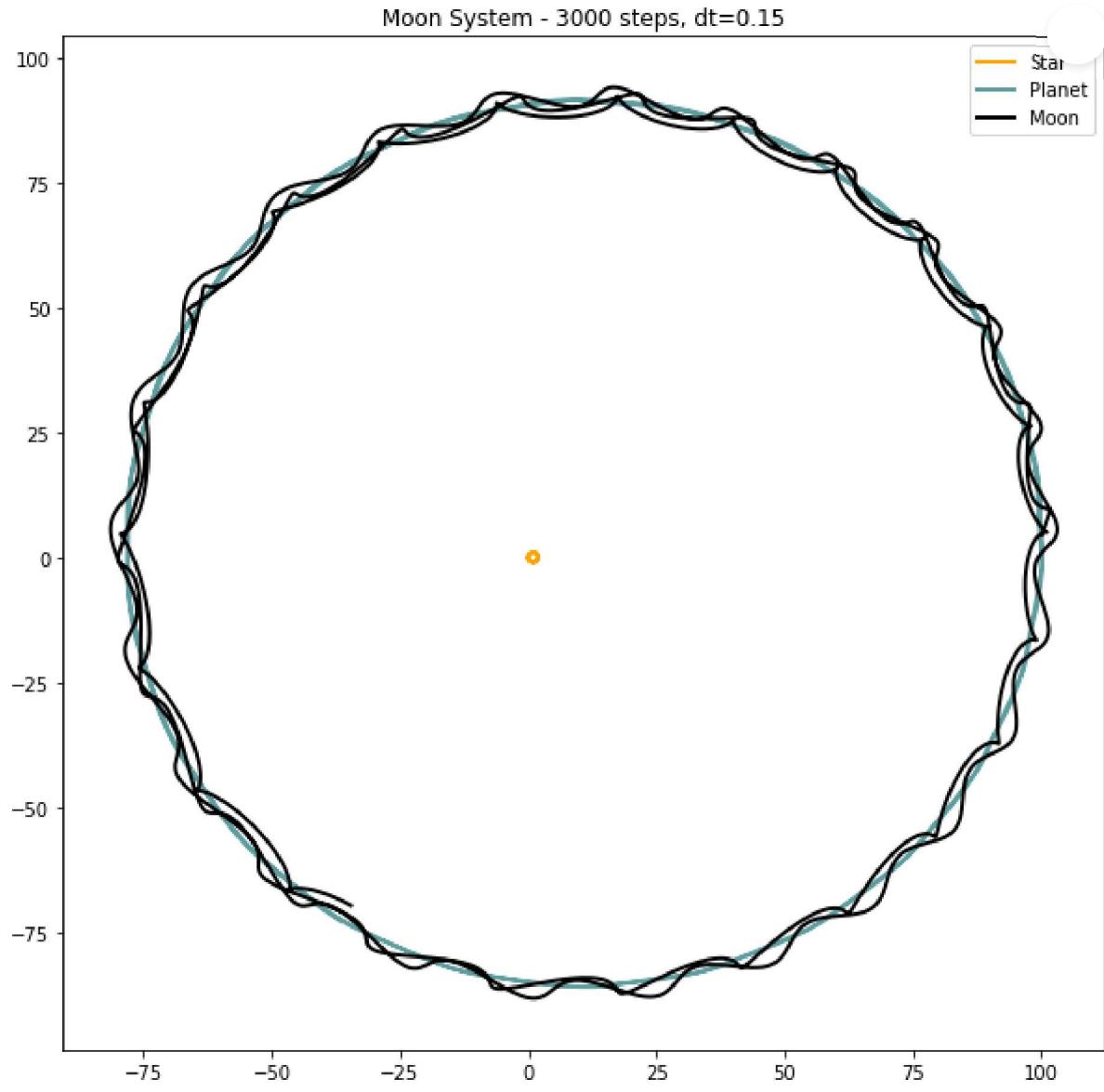
Moon System

Versuchte man, das System Erde-Sonne-Mond zu simulieren, gäbe es ein Problem das ganze maßstabsgetreu und anschaulich zugleich darzustellen, denn der Abstand zwischen Erde und Mond ist gegenüber dem Abstand der beiden zur Sonne so klein, dass man die Bahnen von Erde und Mond auf einer maßstabsgerechten Darstellung kaum unterscheiden kann, wenn man die gesamte Bahn im Blick behalten möchte. Interessant wäre doch herauszufinden, ob es auch stabile Stern-Planet-Mond-Konstellationen gibt, bei denen die Mondbahn sichtbar große Schwankungen um die Planetenbahn macht.

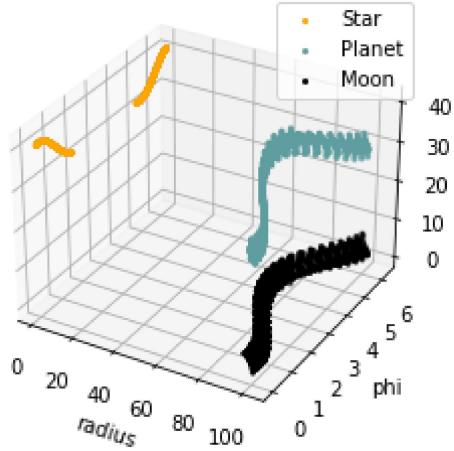
```
In [ ]: p = ParticleFactory("Moon System")
         p.removeOffsetSpeed()
         s = Simulate(h=0.15, n=3000)
         a = Animate(p, s.verlet)

         a.pathPlot()
         a.energyPlot()
         a.phaseSpace()
         a.closePlot()
```

Verlet - | 100.0% Complete



Moon System - 3000 steps, dt=0.15



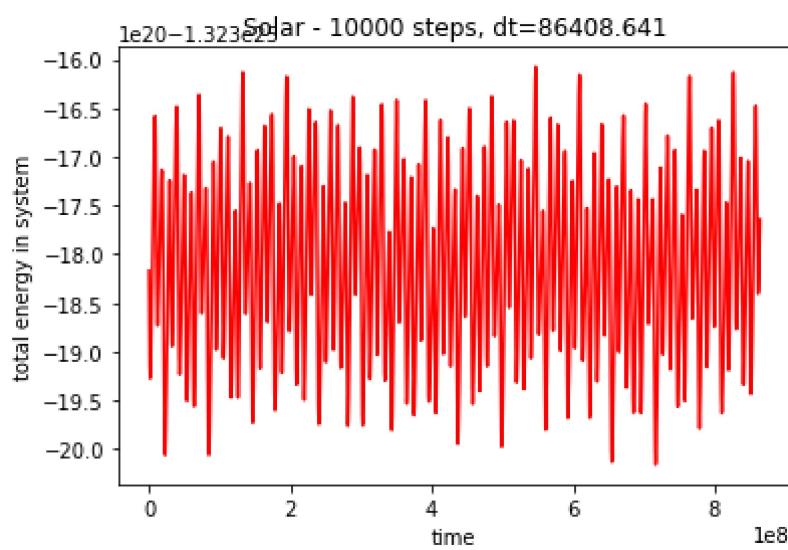
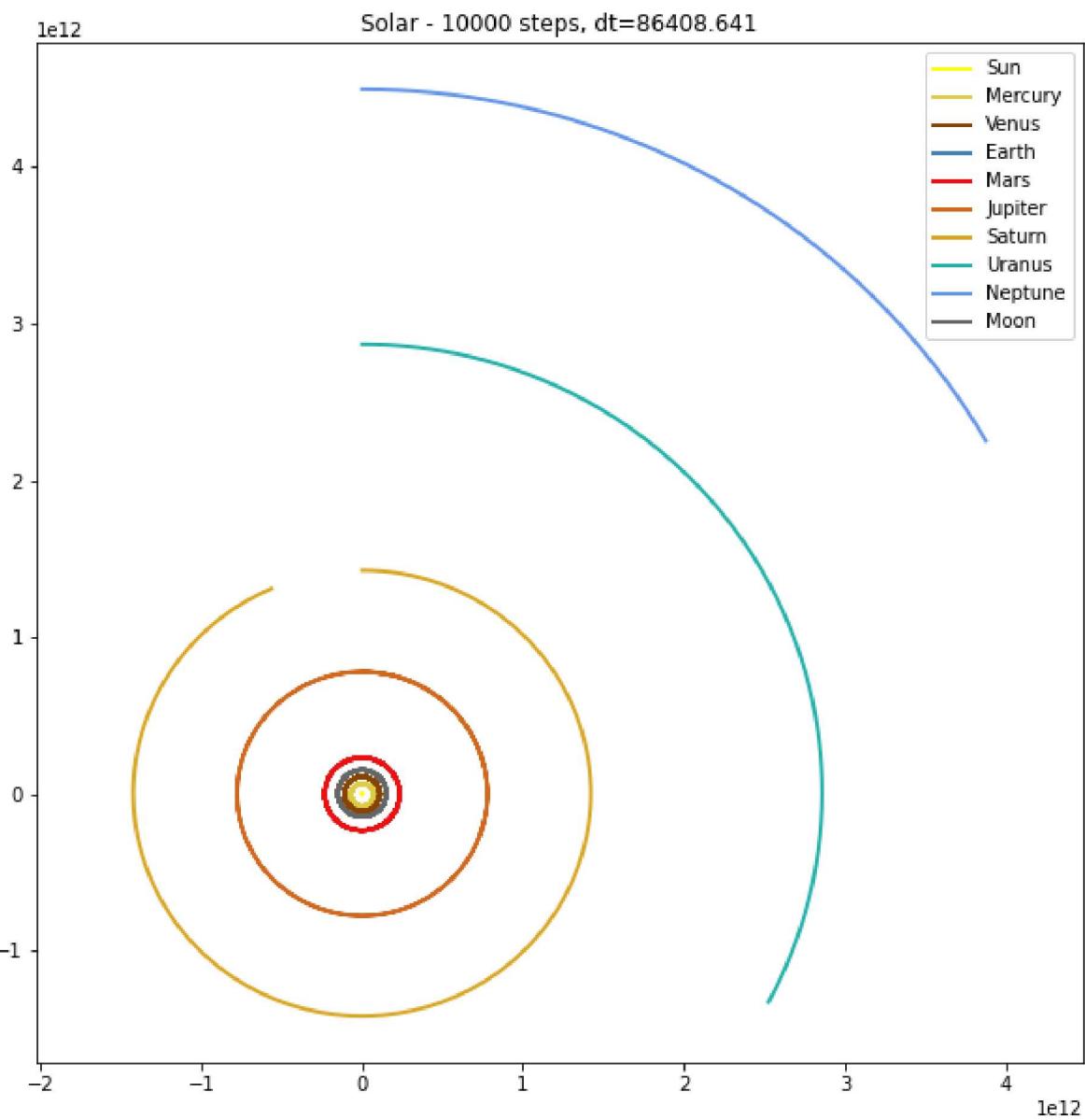
Solarsystem

Eine Simulation unseres Sonnensystems darf natürlich nicht fehlen. Es ist auch gut, um zu testen, dass das Programm auch mit echten Werten funktioniert.

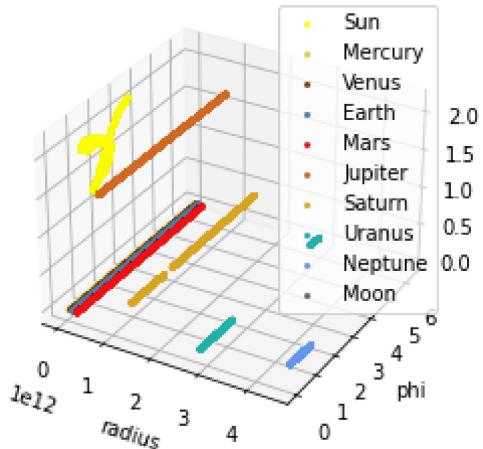
```
In [ ]: p = ParticleFactory("Solar")
p.removeOffsetSpeed()
s = Simulate(h=24*60*60, n=10000)
a = Animate(p, s.verlet)

a.pathPlot()
a.energyPlot()
a.phaseSpace()
a.closePlot()
```

Verlet | - | 100.0% Complete



Solar - 10000 steps, dt=86408.641



Lagrangepunkte

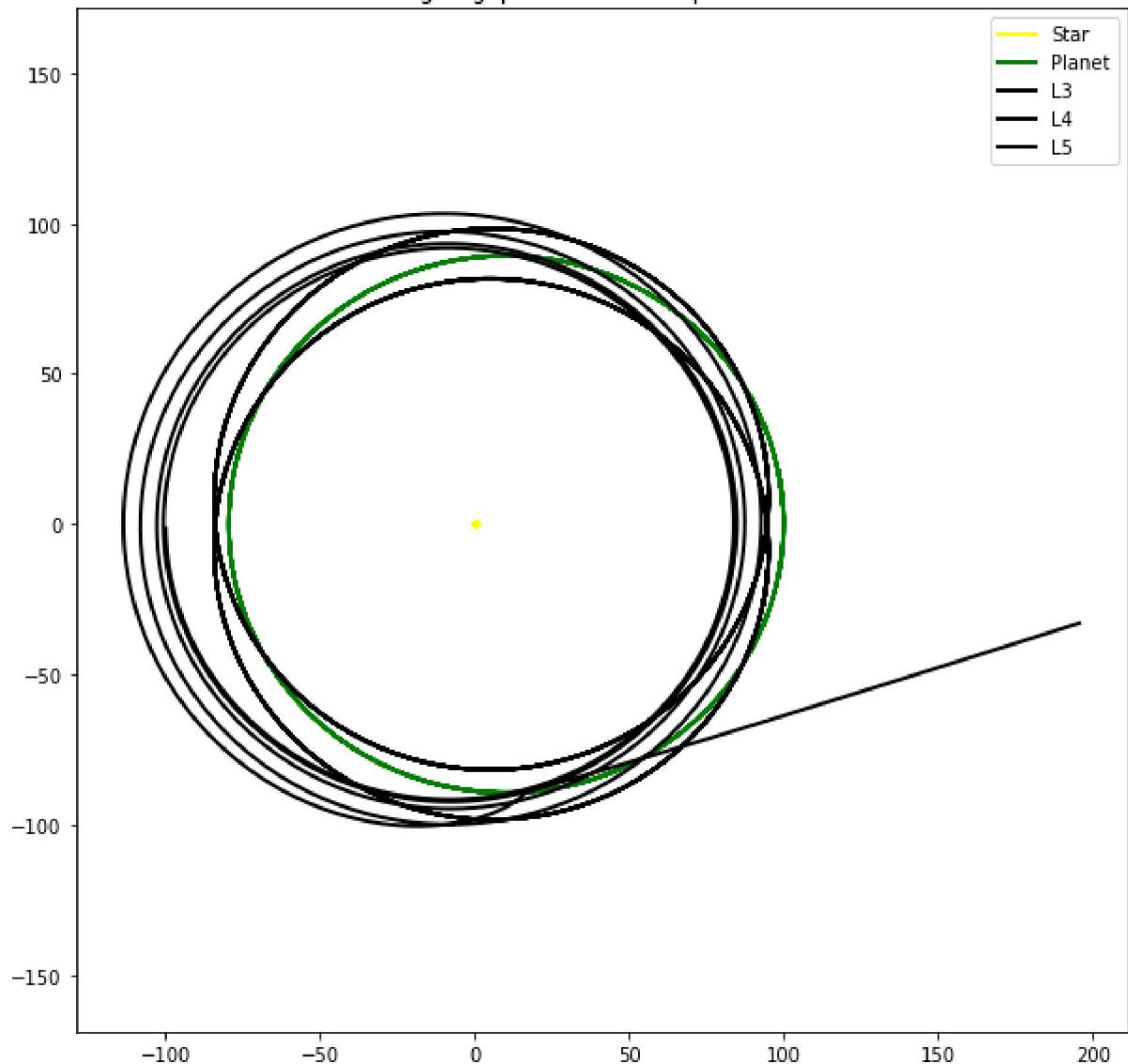
Ein gewisser Herr Lagrange hat mal 5 Punkte in einem 2-Körper-System ausgerechnet an denen keine Kraft auf einen potentiell dort befindlichen kleinen Körper wirken sollte. Hier eine Simulation, in der 3 dieser Punkte mit solchen kleinen Körpern besetzt sind, 2 davon sollten Stabil sein.

```
In [ ]: p = ParticleFactory("Lagrangepoints")
p.removeOffsetSpeed()
s = Simulate(h=0.5, n=1635)
a = Animate(p, s.verlet)

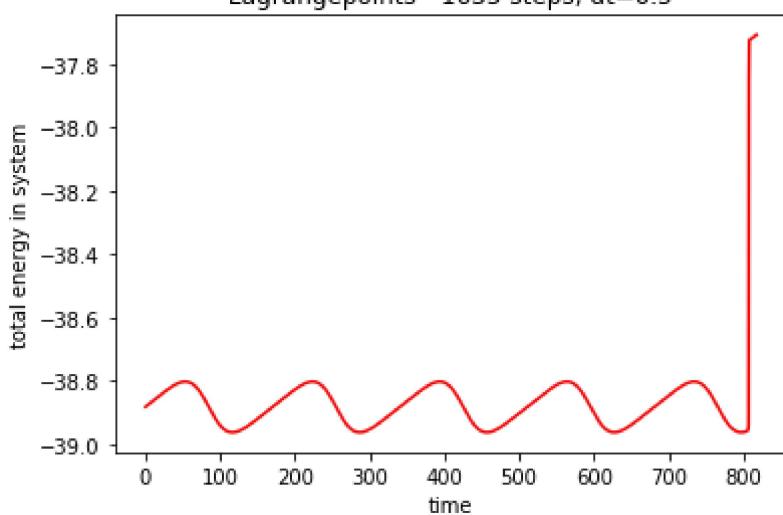
a.pathPlot()
a.energyPlot()
a.phaseSpace()
a.closePlot()
```

Verlet | - | 99.9% Complete

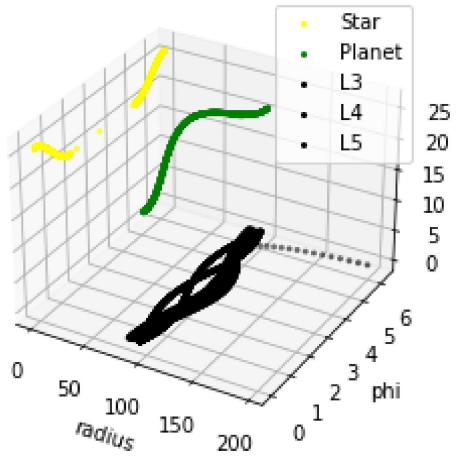
Lagrangepoints - 1635 steps, dt=0.5



Lagrangepoints - 1635 steps, dt=0.5



Lagrange points - 1635 steps, dt=0.5



In der Simulation sieht man, dass die Körper an den Lagrangepunkten tatsächlich eine Weile verweilen, nach einer Weile (je höher man den Zeitschritt dt wählt um so eher) driftet der Körper am instabilen L3 allerdings ab und verlässt in der Simulation sehr schnell das Sonnensystem. Interessant, ist hier dass die physikalische Instabilität für numerische Instabilität sorgt.