Mathesis-Projekt: Galaxiesimulation

Paul Neufeld

12. März 2025

Inhaltsverzeichnis

1 Einführung

Ziel dieses Projekts war die Entwicklung einer 3D-Simulation einer galaxienartigen Struktur in Python. Im Mittelpunkt des Projekts stand die Anwendung physikalischer Gesetze, die die Grundlage der Simulation bildeten. Aufgrund meiner aktuellen Kenntnisse und der technischen Möglichkeiten war es notwendig, bestimmte komplexe Theorien und Formeln zu vereinfachen und stattdessen auf vereinfachte Konzepte zurückzugreifen. Hierzu gehören insbesondere relativistische Effekte sowie komplexe Lösungsmethoden für Differentialgleichungen. Grundsätzlich hatte ich mit dem Projekt drei Ziele, die sich im Lauf der Erarbeitung ergeben haben:

1. Die Erstellung einer optisch ansprechenden und realistisch machbaren 3D-Modellierung einer Spiralgalaxie in Python, deren Struktur von Anfang an im Code festgelegt ist.

2. Vergleich der Genauigkeit (hinsichtlich Energieerhaltung) dreier numerischer Verfahren, die ich, entsprechend ihrer Implementierungsschwierigkeit, nacheinander im Projektverlauf umgesetzt habe.

3. Die Untersuchung, ob sich eine spiralförmige oder scheibenförmige Struktur aus Partikeln bildet, wenn diese zufällig um ein Zentrum verteilt sind und eine bestimmte Rotationsgeschwindigkeit vorgegeben ist.

2 Projektverlauf

Zu Beginn war die Fülle an Informationen, insbesondere im Bereich der Differentialgleichungen, überfordernd für mich. Da ich keinerlei Vorkenntnisse in diesem Bereich hatte und in meinem Mathematikmodul lediglich Lineare Algebra behandelt wurde, stellten die Differentialgleichungen eine besondere Herausforderung dar. Hinzu kam, dass viele der online verfügbaren Artikel die Informationen auf eine sehr komplexe und wenig anfängerfreundliche Weise präsentierten, was das Verständnis zusätzlich schwieriger machte.

Anfangs hatte ich keinen konkreten Plan, was ich genau erreichen wollte. Erst im Laufe der Projektarbeit konkretisierten sich meine Ziele, was auch in der Projektplanung erkennbar ist. So habe ich beispielsweise erst in der Mitte der Projektarbeit festgelegt, den Fokus auf den Vergleich numerischer Verfahren sowie die Untersuchung des Verhaltens von Partikeln ohne festgelegte Spiralstruktur zu legen.

Darüber hinaus habe ich einige Ideen verworfen, da sie den vorgegebenen Zeitrahmen gesprengt hätten. Dazu gehörte die Idee, optische Verbesserungen mithilfe von Shadern umzusetzen sowie den Algorithmus der Kraftberechnung durch eine effizientere Variante wie cKDTree (mit einer Komplexität von $O(n \log(n))$ statt $O(n^2)$) zu ersetzen.

3 Theorie

3.1 Gravitation

Die Gravitation ist eine fundamentale Kraft, die für die Simulation von N-Körpersystemen, wie den Bahnen von Planeten oder auch den Bewegungen in Galaxien, von entscheidender Bedeutung ist. Das Newtonsche Gravitationsgesetz lautet:

$$F = G \cdot \frac{m_1 m_2}{r^2}$$

In einem System mit beliebig vielen Körpern lässt sich die vektorielle Gravitationskraft auf einen einzelnen Körper berechnen, indem man die Gravitationskräfte aller anderen Körper auf ihn summiert:[1]

$$F_i = G \sum_{\substack{j=1 \\ j \neq i}}^{N} \frac{m_i m_j}{|r_j - r_i|^3} (r_j - r_i)$$

3.2 N-Körperproblem

Das N-Körperproblem beschreibt die Schwierigkeit, die Trajektorien (Bahnkurven) von beliebig vielen massebehafteten Körpern in einem System vorherzusagen, da jeder Körper durch die Gravitation von allen anderen Körpern beeinflusst wird. [2]Besonders herausfordernd ist die Berücksichtigung der geringeren gravitativen Einflüsse zwischen weit entfernten Körpern. Die Dynamik des Systems lässt sich durch ein System von Differentialgleichungen zusammen mit einem Anfangswertproblem beschreiben, wobei eine analytische Lösung nur für den speziellen Fall eines Zwei-Körper-Systems existiert, bei dem sich die Bewegungen der Körper in einfachen Bahnen, z.B. Ellipsen, darstellen lassen. Für Systeme mit mehr als zwei Körpern sind nur numerische Lösungsverfahren verfügbar, die eine Annäherung an die tatsächlichen Trajektorien liefern.

3.3 Differentialgleichungen

Eine Differentialgleichung ist eine Gleichung, in der neben einer Funktion auch ihre Ableitungen erscheinen und deren Lösung eine Funktion ist, die diese Gleichung erfüllt.[5] Sie ist gegeben durch

$$\frac{d}{dt} x(t) = f(t, x(t))$$

[4, S. 385] Die Lösungstrajektorie $x(t)$ gibt dann an, wie die Lösung ist. Im Fall eines N-Körperproblems wird die Bewegungsgleichung durch das 2. Newtonsche Gesetz

[1]Die Formel leitet sich aus dem Gravitationsgesetz her, wenn man zur vektoriellen Darstellung den Einheitsvektor multipliziert, um auch die Richtung der Kraft anzugeben.

beschrieben:

$$F = ma \Rightarrow a = \frac{F}{m}$$
$$\Leftrightarrow \frac{d^2}{dt^2}x(t) = \frac{F(t, x(t))}{m}$$

Diese Newtonsche Bewegungsgleichung entspricht einer Differentialgleichung 2.Ordnung, weil die zweite Ableitung der Position $x(t)$ (die Beschleunigung a) eine Rolle spielt. Wir wandeln diese Differentialgleichung in ein System von Differentialgleichungen 1.Ordnung um, da wir so numerische Verfahren zur Lösung nutzen können, die nur für DGL 1.Ordnung definiert sind

$$\frac{d}{dt}x(t) = v(t) \tag{1}$$
$$\frac{d}{dt}v(t) = \frac{F(t, x(t))}{m} \tag{2}$$

3.4 Lösungsmöglichkeiten der Differentialgleichungen

3.4.1 Explizites Euler-Verfahren

Die einfachste Möglichkeit eine gewöhnliche Differentialgleichung zu lösen, ist das Euler-Verfahren.
Wir unterteilen hierfür die Zeit in kleine Schritte der Größe Δt. Die diskreten Zeitpunkte sind dann:

$$t_n = t_0 + n \cdot \Delta t, n = 1, 2, 3, ...$$

Die Ableitung $\frac{d}{dt}x(t)$ kann durch den Differenzenquotienten angenähert werden:[4, S. 392]

$$\frac{d}{dt}x(t) \approx \frac{x(t_{n+1}) - x(t_n)}{\Delta t}$$

Hierbei wird die Annäherung genauer, wenn Δt kleiner wird.
Setzen wir dies nun in die allgemeine Form einer Differentialgleichung ein, erhalten wir:

$$\frac{x(t_{n+1}) - x(t_n)}{\Delta t} \approx f(t_n, x(t_n))$$

Stellen wir nun nach $x(t_{n+1})$ um: [3, S. 8]

$$x(t_{n+1}) = x(t_n) + \Delta t \cdot f(t_n, x(t_n))$$

Besitzen wir einen Anfangswert x_0 zum Zeitpunkt t_0, so können wir also die nachfolgenden Werte approximiert berechnen durch:

$$x(t_{n+1}) = x(t_n) + \Delta t \cdot f(t_n, x(t_n)), \text{wobei } n = 1, 2, 3, ...$$

Wenden wir dies nun auf unsere Gleichungen (1) und (2) an, so erhalten wir:

$$x(n+1) = x_n + \Delta t \cdot v(n)$$
$$v(n+1) = v_n + \Delta t \cdot \frac{F_n}{m}$$

wobei $F_n = F(t, x(t))$ die Kraft zum Zeitpunkt t ist.
Das Euler-Verfahren ist ein numerisches Verfahren **1.Ordnung**, das heißt globale Fehler sind proportional zu Δt (lokale Fehler zu Δt^2). Dies bedeutet, dass wenn die Schrittweite Δt halbiert wird, die Genauigkeit um den Faktor 2 steigt.

3.4.2 Runge-Kutta-Verfahren 4.Ordnung

Wie beim Euler-verfahren wird auch beim RK4 der Ansatz der Differenzenfunktion für eine Näherung der Ableitung verwendet. Allerdings berechnet es die Steigung nicht nur an einem Punkt, sondern nutzt mehrere Zwischenschritte, sogenannte Stützstellen, wodurch die Näherung deutlich genauer wird.[4, ab S.397] Die allgemeine Formel lautet:

$$x_{n+1} = x_n + \Delta t \cdot \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

wobei

$$k_1 = f(t_n, x_n)$$
$$k_2 = f(t_n + \frac{\Delta t}{2}, x_n + \frac{\Delta t}{2} k_1)$$
$$k_3 = f(t_n + \frac{\Delta t}{2}, x_n + \frac{\Delta t}{2} k_2)$$
$$k_4 = f(t_n + \Delta t, x_n + \Delta t k_3)$$

Der globale Fehler des Runge-Kutta-Verfahrens 4.Ordnung ist proportional zu Δt^4

3.4.3 Leap-frog-Verfahren

Das Leapfrogverfahren ist im Gegensatz zum Euler-Verfahren ein Verfahren zur Lösung von DGLen 2.Ordnung [6], also im Falle der Newtonschen Bewegungsgleichung:

$$\frac{d^2}{dt^2} x(t) = \frac{F(t, x(t))}{m}$$

das, wie oben bereits gezeigt,in ein System von DGLen 1.Ordnung umgewandelt werden kann. Der Ansatz des Leapfrog-Verfahrens besteht darin, dass Positionen und Geschwindigkeiten um einen halben Zeitschritt versetzt aktualisiert werden.[7]

$$v_{n+0.5} = v_{n-0.5} + \Delta t \cdot \frac{F(t, x(t))}{m}$$
$$x_{n+1} = x_n + \Delta t \cdot v_{n+0.5}$$

Es hat eine globale Fehleranfälligkeit proportional zu Δt^2, was es zu einem Verfahren 2.Ordnung macht.

3.5 Spiralstruktur

Um die Spiralstruktur zu erzeugen, nutzte ich die Formeln für eine logarithmische Spirale. Der Radius eines jeden Punktes in der Spirale, der den Abstand zum Zentrum bildet, berechnet sich folgendermaßen:[1]

$$r = a \cdot e^{b \cdot \theta}$$

wobei a der Startabstand vom Zentrum, b die Wachstumsrate der Spirale und θ der Winkel in Bogenmaß ist.
Nun lässt sich mithilfe des Radius die x-und y-Koordinate eines jeden Partikels berechnen:

$$x = r \cdot \cos(\theta)$$
$$y = r \cdot \sin(\theta)$$

4 Codeanalyse

Im Folgenden möchte die einzelnen Codebestandteile und Funktionen erörtern.

4.1 Spiralsimulation

4.1.1 Funktion:Spiralstruktur

```
    def spirale_mehrere_arme(a, b, n, arme, breite=4):
      punkte_pro_arm = n // arme
      orte = []

      for arm in range(arme):
        theta = np.linspace(0, 4 * np.pi, punkte_pro_arm)
        r = a * np.exp(b * theta)

        r_streuung = r * np.random.uniform(1 - breite * 0.05, 1 + breite * 0.05, size
=punkte_pro_arm)
        theta_streuung = theta + np.random.uniform(-0.15, 0.15, size=punkte_pro_arm)

        versatz = (2 * np.pi / arme) * arm
        x = r_streuung * np.cos(theta_streuung + versatz)
        y = r_streuung * np.sin(theta_streuung + versatz)
        z = np.random.normal(0, 0.5, size=punkte_pro_arm)

        orte.append(np.column_stack((x, y, z)))

      return np.vstack(orte)

```

Diese Funktion berechnet die Startposition beliebig vieler Partikel in Form einer logarithmischen Spirale und nimmt dabei als Argumente neben den bereits in dem Abschnitt Spiralstruktur genannten Variablen für die Formel einer logarithmischen Spirale, zusätzlich die Variable *arme*, die festlegt, wie viele Arme die Spirale hat. Zusätzlich werden zufällige Streuungen mit *np.random* hinzugefügt, um die Spirale realistischer zu gestalten. Die Partikel werden mit $punkte\_pro\_arm = n \;//\; arme$ gleichmäßig auf

die Arme verteilt. Ansonsten erfolgt die Berechnung der x-und y-Koordinaten, wie bereits im Theorieabschnitt erwähnt, während z in einer normalverteilten Zufallsgröße mit *np.random.normal* festgelegt ist.

4.1.2 Berechnung der Gesamtenergie im System

```
    def Energie_kin(massen,geschw):
      return 0.5 * np.sum(massen*np.sum(geschw**2, axis=1))

    def Energie_pot(orte,massen,G):
      E_pot = 0
      for i in range(len(orte)):
        for j in range(i + 1, len(orte)):
          r = np.linalg.norm(orte[j] - orte[i])
          if r > 0:
            E_pot -= G * massen[i] * massen[j] / r
      return E_pot

    def Energie_gesamt(orte, massen, G, geschw):
      E_kin = Energie_kin(massen, geschw)
      E_pot = Energie_pot(orte, massen, G)
      return E_kin + E_pot

```

Dieser Codeabschnitt dient der Berechnung der gesamten Energie im System, die aus kinetischer und potentieller Energie besteht. Für die kinetische Energie wird *np.sum* genutzt, sowohl um die gesamte Geschwindigkeit eines Partikels mit x,y und z Komponente zu berechnen, als auch, um die Energie eines jeden Partikels aufzusummieren.
Die potentielle Energie wird mittels einer Doppelschleife berechnet und folgt dem Newtonschen Gravitationsgesetz. Sie ist negativ, da in einem Gravitationsfeld der Nullpunkt der potentiellen Energie typischerweise mit einem unendlichen Radius festgelegt wird.
Die Gesamtenergie wird dann entsprechend aus der Summe von der potentiellen und der kinetischen Energie berechnet.

4.1.3 Berechnung der Kreisbahngeschwindigkeit

```
  for i in range(len(orte) - 1):
    r = orte[i] - orte[-1]  # Vektor r von Objekt i zum Zentrum (zentrale Masse)
    distanz = np.linalg.norm(r)
    if distanz > 0:
      einheitsvektor_r = r / distanz
      tangential = np.cross(einheitsvektor_r, np.array([0, 0, 1]))
      tangential /= np.linalg.norm(tangential)
      geschwindigkeit = np.sqrt((G*massen[-1])/distanz)
      geschw[i] = geschwindigkeit * tangential

```

Diese Schleife berechnet für jeden einzelnen Partikel die Tangentialgeschwindigkeit (Kreisbahngeschwindigkeit) um das Zentrum herum. Hierfür wird das Kreuzprodukt mit *np.cross* genutzt, damit die Geschwindigkeit in Richtung der x-y-Achse verläuft. Diese Tangentialgeschwindigkeit wird anschließend normiert und mit der Kreisbahngeschwindigkeit multipliziert. Diese ergibt sich aus der Definition der Zentrifugalkraft, damit ein Kräftegleichgewicht entsteht:[8]

$$v = \sqrt{\frac{GM}{r}}$$

4.1.4 Kraftberechnung

```
    def kraftrechnung(orte, massen):
      kraefte = np.zeros_like(orte)
      for i in range(len(orte)):
        for j in range(len(orte)):
          if i != j:
            r = orte[j] - orte[i]
            distanz = np.linalg.norm(r)
            if distanz > 0:
              Kraft = G * ((massen[i] * massen[j]) / distanz ** 2)
              kraefte[i] += Kraft * r / distanz
      return kraefte

```

Diese Funktion berechnet für jeden Partikel im System die Summe der auf ihn wirkenden Gravitation der anderen Partikel. Am Anfang wird mit *np.zeros_like(orte)* ein array erstellt, das dieselbe Form wie das array *orte* hat und mit Nullen gefüllt ist. Innerhalb der Doppelschleife wird für jeden einzelnen Eintrag die Null durch die entsprechend aufsummierte Gravitationskraft ersetzt. Die Berechnung erfolgt nach dem Newtonschen Gravitationsgesetz, wie in Abschnitt 3.1 bereits gezeigt.

4.1.5 Die verschiedenen numerischen Verfahren

Euler-Verfahren :

```
    def simulate(orte, geschw, massen, dt):
      global verlauf
      verlauf = [orte.copy()]
      for _ in range(1500):
        kraefte = kraftrechnung(orte, massen)
        beschleunigung = kraefte / massen[:,np.newaxis]
        geschw += beschleunigung * dt
        orte += geschw * dt
        verlauf.append(orte.copy())
      return np.array(verlauf)

```

Diese Funktion nutzt das Euler-Verfahren zur Berechnung der Trajektorien (Bahnkurve) der Partikel. Der Algorithmus fußt auf der for-Schleife, die in einer beliebig wählbaren Schrittzahl, in diesem Fall 1500, zuerst mit der Funktion *kraftrechnung* die Kräfte als array berechnet und anschließend die Beschleunigung ermittelt. Die anschließende angenäherte rekursive Berechnung der neuen Position und Geschwindigkeiten erfolgt wie in Abschnitt 3.4.1 bereits erörtert.

Runge-Kutta-Verfahren 4.Ordnung :

```
    def simulate(orte, geschw, massen, dt):
      global verlauf
      verlauf = [orte.copy()]

      def beschleunigung(orte, massen):
        kraefte = kraftrechnung(orte, massen)
        return kraefte / massen[:, np.newaxis]

      for _ in range(1500):
        # Erste Stufe
        a1 = beschleunigung(orte, massen)
```

```
        v1 = geschw

        # Zweite Stufe
        v2 = geschw + a1 * (dt / 2)
        x2 = orte + v1 * (dt / 2)
        a2 = beschleunigung(x2, massen)

        # Dritte Stufe
        v3 = geschw + a2 * (dt / 2)
        x3 = orte + v2 * (dt / 2)
        a3 = beschleunigung(x3, massen)

        # Vierte Stufe
        v4 = geschw + a3 * dt
        x4 = orte + v3 * dt
        a4 = beschleunigung(x4, massen)

        orte += (dt / 6) * (v1 + 2 * v2 + 2 * v3 + v4)
        geschw += (dt / 6) * (a1 + 2 * a2 + 2 * a3 + a4)

        verlauf.append(orte.copy())

    return np.array(verlauf)

```

Hier erfolgt derselbe Ansatz wie beim Euler-Verfahren mit dem array *verlauf*, das alle in den Zeitschritten berechneten neuen Positionen speichert. Für die Berechnung mittels Runge-Kutta-Verfahren möchte ich auf Abschnitt 3.4.2 verweisen.

Leapfrog-Verfahren :

```
def simulate(orte, geschw, massen, dt):
    global verlauf
    verlauf = [orte.copy()]

    for _ in range(1500):
        # Berechne die Beschleunigung zur aktuellen Zeit
        kraefte = kraftrechnung(orte, massen)
        a = kraefte / massen[:, np.newaxis]

        # Leapfrog: Geschwindigkeit um halben Zeitschritt updaten
        geschw_halb = geschw + 0.5 * a * dt

        # Positionen mit der halb-geschobenen Geschwindigkeit updaten
        orte += geschw_halb * dt

        kraefte = kraftrechnung(orte, massen)
        a_neu = kraefte / massen[:, np.newaxis]

        geschw = geschw_halb + 0.5 * a_neu * dt

        verlauf.append(orte.copy())

    return np.array(verlauf)

```

Auch für die Leapfrog-Variante habe ich denselben Ansatz genutzt und die Berechnung wie in Abschnitt 3.4.3 vollzogen. Allerdings umgehe ich im Code das Problem, dass man für den Anfangswert $v_{-0.5}$ benötigt. Dadurch wird die Implementierung einfacher als das klassische Leapfrog-Verfahren und trotzdem bietet es die gleiche Genauigkeit.

4.1.6 Visualisierung

Für die Visualisierung habe ich matplotlib.animation.FuncAnimation genutzt:

```
    def init():
      partikel._offsets3d = ([], [], [])
      zentrum._offsets3d = ([],[],[])
      return partikel, zentrum, nebel

    # Update-Funktion
    def update(u):
      orte = x[u]
      partikel._offsets3d = (orte[:-1, 0], orte[:-1, 1], orte[:-1, 2])  #orte[:,0]
    ist die x-Koordinate, orte[:,1] die y-Koordinate usw.
      zentrum._offsets3d = (orte[-1:, 0], orte[-1:, 1], orte[-1:, 2])
      return partikel, zentrum

    ani = animation.FuncAnimation(fig, update, frames=range(0, len(x), 10), init_func
    =init, interval=50, repeat=True)

```

Hierbei fungiert die init-Funktion dazu, die Partikel, das Zentrum und die Nebelpartikel an matplotlib zu übergeben.
Update() aktualisiert sowohl die Partikel- als auch die Zentrumspostion für den Zeitschritt u. Die Nebelpartikel werden nicht animiert, da der Rechenaufwand hierfür viel zu groß wäre.
Schließlich wird die Animation mit FuncAnimation gestartet.

4.2 Simulation ohne Spirale

Die grundlegenden Funktionen, bis auf die Spiralstrukturfunktion, bleiben gleich.

4.2.1 zufällige Anordnung

```
    def scheibenverteilung(radius, n, hoehe=40.0):
      r = radius * np.sqrt(np.random.rand(n))
      theta = np.random.uniform(0, 2 * np.pi, n)
      x = r * np.cos(theta)
      y = r * np.sin(theta)
      z = np.random.uniform(-hoehe, hoehe, n)

      return np.column_stack((x, y, z))

```

Diese Funktion positioniert die Partikel in einer zufälligen zylinderförmigen Verteilung um das Zentrum, wobei die vertikale Ausdehnung beliebig verändert werden kann.

5 Projektergebnisse

5.1 Optisch ansprechende Simulation

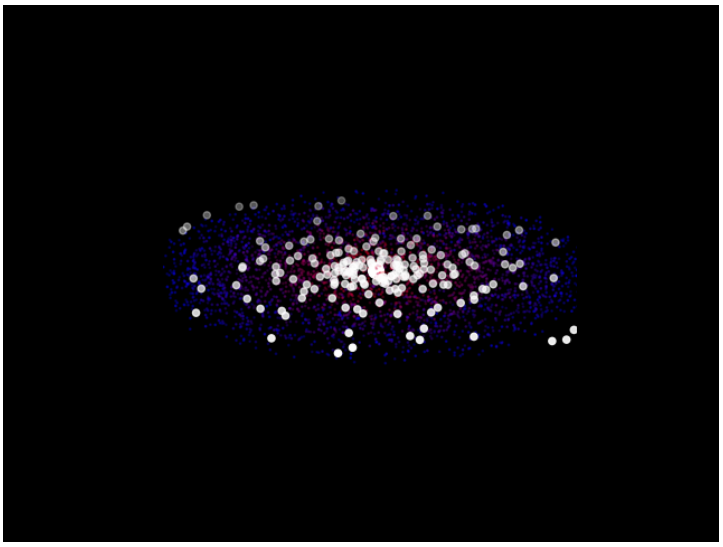

Abbildung 1: Resultat des Codes mit 200 Partikeln und 2 Armen

Man sieht deutlich die Spiralstruktur, aber auch die Unregelmäßigkeiten, welche durch die zufällige Streuung einhergehen. Die Nebelpartikel sorgen ebenfalls für eine optisch ansprechende 3D-Grafik.

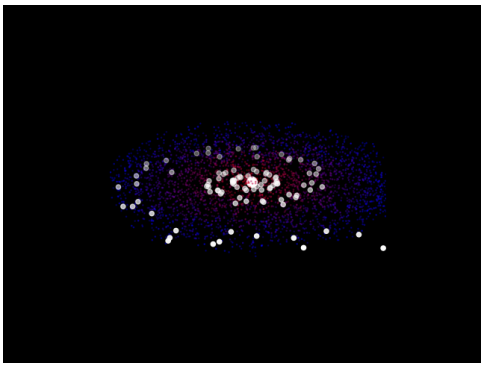

Abbildung 2: Resultat des Codes mit 100 Partikeln und 1 Arm

5.2 Vergleich der numerischen Verfahren

Die Visualisierung der Energieerhaltung aller drei getesteten numerischen Verfahren ergab mit $dt = 0.01$ folgende Resultate: Offensichtlich sieht man, dass das Euler-Verfahren die schlechteste numerische Stabilität liefert. Runge-Kutta- sowie Leapfrog-Verfahren liefern deutlich bessere Ergebnisse, wobei das Leapfrog-Verfahren noch ein Stück besser ist,

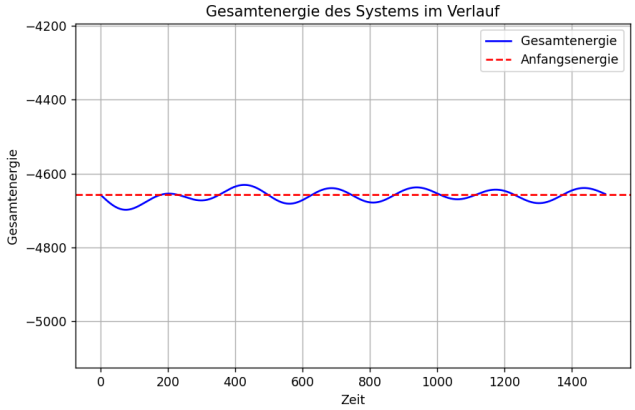

Abbildung 3: Energieerhaltung mit Euler-Verfahren

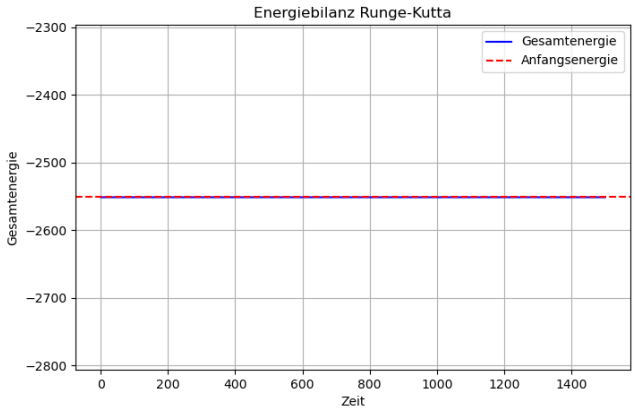

Abbildung 4: Energieerhaltung mit Runge-Kutta-Verfahren 4.Ordnung

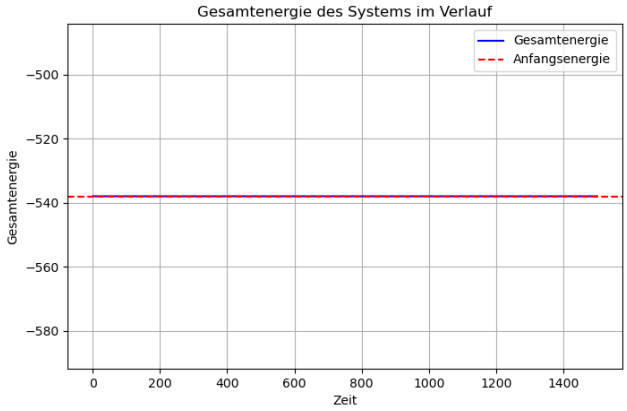

Abbildung 5: Energieerhaltung mit Leapfrog-Verfahren

was auf die symplektische Natur dieses Integrators zurückzuführen ist. Insgesamt liefern jedoch alle Verfahren physikalisch korrekte Ergebnisse, die ab einem Zeitintervall von $dt = 0.01$ stabile Ergebnisse liefern. Zudem sieht man anhand der negativen Gesamtenergie, dass die Rotation der Spiralarme gebunden ist, womit das Ziel stabiler Bahnen ebenfalls erreicht wurde.

5.3 Selbstorganisation des Systems ohne vorgegebene Spiralstruktur

Ich habe abschließend getestet, was geschieht, wenn ich im Code keine Spiralstruktur vorgebe, sondern die Partikel zylinderförmig mit zufälligen Positionen anordnen lasse. Formieren sich die Partikel zu einer rotierenden Scheibe oder bilden sie sogar eine Art Spiralstruktur? Das Resultat ist, dass sich die Partikel im Laufe der Simulation schrittweise scheibenförmig anordnen und die durschnittliche vertikale Ausdehnung abnimmt. Der Grund für die scheibenförmige Anordnung der Partikel ist die Tangentialgeschwin-

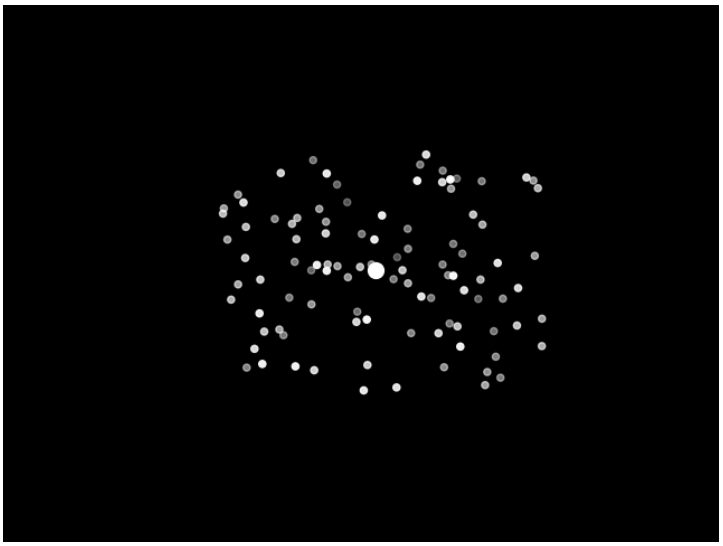

Abbildung 6: Partikel am Anfang der Simulation

digkeit in xy-Richtung und die gleichzeitige Gravitationskraft, die vom Zentrum aus wirkt. Durch die Berücksichtigung der Zentrifugalkraft bleiben die Partikel stabil und kollabieren nicht.
Allerdings entsteht keine Spiralstruktur: Hierfür sind weit komplexere Bedingungen notwendig, die ich in dieser Simulation nicht berücksichtigt habe. Dazu zählen Dichtewellen, Differentialrotation und gravitative Beeinflussungen anderer Körper.

Abbildung 7: Partikel am Ende der Simulation

6 Fazit und Ausblick

Ich bin mit den Ergebnissen sehr zufrieden und bin der Meinung, dass für die (bedingt durch wenig Erfahrung) limitierten Ressourcen, die ich zur Verfügung hatte, ein optisch sehr schönes Modell einer Galaxie zustande gekommen ist. Auch die Energieerhaltung hat sehr gut funktioniert, und auch meine Runge-Kutta-Variante liefert letzten Endes stabile Simulationen, obwohl ich leider bis fast zum Schluss Fehler im Code hatte.
Natürlich ist mir bewusst, dass diese Simulation meilenweit von einer physikalisch realistischen Modellierung entfernt ist. Dort spielen nämlich noch viel theoretischere und kompliziertere Theorien wie Relativitätstheorie und Dunkle Materie bzw. Energie eine Rolle, die ich nicht berücksichtigen konnte. Eine zukünftige Fortführung dieses Projektes könnte versuchen ein Dunkle-Materie-Modell oder einen einfacheren Ersatz dafür zu implementieren, um so auch die Rotationskurve einer Galaxie zu berücksichtigen. So würde man auch Beobachtungen neben der klassischen Newtonschen Mechanik berücksichtigen.

Literatur

[1] *Logarithmic Spiral – from Wolfram MathWorld*. URL: https://mathworld.wolfram.com/LogarithmicSpiral.html (besucht am 12. 03. 2025).

[2] *N-Körper-Problem – Wikipedia*. URL: https://de.wikipedia.org/wiki/N-K%C3%B6rper-Problem (besucht am 10. 03. 2025).

[3] Mikael Sahrling. *Analog Circuit Simulators for Integrated Circuit Designers: Numerical Recipes in Python*. Cham: Springer International Publishing, 2021. ISBN: 978-3-030-64205-1 978-3-030-64206-8. DOI: 10.1007/978-3-030-64206-8. URL: https://link.springer.com/10.1007/978-3-030-64206-8 (besucht am 12. 03. 2025).

[4] David E. Stewart. *Numerical Analysis: A Graduate Course*. Bd. 4. CMS/CAIMS Books in Mathematics. Cham: Springer International Publishing, 2022. ISBN: 978-3-031-08120-0 978-3-031-08121-7. DOI: 10.1007/978-3-031-08121-7. URL: https://link.springer.com/10.1007/978-3-031-08121-7 (besucht am 12. 03. 2025).

[5] Wilfried Weißgerber. *Mathematik zu Elektrotechnik für Ingenieure: Lehr- und Arbeitsbuch für das Grundstudium*. Wiesbaden: Springer Fachmedien Wiesbaden, 2023. ISBN: 978-3-658-40836-7 978-3-658-40837-4. DOI: 10.1007/978-3-658-40837-4. URL: https://link.springer.com/10.1007/978-3-658-40837-4 (besucht am 08. 03. 2025).

[6] Dr H Wolff. „Eine einfache Schrittweitensteuerung für das Leapfrog-Verfahren“. In: (). URL: https://hwolff.hier-im-netz.de/.cm4all/uproc.php/0/docs/Leapfrog-Schrittweitensteuerung.pdf.

[7] *ws1819:sternensystem [Mathesis Wiki]*. URL: https://www.mintgruen.tu-berlin.de/mathesisWiki/doku.php?id=ws1819:sternensystem (besucht am 12. 03. 2025).

[8] *Zentrifugalkraft – Wikipedia*. URL: https://de.wikipedia.org/wiki/Zentrifugalkraft (besucht am 10. 03. 2025).