Università di Cagliari - Corso di Laurea in Fisica - Corso di Fondamenti di Fisica Computazionale - AA 2023/24

Mirco Aresu 607365283

4 maggio 2024

Assignment n.1 – ODE: problema planetario a due corpi

-

0.1 1a

Simulate le traiettoria di Venere nel campo gravitazionale del Sole includendo l'interazione gravitazionale tra i due pianeti.

Introduzione

Il sistema solare è un affascinante sistema planetario che comprende una vasta varietà di corpi celesti che orbitano attorno al Sole. Uno dei fenomeni più interessanti è rappresentato dalle orbite dei pianeti, che sono governate dalle leggi della gravitazione universale di Newton. In questa simulazione, ci concentreremo sull'orbita di Venere, il secondo pianeta del sistema solare in ordine di distanza dal Sole. Concentrandoci esclusivamente sull'interazione gravitazionale tra il Sole e Venere, utilizzeremo la simulazione numerica per predire e tracciare l'orbita di Venere rispetto al sole, esaminando l'effetto della forza gravitazionale del Sole sulla traiettoria orbitale di Venere. Possiamo farci una prima idea di come Venere orbita rispetto agli altri pianeti con il tool della

NASA: NASA Solar System Dynamics. [Orbita di Venere Attorno al Sole]: Un famoso esempio del mondo reale di un sistema a due corpi è forse l'orbita di Venere attorno al Sole. Anche se il nostro sistema solare è complesso, consideriamo Venere e il Sole come un sistema a due corpi semplificato. Venere è il secondo pianeta dal Sole e la sua orbita ellittica ma è abbastanza vicina a essere un cerchio.

Dati Astronomici:

- Massa di Venere (M_{Venere}): Utilizzeremo il valore della massa di Venere, pari a $M_{\text{Venere}} = 4.8675 \times 10^{24}$ kg.
- Massa del Sole ($M_{\odot}$): Utilizzeremo il valore della massa del Sole, pari a $M_{\odot} = 1.989 \times 10^{30}$ kg.
- Semi-Maggior Asse di Venere (a): Il semi-maggior asse dell'orbita di Venere è di circa 0.723 Unità Astronomiche (AU), dove 1 AU corrisponde a 149597870700 metri.
- Velocità Orbitale di Venere (v_v): La velocità orbitale di Venere è di 35.02 km/s, corrispondente a 7.39 AU/anno.
- Periodo Orbitale di Venere (T): Il periodo orbitale di Venere è di 0.615 anni.
- G (Costante Gravitazionale Universale): Utilizzeremo il valore della costante gravitazionale, $G = 6.674 \times 10^{-11}$ m^3 kg^{-1} s^{-2}.

Legge di Gravitazione di Newton

La legge di gravitazione di Newton afferma che due corpi puntiformi si attraggono con una forza gravitazionale proporzionale al prodotto delle loro masse e inversamente proporzionale al quadrato della distanza tra di essi. Questo principio è espresso dall'equazione:

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

dove:

- $\mathbf{F}$ è la forza gravitazionale tra i due corpi,
- G è la costante gravitazionale universale,

– m_1 e m_2 sono le masse dei due corpi,

– r è la distanza tra i due corpi,

– $\hat{\mathbf{r}}$ è il versore diretto lungo la linea che congiunge i due corpi.

Equazione del moto

Dopo aver applicato la seconda legge di Newton al problema planetario a due corpi, possiamo scrivere le equazioni del moto per un corpo di massa m_i soggetto alla forza gravitazionale esercitata da un altro corpo di massa m_j:

$$\frac{d^2\vec{r}_i}{dt^2} = -\frac{Gm_j(\vec{r}_i - \vec{r}_j)}{|\vec{r}_{ij}|^3}$$

dove $\vec{r}_i$ e $\vec{r}_j$ sono le posizioni vettoriali dei due corpi e $\vec{r}_{ij} = \vec{r}_i - \vec{r}_j$ è il vettore che va dal corpo j al corpo i. La G è la costante di gravitazione universale.

Per il nostro caso, considerando Venere ($m_i = M_{\text{Venere}}$) e il Sole ($m_j = M_\odot$), le equazioni diventano:

$$\frac{d^2\vec{r}_{\text{Venere}}}{dt^2} = -\frac{GM_\odot(\vec{r}_{\text{Venere}} - \vec{r}_\odot)}{|\vec{r}_{\text{Venere}\odot}|^3}$$

Assumendo che il Sole sia fermo all'origine del nostro sistema di riferimento, possiamo semplificare ulteriormente la relazione come segue:

$$\frac{d^2\vec{r}_{\text{Venere}}}{dt^2} = -\frac{GM_\odot\vec{r}_{\text{Venere}}}{|\vec{r}_{\text{Venere}}|^3}$$

Le equazioni differenziali di secondo ordine possono essere scomposte in due equazioni del primo ordine, una per la velocità e una per la posizione:

$$\frac{d\vec{v}_{\text{Venere}}}{dt} = -\frac{GM_\odot\vec{r}_{\text{Venere}}}{|\vec{r}_{\text{Venere}}|^3}$$
$$\frac{d\vec{r}_{\text{Venere}}}{dt} = \vec{v}_{\text{Venere}}$$

Queste equazioni descrivono come la posizione e la velocità di Venere cambiano nel tempo sotto l'influenza della gravità del Sole.

Centro di Massa

Un altro concetto utile da tenere a mente quando si modellano sistemi planetari è il centro di massa del sistema. Il centro di massa è il punto dove la somma dei momenti di massa del sistema è zero, ossia il punto in cui la massa totale del sistema può essere considerata concentrata e attorno al quale il sistema può ruotare in equilibrio.

La formula per trovare il centro di massa di un sistema e la sua velocità è piuttosto semplice e si basa sulla media pesata delle posizioni e delle velocità, pesate rispetto alle masse dei corpi:

$$\vec{r}_{cdm} = \frac{\sum_{i=1}^{n} m_i \vec{r}_i}{\sum_{i=1}^{n} m_i}$$

$$\vec{v}_{cdm} = \frac{\sum_{i=1}^{n} m_i \vec{v}_i}{\sum_{i=1}^{n} m_i}$$

dove:

- $\vec{r}_{cdm}$ è il vettore posizione del centro di massa,
- $\vec{v}_{cdm}$ è il vettore velocità del centro di massa,
- m_i è la massa dell'i-esimo corpo,
- $\vec{r}_i$ è il vettore posizione dell'i-esimo corpo,
- $\vec{v}_i$ è il vettore velocità dell'i-esimo corpo.

Questo ci predispone per un problema a più corpi ma prima di modellare un sistema a tre corpi, modelliamo un sistema a due corpi per osservare il suo comportamento e poi estendiamo il codice per lavorare con tre corpi.

Adimensionalizzazione

Prima di risolvere le equazioni del movimento, è necessario adimensionalizzare il sistema. Ciò significa convertire tutte le quantità che hanno dimensioni (come posizione, velocità, massa, ecc.) in quantità adimensionali con magnitudini vicine all'unità. I motivi per fare ciò sono:

- Nelle equazioni differenziali, termini differenti possono avere ordini di grandezza molto diversi (oltre 10^{24}). Una tale disparità può portare a una lenta convergenza dei metodi numerici.
- Se la magnitudine di tutti i termini diventa vicina all'unità, tutti i calcoli diventeranno meno costosi dal punto di vista computazionale.
- Otteniamo un punto di riferimento rispetto alla scala. Ad esempio, una massa espressa come un multiplo della massa del Sole è più intuitiva rispetto a un numero in kg.

Per adimensionalizzare le equazioni, ogni quantità viene divisa per una quantità di riferimento fissa. Per esempio, dividete i termini di massa per la massa del Sole, i termini di posizione per la distanza media tra il Sole e Venere, i termini di tempo per il periodo orbitale di Venere e i termini di velocità per la velocità orbitale media di Venere.

Quando dividiamo ogni termine per la quantità di riferimento, dobbiamo anche moltiplicarlo per evitare di cambiare l'equazione. Tutti questi termini insieme alla costante G possono essere raggruppati in una costante, diciamo K, per la prima equazione e K' per la seconda. Quindi, le equazioni adimensionalizzate sono come segue:

$$\frac{d\bar{\vec{v}}_i}{dt} = K\frac{\bar{m}_j\bar{\vec{r}}_{ij}}{\bar{r}_{ij}^3}$$

$$\frac{d\bar{\vec{r}}_i}{dt} = K'\bar{\vec{v}}_i$$

La barra sopra i termini indica che i termini sono adimensionali. Quindi queste sono le equazioni finali che useremo nella nostra simulazione.

Preparazione della Simulazione

Nel codice sottostante, implementiamo un modello semplificato che simula l'orbita di Venere intorno al Sole. Iniziamo definendo le costanti fisiche, le condizioni iniziali e le funzioni necessarie per calcolare le forze gravitazionali e le equazioni del moto. Tramite il metodo di integrazione numerica 'odeint'.

odeint: Dietro le Quinte

La funzione `odeint` è un integratore numerico utilizzato per risolvere sistemi di equazioni differenziali ordinarie (ODEs). È basato sui metodi numerici inizialmente sviluppati nel pacchetto FORTRAN chiamato ODEPACK. `odeint` è ampiamente utilizzato in campo scientifico e ingegneristico per la sua robustezza e efficacia nel trattare una vasta gamma di problemi.

Equazioni Differenziali Ordinarie (ODEs)

Le ODEs descrivono il cambiamento delle variabili di stato in funzione di una singola variabile indipendente, tipicamente il tempo. Matematicamente, un'ODE può essere espressa nella forma:

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

dove $\mathbf{y}$ è il vettore di stato del sistema, t è la variabile indipendente, e f è una funzione che determina le dinamiche del sistema.

Metodi Numerici per ODEs

`odeint` utilizza principalmente il metodo di integrazione chiamato LSODA (Livermore Solver for Ordinary Differential Equations with Automatic method switching for stiff and non-stiff problems), che può alternare tra due metodi numerici a seconda della "rigidità" del problema:

- **Metodo di Adams**: Usato per problemi non rigidi, utilizza un approccio predittore-correttore per stimare la soluzione dell'ODE a ogni passo.
- **Metodo BDF (Backward Differentiation Formula)**: Adatto per problemi rigidi, il BDF è stabile e efficace ma computazionalmente più intenso.

Integrazione Numerica

L'integrazione numerica di un'ODE implica l'approssimazione della soluzione a intervalli discreti. `odeint` calcola la soluzione $\mathbf{y}(t)$ a incrementi specificati di t attraverso i seguenti passaggi:

- **Inizializzazione**: Stabilire le condizioni iniziali $\mathbf{y}(t_0)$.
- **Passo di Integrazione**: Calcolare $\mathbf{y}(t + \Delta t)$ da $\mathbf{y}(t)$ usando il metodo scelto (Adams o BDF).
- **Controllo dell'Errore**: Verificare l'errore di ogni passo e aggiustare la dimensione del passo Δt per mantenere l'errore sotto una soglia specificata.
- **Iterazione**: Ripetere il passo di integrazione fino a raggiungere il valore finale di t.

Questi metodi e strategie rendono `odeint` uno strumento potente e flessibile per la simulazione numerica di sistemi dinamici complessi. Risolviamo quindi il sistema di equazioni differenziali che descrivono l'interazione gravitazionale tra Venere e il Sole. L'output della simulazione è un'orbita che rappresenta la traiettoria di Venere su un periodo approssimativo di un anno 'venusiano'.

Abbiamo inoltre introdotto il concetto di adimensionalizzazione per semplificare le equazioni e migliorare l'efficienza computazionale della simulazione. Calcoliamo poi l'eccentricità dell'orbita di Venere, che fornisce informazioni aggiuntive sul suo percorso ellittico intorno al Sole. Sebbene l'orbita di Venere sia quasi circolare, l'eccentricità ci permette di quantificare quanto l'orbita si discosti da una perfetta circonferenza.

L'eccentricità e dell'orbita di Venere è calcolata come:

$$e = \sqrt{1 - \left(\frac{b}{a}\right)^2}$$

dove a è il semiasse maggiore e b è il semiasse minore dell'orbita ellittica. Questo valore ci aiuta a capire la forma dell'orbita nel contesto della legge delle orbite di Keplero.

```python
#Assignment1.py
# %%
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

# Costanti
G = 6.67408e-11  # Costante gravitazionale universale, N-m2/kg2
M_sun = 1.989e+30  # Massa del Sole, kg
M_venus = 4.8675e+24  # Massa di Venere, kg

# Condizioni iniziali per Venere e il Sole
r_venus = np.array([108.21e9, 0])  # Posizione iniziale di Venere (vicino al perielio), m
v_venus = np.array([0, 35.02e3])  # Velocità iniziale di Venere, m/s
r_sun = np.array([0, 0])  # Il Sole rimane all'origine
v_sun = np.array([0, 0])  # Velocità iniziale del Sole

# Convertiamo i vettori di posizione in array e troviamo il baricentro
r_com = (M_venus * r_venus + M_sun * r_sun) / (M_venus + M_sun)

# Equazioni di moto per il problema a due corpi
def EquazioniDueCorpi(w, t, G, m1, m2):
    r1 = w[:2]
    r2 = w[2:4]
    v1 = w[4:6]
    v2 = w[6:8]
    r = np.linalg.norm(r2 - r1)
    dv1bydt = G * m2 * (r2 - r1) / r**3
    dv2bydt = G * m1 * (r1 - r2) / r**3
    dr1bydt = v1
    dr2bydt = v2
    derivs = np.concatenate((dr1bydt, dr2bydt,
```

```
        dv1bydt, dv2bydt))
    return derivs

# Convertiamo l'array delle condizioni iniziali in una lista
init_params = np.array([r_sun, r_venus, v_sun, v_venus])
init_params = init_params.flatten()

# Intervallo di tempo della simulazione (un anno su Venere ~ 225 giorni terrestri)
t = np.linspace(0, 225*24*3600, 1000)

# Risolviamo le equazioni differenziali
sol_due_corpi = odeint(EquazioniDueCorpi, init_params, t, args=(G, M_sun, M_venus))

# Estraiamo le posizioni di Venere e del Sole
r_sun_sol = sol_due_corpi[:, :2]
r_venus_sol = sol_due_corpi[:, 2:4]

# Calcoliamo l'estensione massima dell'orbita di Venere per impostare i limiti del grafico
max_distance = np.max(np.linalg.norm(r_venus_sol, axis=1))

# Calcoliamo l'eccentricità
a = np.max(np.linalg.norm(r_venus_sol, axis=1))  # Semiasse maggiore
b = np.min(np.linalg.norm(r_venus_sol, axis=1))  # Semiasse minore
eccentricità = np.sqrt(1 - (b**2 / a**2))

# Plottiamo le orbite di Venere e del Sole
plt.figure(figsize=(10, 10))
plt.plot(r_sun_sol[:, 0], r_sun_sol[:, 1], 'o', label='Sole', markersize=10, color='yellow', zorder=5)  # Sole come un punto giallo
plt.plot(r_venus_sol[:, 0], r_venus_sol[:, 1], label=f'Venere (Eccentricità={eccentricità:.4f})',
```

```
    color='orange')  # Orbita di Venere
plt.xlabel('Coordinate x (m)')
plt.ylabel('Coordinate y (m)')
plt.title('Visualizzazione dell\'Orbita di Venere
    Attorno al Sole')
plt.legend()
plt.grid(True)
plt.axis('equal')
plt.xlim([-max_distance*1.1, max_distance*1.1])
plt.ylim([-max_distance*1.1, max_distance*1.1])
plt.show()
```

Listing 1: Codice prima simulazione

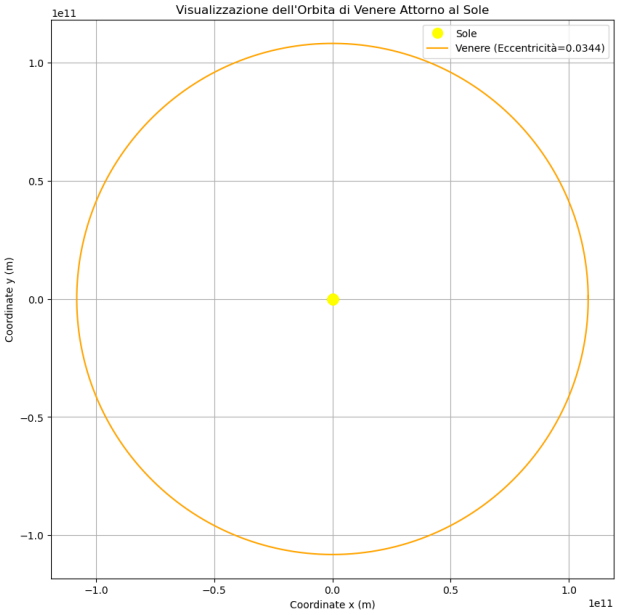

Figura 1: Orbita di Venere attorno al Sole

•

0.2 1b

Stabilite il massimo timestep che garantisca una accuratezza adeguata nella stima del periodo di rotazione.

Determinazione del Timestep

Per garantire l'accuratezza nella stima del periodo di rotazione nel nostro modello di simulazione, si possono seguire diversi approcci come l'analisi del timestep o dell'energia.

Per semplicità si può scegliere per esempio di utilizzare un timestep che sia una piccola frazione del periodo orbitale di Venere, iniziando ad esempio con 1/100 del suo periodo e valutando la stabilità dell'orbita.

Approccio Semplice: Considerando che l'orbita di Venere intorno al Sole è relativamente stabile e prevedibile, scegliamo di utilizzare un timestep fisso piccolo, come descritto sopra, per la nostra simulazione iniziale. Questo metodo è semplice da implementare e fornisce una buona stima iniziale per la maggior parte delle simulazioni orbitali.

Teoria e Implementazione

Forza Gravitazionale

La forza gravitazionale tra due corpi è descritta dalla legge di gravitazione universale di Newton. Per due corpi di massa m_1 e m_2 separati da una distanza r, la forza esercitata è proporzionale al prodotto delle loro masse e inversamente proporzionale al quadrato della loro distanza. La formula è:

$$\vec{F} = -G\frac{m_1 m_2}{\|\vec{r}\|^3}\vec{r}$$

dove G è la costante gravitazionale universale, $\vec{r}$ è il vettore che punta da un corpo all'altro, e $\|\vec{r}\|$ è la norma euclidea di $\vec{r}$, rappresentando la distanza.

Giustificazione del Termine $\|\vec{r}\|^3$

La formula classica della forza gravitazionale di Newton è espressa come:

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

dove F è la magnitudine della forza, r è la distanza scalare tra i centri di massa dei due corpi, e G è la costante gravitazionale. Tuttavia, quando trattiamo con vettori in tre dimensioni, dobbiamo considerare la direzione oltre alla magnitudine della forza.

Direzione della Forza

La forza gravitazionale agisce lungo la linea che unisce i centri di massa dei due corpi. Per incorporare la direzione nel nostro modello, utilizziamo il vettore posizione $\vec{r}$, che punta da un corpo all'altro. Per garantire che la forza abbia la direzione corretta e che sia una forza attrattiva (ossia, che i corpi si attraggano e non si respingano), moltiplichiamo la magnitudine della forza per il versore di $\vec{r}$, che è definito come $\frac{\vec{r}}{\|\vec{r}\|}$.

Normalizzazione e Uso di $\|\vec{r}\|^3$

Il versore di $\vec{r}$ garantisce che la forza sia diretta lungo $\vec{r}$, ma per incorporare questo in una formula che usa il vettore completo $\vec{r}$ piuttosto che la sua magnitudine scalare r, dobbiamo considerare il cubo della norma di $\vec{r}$. Ciò è dovuto al fatto che la formula originale divide la forza per r^2, che è la norma al quadrato di $\vec{r}$. Quando moltiplichiamo per il versore $\frac{\vec{r}}{\|\vec{r}\|}$ per ottenere la direzione, il denominatore deve essere adattato per mantenere le unità corrette e l'integrità fisica della legge. Quindi, la forza gravitazionale vettoriale diventa:

$$\vec{F} = -G\frac{m_1 m_2}{\|\vec{r}\|^3}\vec{r}$$

Questo garantisce che la magnitudine della forza sia corretta e che la sua direzione sia lungo $\vec{r}$, indicando una forza attrattiva che diminuisce con il quadrato della distanza tra i due corpi, coerentemente con la legge di gravitazione universale. **Implementazione in Python:**

```
1
2 def gravitational_force(m1, m2, r):
3     return -G * m1 * m2 / np.linalg.norm(r)**3 * r
```

Metodo di Euler

Il metodo di Eulero è un approccio diretto per integrare le equazioni del moto, usando la velocità e l'accelerazione per aggiornare la posizione e la velocità di Venere ad ogni passo temporale, lo usiamo a solo scopo didattico.

Formule teoriche:

$$\vec{r}_{\text{next}} = \vec{r} + \vec{v}\Delta t, \quad \vec{v}_{\text{next}} = \vec{v} + \frac{\vec{F}}{m}\Delta t$$

Implementazione in Python:

```
1 def euler(r, v, dt):
2     f = gravitational_force(M_sun, M_venus, r)
3     r_next = r + v * dt
4     v_next = v + f / M_venus * dt
5     return r_next, v_next
```

Metodo di Euler-Cromer

Il metodo di Euler-Cromer è una variante del metodo di Euler che aggiorna prima la velocità e poi utilizza questa velocità aggiornata per calcolare la nuova posizione. Questo metodo è spesso più stabile e accurato per certi tipi di problemi dinamici ma rimane comunque didattico.

Formule teoriche:

$$\vec{v}_{\text{next}} = \vec{v} + \frac{\vec{F}}{m}\Delta t, \quad \vec{r}_{\text{next}} = \vec{r} + \vec{v}_{\text{next}}\Delta t$$

Implementazione in Python:

```python
def euler_cromer(r, v, dt):
    f = gravitational_force(M_sun, M_venus, r)
    v_next = v + f / M_venus * dt
    r_next = r + v_next * dt
    return r_next, v_next
```

Metodo di Verlet

Il metodo di Verlet è comunemente utilizzato nella simulazione di sistemi dinamici che richiedono la conservazione dell'energia, come i sistemi orbitali. Una caratteristica distintiva di questo metodo è che non richiede l'uso esplicito delle velocità per calcolare le nuove posizioni, il che lo rende particolarmente utile per le simulazioni che necessitano di stabilità a lungo termine dell'energia.

Formule teoriche: Nel metodo di Verlet, la nuova posizione $\vec{r}_{\text{new}}$ di un corpo viene calcolata usando la posizione corrente $\vec{r}$, la posizione precedente $\vec{r}_{\text{old}}$, e l'accelerazione $\vec{a} = \frac{\vec{F}}{m}$ dove $\vec{F}$ è la forza netta che agisce sul corpo e m è la sua massa. L'equazione del metodo di Verlet è data da:

$$\vec{r}_{\text{new}} = 2\vec{r} - \vec{r}_{\text{old}} + \vec{a}\Delta t^2$$

Questa formula predice la posizione futura basandosi esclusivamente sulle posizioni corrente e passata e sull'accelerazione attuale, senza necessità di memorizzare le velocità.

Implementazione in Python:

```python
    def verlet(r, old_r, dt):
    # Calcola la forza gravitazionale al tempo
       corrente
    f = gravitational_force(M_sun, M_venus, r)
    # Calcola l'accelerazione
    new_acc = f / M_venus
    # Aggiorna la posizione usando Verlet
    new_r = 2 * r - old_r + new_acc * dt**2
    # La posizione precedente diventa la corrente
       per il prossimo passo
    return new_r, r
```

Come si osserva in Figura 2, il metodo di Verlet (verde) presenta la traiettoria più stabile, indicando una conservazione energetica superiore. Il metodo di Euler-Cromer (arancione) offre una stabilità intermedia, mentre il metodo di Euler (blu) risulta essere il meno stabile con una tendenza a divergere maggiormente.

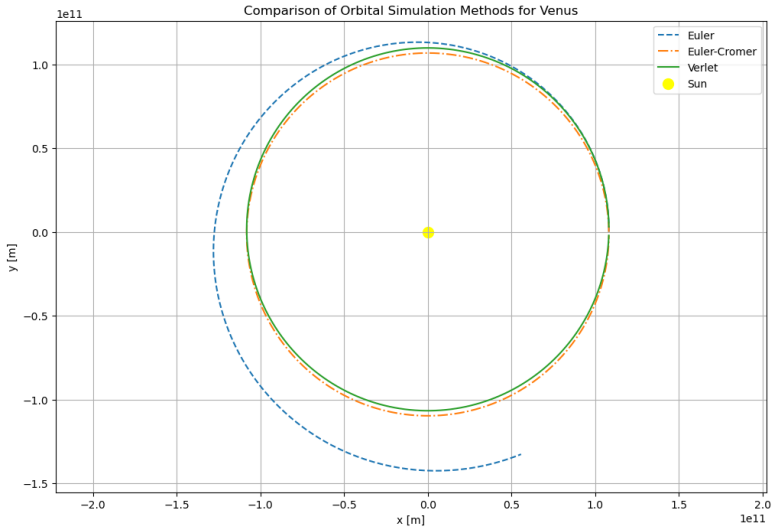

Figura 2: Confronto tra metodi di simulazione orbitale per Venere. Il metodo Verlet mostra la maggiore stabilità orbitale, seguito da Euler-Cromer, mentre il metodo di Euler evidenzia l'instabilità più pronunciata.

```
1
2  #Assignment1b.py
3  # %%
4  import numpy as np
5  import matplotlib.pyplot as plt
6
7  # Constants
8  G = 6.67408e-11  # Gravitational constant, N-m2/kg2
9  M_sun = 1.989e+30  # Sun's mass, kg
10 M_venus = 4.8675e+24  # Venus' mass, kg
11 initial_distance = 108.21e9  # Initial distance to
       Venus at perihelion, m
12 initial_velocity = 35.02e3  # Initial velocity of
       Venus, m/s
```

```
13
14 # Time span
15 one_venus_year = 225 * 24 * 3600  # in seconds
16
17 # Initial conditions
18 r_venus = np.array([initial_distance, 0])  # Initial
       position (perihelion)
19 v_venus = np.array([0, initial_velocity])  # Initial
       velocity
20
21 # Define gravitational force function
22 def gravitational_force(m1, m2, r):
23     return -G * m1 * m2 / np.linalg.norm(r)**3 * r
24
25 # Define integration methods
26 def euler(r, v, dt):
27     f = gravitational_force(M_sun, M_venus, r)
28     r_next = r + v * dt
29     v_next = v + f / M_venus * dt
30     return r_next, v_next
31
32 def euler_cromer(r, v, dt):
33     f = gravitational_force(M_sun, M_venus, r)
34     v_next = v + f / M_venus * dt
35     r_next = r + v_next * dt
36     return r_next, v_next
37
38 def verlet(r, old_r, dt):
39     f = gravitational_force(M_sun, M_venus, r)
40     new_acc = f / M_venus
41     new_r = 2 * r - old_r + new_acc * dt**2
42     return new_r, r
43
44 # Simulation setup
45 dt = 86400  # One day in seconds
46 n_steps = int(one_venus_year / dt)
47
48 # Initialize arrays
49 positions_euler = [r_venus.copy()]
50 positions_cromer = [r_venus.copy()]
```

```
positions_verlet = [r_venus.copy(), r_venus + v_venus * dt]  # Starting positions for Verlet

velocities_euler = [v_venus.copy()]
velocities_cromer = [v_venus.copy()]

# Simulation loop
for i in range(1, n_steps):
    new_r_euler, new_v_euler = euler(positions_euler[-1], velocities_euler[-1], dt)
    positions_euler.append(new_r_euler)
    velocities_euler.append(new_v_euler)

    new_r_cromer, new_v_cromer = euler_cromer(positions_cromer[-1], velocities_cromer[-1], dt)
    positions_cromer.append(new_r_cromer)
    velocities_cromer.append(new_v_cromer)

    if i < n_steps - 1:  # Verlet needs the previous two positions to compute the next one
        new_r_verlet, old_r_verlet = verlet(positions_verlet[-1], positions_verlet[-2], dt)
        positions_verlet.append(new_r_verlet)

# Convert lists to numpy arrays for plotting
positions_euler = np.array(positions_euler)
positions_cromer = np.array(positions_cromer)
positions_verlet = np.array(positions_verlet)

# Plot the results
plt.figure(figsize=(12, 8))
plt.plot(positions_euler[:, 0], positions_euler[:, 1], label='Euler', linestyle='--')
plt.plot(positions_cromer[:, 0], positions_cromer[:, 1], label='Euler-Cromer', linestyle='-.')
plt.plot(positions_verlet[1:, 0], positions_verlet[1:, 1], label='Verlet', linestyle='-')
plt.scatter([0], [0], color='yellow', s=100, label='
```

```
    Sun')  # Sun's position
plt.title('Comparison of Orbital Simulation Methods
    for Venus')
plt.xlabel('x [m]')
plt.ylabel('y [m]')
plt.legend()
plt.grid(True)
plt.axis('equal')
plt.show()
```

Listing 2: Codice Figura 2

Determinazione del Massimo Timestep

Per stimare il periodo di rotazione di Venere con un'accuratezza adeguata, abbiamo adottato un approccio iterativo che riduce il timestep finché la differenza tra il periodo simulato e quello reale è accettabile. Abbiamo iniziato con un timestep iniziale relativamente grande e lo abbiamo ridotto sistematicamente.

La tolleranza dell'accuratezza è stata impostata al 1% del periodo orbitale reale. Il criterio di arresto è stato raggiunto quando la differenza percentuale tra il periodo simulato e il periodo orbitale noto è diventata inferiore a tale soglia.

Algoritmo di Stima del Periodo

Utilizziamo la posizione di Venere ad ogni passo per determinare quando attraversa un asse. Questo punto di attraversamento è utilizzato come una stima del periodo orbitale. La precisione di questa stima migliora con un timestep minore.

```
import numpy as np
import matplotlib.pyplot as plt

# Costanti
G = 6.67408e-11  # Costante gravitazionale, N-m2/kg2
M_sun = 1.989e+30  # Massa del Sole, kg
```

```
M_venus = 4.8675e+24  # Massa di Venere, kg
initial_distance = 108.21e9  # Distanza iniziale da Venere al perielio, m
initial_velocity = 35.02e3  # Velocità iniziale di Venere, m/s

# Intervallo temporale
one_venus_year = 225 * 24 * 3600  # in secondi

# Condizioni iniziali
r_venus = np.array([initial_distance, 0])  # Posizione iniziale (perielio) di Venere
v_venus = np.array([0, initial_velocity])  # Velocità iniziale di Venere

# Definizione della funzione di forza gravitazionale
def gravitational_force(m1, m2, r):
    return -G * m1 * m2 / np.linalg.norm(r)**3 * r

# Definizione dei metodi di integrazione
def euler(r, v, dt):
    f = gravitational_force(M_sun, M_venus, r)
    r_next = r + v * dt
    v_next = v + f / M_venus * dt
    return r_next, v_next

def euler_cromer(r, v, dt):
    f = gravitational_force(M_sun, M_venus, r)
    v_next = v + f / M_venus * dt
    r_next = r + v_next * dt
    return r_next, v_next

def verlet(r, old_r, dt):
    f = gravitational_force(M_sun, M_venus, r)
    new_acc = f / M_venus
    new_r = 2 * r - old_r + new_acc * dt**2
    return new_r, r

# Funzione per calcolare l'energia cinetica
def kinetic_energy(v):
```

```
    return 0.5 * np.linalg.norm(v)**2

# Funzione per calcolare l'energia potenziale
def potential_energy(r):
    return -G * M_sun * M_venus / np.linalg.norm(r)

# Funzione per calcolare l'energia totale
def total_energy(r, v):
    return kinetic_energy(v) + potential_energy(r)

# Impostazione della simulazione
dt_values = [3600, 10800, 21600, 43200, 86400,
   172800, 259200, 432000, 604800, 2592000, 7776000,
    15552000, 31536000, 3153600000, 31536000000]  #
   secondi
dt_labels = ["1 ora", "3 ore", "6 ore", "12 ore", "1
    giorno", "2 giorni", "3 giorni", "5 giorni", "1
   mese", "3 mesi", "6 mesi", "1 anno", "100 anni (
   secolo)", "1000 anni (millennio)"]

for dt, dt_label in zip(dt_values, dt_labels):
    n_steps = int(one_venus_year / dt)

    # Inizializzazione degli array
    positions_euler = [r_venus.copy()]
    positions_cromer = [r_venus.copy()]
    positions_verlet = [r_venus.copy(), r_venus +
       v_venus * dt]  # Posizioni iniziali per il
       Verlet

    velocities_euler = [v_venus.copy()]
    velocities_cromer = [v_venus.copy()]

    # Ciclo di simulazione
    for i in range(1, n_steps):
        new_r_euler, new_v_euler = euler(
           positions_euler[-1], velocities_euler
           [-1], dt)
        positions_euler.append(new_r_euler)
        velocities_euler.append(new_v_euler)
```

```
73
74          new_r_cromer, new_v_cromer = euler_cromer(
                positions_cromer[-1], velocities_cromer
                [-1], dt)
75          positions_cromer.append(new_r_cromer)
76          velocities_cromer.append(new_v_cromer)
77
78          if i < n_steps - 1:  # Il Verlet ha bisogno
                delle due posizioni precedenti per
                calcolare la successiva
79              new_r_verlet, old_r_verlet = verlet(
                    positions_verlet[-1],
                    positions_verlet[-2], dt)
80              positions_verlet.append(new_r_verlet)
81
82      # Conversione delle liste in array numpy per il
            plotting
83      positions_euler = np.array(positions_euler)
84      positions_cromer = np.array(positions_cromer)
85      positions_verlet = np.array(positions_verlet)
86
87      # Calcolo delle energie iniziali e finali per
            ogni metodo
88      initial_energy_euler = total_energy(
            positions_euler[0], velocities_euler[0])
89      final_energy_euler = total_energy(
            positions_euler[-1], velocities_euler[-1])
90      energy_difference_euler = final_energy_euler -
            initial_energy_euler
91
92      initial_energy_cromer = total_energy(
            positions_cromer[0], velocities_cromer[0])
93      final_energy_cromer = total_energy(
            positions_cromer[-1], velocities_cromer[-1])
94      energy_difference_cromer = final_energy_cromer -
            initial_energy_cromer
95
96      initial_energy_verlet = total_energy(
            positions_verlet[0], positions_verlet[1] -
            positions_verlet[0])
```

```
    final_energy_verlet = total_energy(
        positions_verlet[-1], positions_verlet[-1] -
        positions_verlet[-2])
    energy_difference_verlet = final_energy_verlet -
        initial_energy_verlet

    # Stampare le differenze di energia per ogni
        metodo
    print("dt =", dt_label, "(corrispondente a", dt,
        "secondi)")
    print("Differenza di energia (Euler):",
        energy_difference_euler)
    print("Differenza di energia (Euler-Cromer):",
        energy_difference_cromer)
    print("Differenza di energia (Verlet):",
        energy_difference_verlet)
```

Listing 3: Codice Figura 3

Tabella 1: Differenza di energia

dt	Euler	E-Cromer	Verlet
1 ora	8.574×10^{31}	-3.723×10^{28}	3.703×10^{28}
3 ore	2.475×10^{32}	-8.680×10^{28}	8.668×10^{28}
6 ore	4.699×10^{32}	-9.833×10^{28}	9.829×10^{28}
12 ore	8.590×10^{32}	1.128×10^{29}	-1.129×10^{29}
1 giorno	1.482×10^{33}	1.533×10^{30}	-1.534×10^{30}
2 giorni	2.327×10^{33}	1.349×10^{31}	-1.353×10^{31}
3 giorni	2.868×10^{33}	2.300×10^{31}	-2.310×10^{31}
5 giorni	3.494×10^{33}	7.567×10^{31}	-7.665×10^{31}
1 mese	3.842×10^{33}	1.809×10^{32}	-1.860×10^{32}
3 mesi	4.730×10^{33}	2.608×10^{33}	2.515×10^{33}
6 mesi	3.766×10^{33}	4.959×10^{33}	3.766×10^{33}
1 anno	0.0	0.0	4.808×10^{33}
100 anni	0.0	0.0	5.389×10^{33}
1000 anni	0.0	0.0	5.965×10^{33}

I risultati sopra non ci hanno soddisfatto è interessante vedere quanto è facile impegnarsi e creare una simulazione poco efficiente ma non

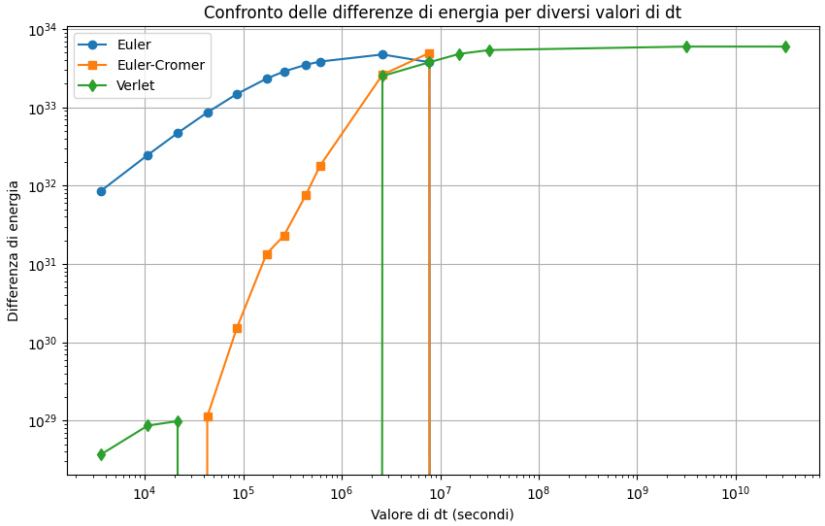

Figura 3: Confronto delle differenze di energia per diversi valori di dt

quanto riniziare da zero e dare un significato più adeguato. Quindi abbiamo cercato di ricreare una simulazione più accurata, la tabella generata dal codice che la segue rappresenta la nostra prima idea di considerare un punto di timestep dove la simulazione perde stabilità. Questo nuovo codice si presta a contare il numero delle volte in cui in un piano 2D la nostra massa passa nell'asse delle y=0, nella riga 91 del codice è possibile intuire che stiamo considerando il periodo, abbiamo messo /2 in modo tale da mettere in relazione gli anni venusiani con un periodo di rotazione orbitale completo. Per cui potremmo considerare un anno venusiano come un timestep.

Tabella 2: Periodo orbitale

Anni Venusiani	Metodo di Euler	Metodo di Euler-Cromer	Metodo di Verlet
5	2	5	5
8	3	8	8
10	4	10	10
100	17	100	100
500	40	500	500
1000	58	1001	1001
5000	127	5006	5006
10000	176	10013	10013

```
# %%
import numpy as np
import matplotlib.pyplot as plt

# Constants
G = 6.67408e-11  # Gravitational constant, N-m2/kg2
M_sun = 1.989e+30  # Sun's mass, kg
M_venus = 4.8675e+24  # Venus' mass, kg
initial_distance = 108.21e9  # Initial distance to
    Venus at perihelion, m
initial_velocity = 35.02e3  # Initial velocity of
    Venus, m/s

# Anni di interesse
years_of_interest = [5, 8, 10, 100, 500,
    1000,5000,10000]

# Time span
dt = 86400  # One day in seconds

# Define gravitational force function
def gravitational_force(m1, m2, r):
    return -G * m1 * m2 / np.linalg.norm(r)**3 * r

# Define integration methods
def euler(r, v, dt):
    f = gravitational_force(M_sun, M_venus, r)
```

```
25      r_next = r + v * dt
26      v_next = v + f / M_venus * dt
27      return r_next, v_next
28
29  def euler_cromer(r, v, dt):
30      f = gravitational_force(M_sun, M_venus, r)
31      v_next = v + f / M_venus * dt
32      r_next = r + v_next * dt
33      return r_next, v_next
34
35  def verlet(r, old_r, dt):
36      f = gravitational_force(M_sun, M_venus, r)
37      new_acc = f / M_venus
38      new_r = 2 * r - old_r + new_acc * dt**2
39      return new_r, r
40
41  # Simulation loop for each year of interest
42  for anni in years_of_interest:
43      # Time span for the current year
44      one_venus_year = anni * 225 * 24 * 3600  # in
            seconds
45      n_steps = int(one_venus_year / dt)
46
47      # Initial conditions
48      r_venus = np.array([initial_distance, 0])  #
            Initial position (perihelion)
49      v_venus = np.array([0, initial_velocity])  #
            Initial velocity
50
51      # Initialize arrays
52      positions_euler = [r_venus.copy()]
53      positions_cromer = [r_venus.copy()]
54      positions_verlet = [r_venus.copy(), r_venus +
            v_venus * dt]  # Starting positions for
            Verlet
55
56      velocities_euler = [v_venus.copy()]
57      velocities_cromer = [v_venus.copy()]
58
59      # Simulation loop
```

```
60      for i in range(1, n_steps):
61          new_r_euler, new_v_euler = euler(
                positions_euler[-1], velocities_euler
                [-1], dt)
62          positions_euler.append(new_r_euler)
63          velocities_euler.append(new_v_euler)
64
65          new_r_cromer, new_v_cromer = euler_cromer(
                positions_cromer[-1], velocities_cromer
                [-1], dt)
66          positions_cromer.append(new_r_cromer)
67          velocities_cromer.append(new_v_cromer)
68
69          if i < n_steps - 1:  # Verlet needs the
                previous two positions to compute the
                next one
70              new_r_verlet, old_r_verlet = verlet(
                    positions_verlet[-1],
                    positions_verlet[-2], dt)
71              positions_verlet.append(new_r_verlet)
72
73      # Convert lists to numpy arrays for counting
            periods
74      positions_euler = np.array(positions_euler)
75      positions_cromer = np.array(positions_cromer)
76      positions_verlet = np.array(positions_verlet)
77
78      # Calcolo dei periodi orbitali per ciascun
            metodo
79      def count_periods(positions, method):
80          crossings = 0  # Contatore di
                attraversamenti della coordinata y = 0
81          prev_x = positions[0, 0]  # Inizializzazione
                della coordinata x precedente
82          prev_y = positions[0, 1]  # Inizializzazione
                della coordinata y precedente
83          for pos in positions:
84              x, y = pos
85              if y * prev_y < 0:  # Verifica se siamo
                    passati attraverso y = 0
```

```
                crossings += 1
                deviation = x - prev_x  # Calcolo del discostamento in x
                #print("Deviazione in x al passaggio attraverso y = 0 (Metodo", method + "):", deviation, "metri")
            prev_x = x
            prev_y = y
        return crossings//2

    # Calcolo dei periodi orbitali per ciascun metodo
    periods_euler = count_periods(positions_euler, "Euler")
    periods_cromer = count_periods(positions_cromer, "Euler-Cromer")
    periods_verlet = count_periods(positions_verlet, "Verlet")

    print("Numero di periodi orbitali per il metodo di Euler (anni:", anni, "):", periods_euler)
    print("Numero di periodi orbitali per il metodo di Euler-Cromer (anni:", anni, "):", periods_cromer)
    print("Numero di periodi orbitali per il metodo di Verlet (anni:", anni, "):", periods_verlet)
```

Listing 4: Codice 4, Tabella 2

Dalla Tabella 2 possiamo notare segni di cedimento tra i 500 e i 1000 anni Venusiani. In conclusione visto che il codice precedente(figura 3) non calcolava accuratamente le energie abbiamo riprovato a farlo da zero ottenendo i seguenti risultati da cui possiamo vedere quanto eulercromer e verlet di per se siano stabili per questo caso.

```
import numpy as np
import matplotlib.pyplot as plt

# Constants
```

```
G = 6.67408e-11  # Gravitational constant, N-m2/kg2
M_sun = 1.989e+30  # Sun's mass, kg
M_venus = 4.8675e+24  # Venus' mass, kg
initial_distance = 108.21e9  # Initial distance to Venus at perihelion, m
initial_velocity = 35.02e3  # Initial velocity of Venus, m/s

# Time span
dt = 86400  # One day in seconds
years_of_interest = [5, 8, 10, 100, 500, 1000, 5000]  # Years of interest
methods = ['Euler', 'Euler-Cromer', 'Verlet']

# Define gravitational force function
def gravitational_force(m1, m2, r):
    return -G * m1 * m2 / np.linalg.norm(r)**3 * r

# Define integration methods
def euler(r, v, dt):
    f = gravitational_force(M_sun, M_venus, r)
    r_next = r + v * dt
    v_next = v + f / M_venus * dt
    return r_next, v_next

def euler_cromer(r, v, dt):
    f = gravitational_force(M_sun, M_venus, r)
    v_next = v + f / M_venus * dt
    r_next = r + v_next * dt
    return r_next, v_next

def verlet(r, old_r, dt):
    f = gravitational_force(M_sun, M_venus, r)
    new_acc = f / M_venus
    new_r = 2 * r - old_r + new_acc * dt**2
    return new_r, r

# Function to run simulation and return energy data
def simulate(anni, method):
    one_venus_year = anni * 225 * 24 * 3600  #
```

```
    Seconds in one Venus year
    n_steps = int(one_venus_year / dt)
    r_venus = np.array([initial_distance, 0])
    v_venus = np.array([0, initial_velocity])

    positions = [r_venus.copy()]
    if method != 'Verlet':
        velocities = [v_venus.copy()]
    else:
        positions.append(r_venus + v_venus * dt)  # Initial position for Verlet

    energy_kinetic = []
    energy_potential = []
    energy_total = []

    for i in range(1, n_steps):
        if method == 'Euler':
            new_r, new_v = euler(positions[-1], velocities[-1], dt)
            velocities.append(new_v)
        elif method == 'Euler-Cromer':
            new_r, new_v = euler_cromer(positions[-1], velocities[-1], dt)
            velocities.append(new_v)
        elif method == 'Verlet':
            if i < n_steps - 1:
                new_r, old_r = verlet(positions[-1], positions[-2], dt)
            else:
                new_r = positions[-1]
        positions.append(new_r)

        # Calculate velocities for Verlet
        if method == 'Verlet':
            v = (new_r - positions[-2]) / dt
        else:
            v = velocities[-1]

        # Calculate energies
```

```
77          kinetic_energy = 0.5 * M_venus * np.linalg.
               norm(v)**2
78          potential_energy = -G * M_sun * M_venus / np
               .linalg.norm(new_r)
79          total_energy = kinetic_energy +
               potential_energy
80          energy_kinetic.append(kinetic_energy)
81          energy_potential.append(potential_energy)
82          energy_total.append(total_energy)
83
84      return energy_kinetic, energy_potential,
           energy_total
85
86  # Create subplots
87  fig, axs = plt.subplots(len(years_of_interest), len(
       methods), figsize=(15, 20), constrained_layout=
       True)
88
89  for i, year in enumerate(years_of_interest):
90      for j, method in enumerate(methods):
91          kinetic, potential, total = simulate(year,
               method)
92          ax = axs[i, j]
93          ax.plot(kinetic, label='Kinetic Energy',
               linestyle='-.')
94          ax.plot(potential, label='Potential Energy',
                linestyle='--')
95          ax.plot(total, label='Total Energy',
               linestyle='-')
96          ax.set_title(f'{method} - Year: {year}')
97          ax.set_xlabel('Time Steps')
98          ax.set_ylabel('Energy (Joules)')
99          if i == 0 and j == 0:
100             ax.legend()
101
102 plt.suptitle('Energy Evolution for Venus Orbit
       Across Different Methods and Years')
103 plt.show()
```

Listing 5: Codice 5, Grafico Energie

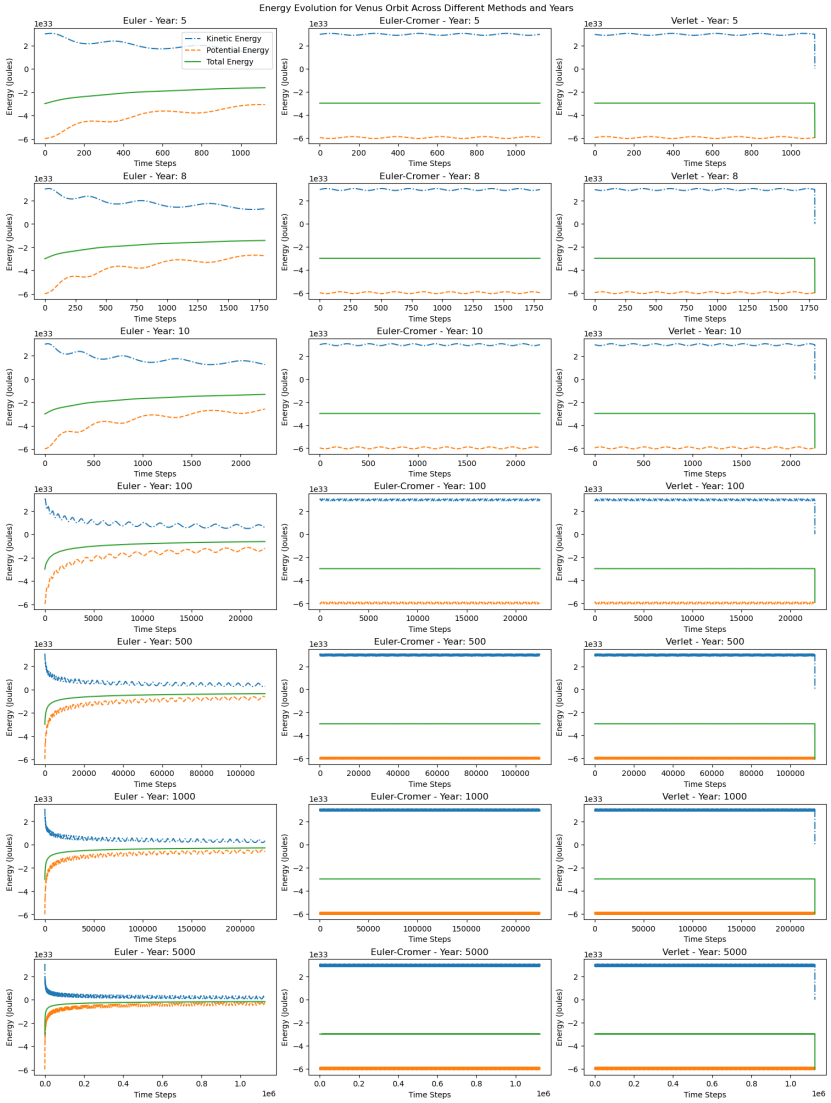

Figura 4: Grafici energie

-

0.3 1c

Verificate la veridicità della terza legge di Keplero per i pianeti Venere, Terra, e Marte.

Verifica della Terza Legge di Keplero

La Terza Legge di Keplero, nota anche come Legge dei Periodi, stabilisce che per tutti i pianeti del sistema solare, il quadrato del periodo orbitale T (il tempo impiegato per completare un'orbita) è proporzionale al cubo del semiasse maggiore a della loro orbita ellittica. In termini matematici, questa relazione è espressa dalla formula:

$$T^2 = k \cdot a^3 \tag{1}$$

dove k è una costante di proporzionalità che è la stessa per tutti i pianeti che orbitano attorno allo stesso corpo centrale, in questo caso, il Sole.

I dati per il periodo orbitale e il semiasse maggiore dei pianeti Venere, Terra e Marte saranno presi dalle osservazioni e dalle misurazioni fornite dalla *NASA*. Utilizzeremo questi dati per calcolare il valore di k per ciascun pianeta e verificare se si conformano alla legge di Keplero.

Dati Astronomici

I seguenti dati sono stati ottenuti dalle pagine di riferimento del NASA Planetary Fact Sheet:

- Venere: Periodo Orbitale $T_{Venere} = 0.61519726$ anni terrestri, Semiasse Maggiore $a_{Venere} = 0.723332$ AU.
- Terra: Periodo Orbitale $T_{Terra} = 1$ anno terrestre, Semiasse Maggiore $a_{Terra} = 1$ AU.
- Marte: Periodo Orbitale $T_{Marte} = 1.8808158$ anni terrestri, Semiasse Maggiore $a_{Marte} = 1.523662$ AU.

main.py Save Run Output

```python
# Definizione dei dati per Venere, Terra e Marte
T_Venere = 0.615  # in anni
a_Venere = 0.723  # in AU

T_Terra = 1.000  # in anni
a_Terra = 1.000  # in AU

T_Marte = 1.881  # in anni
a_Marte = 1.524  # in AU

# Calcolo di k per ciascun pianeta
k_Venere = T_Venere**2 / a_Venere**3
k_Terra = T_Terra**2 / a_Terra**3
k_Marte = T_Marte**2 / a_Marte**3

print("Valore di k per Venere:", k_Venere)
print("Valore di k per Terra:", k_Terra)
print("Valore di k per Marte:", k_Marte)
```

```
Valore di k per Venere: 1.0007724463019798
Valore di k per Terra: 1.0
Valore di k per Marte: 0.9995918121757503

=== Code Execution Successful ===
```

Figura 5: Rappresentazione delle orbite dei pianeti e della terza legge di Keplero.

Utilizzando questi dati, calcoleremo il valore di k per ciascun pianeta e li confronteremo per verificarne la coerenza con la terza legge di Keplero.

```python
import numpy as np
import matplotlib.pyplot as plt

# Dati per Venere, Terra e Marte
data = {
    'Venere': {'T': 0.61519726, 'a': 0.723332},
    'Terra': {'T': 1.0, 'a': 1.0},
    'Marte': {'T': 1.8808158, 'a': 1.523662}
}

# Calcolare k per ciascun pianeta e preparare i dati per il grafico
T_squared = []
a_cubed = []
labels = []

for planet, values in data.items():
    T = values['T']
    a = values['a']
    T_squared.append(T**2)
```

```
    a_cubed.append(a**3)
    labels.append(planet)

# Convertire le liste in array NumPy per facilitare i calcoli e il plotting
T_squared = np.array(T_squared)
a_cubed = np.array(a_cubed)

# Creare un grafico log-log
plt.figure(figsize=(8, 6))
plt.loglog(a_cubed, T_squared, 'bo', base=10)  # Punti dei pianeti

# Aggiungere annotazioni ai punti
for i, label in enumerate(labels):
    plt.annotate(label, (a_cubed[i], T_squared[i]))

# Aggiungere una linea diagonale per rappresentare la relazione lineare attesa
# La linea ha una pendenza di 1 in scala log-log
x = np.linspace(min(a_cubed)*0.8, max(a_cubed)*1.2, 400)
plt.plot(x, x, 'r--')  # Linea rossa tratteggiata

# Etichette e titolo
plt.xlabel('Semiasse Maggiore al Cubo (AU^3)')
plt.ylabel('Periodo Orbitale al Quadrato (anni^2)')
plt.title('Verifica della Terza Legge di Keplero in Scala Log-Log')
plt.grid(True)
plt.show()
```

Listing 6: Codice Grafico Keplero

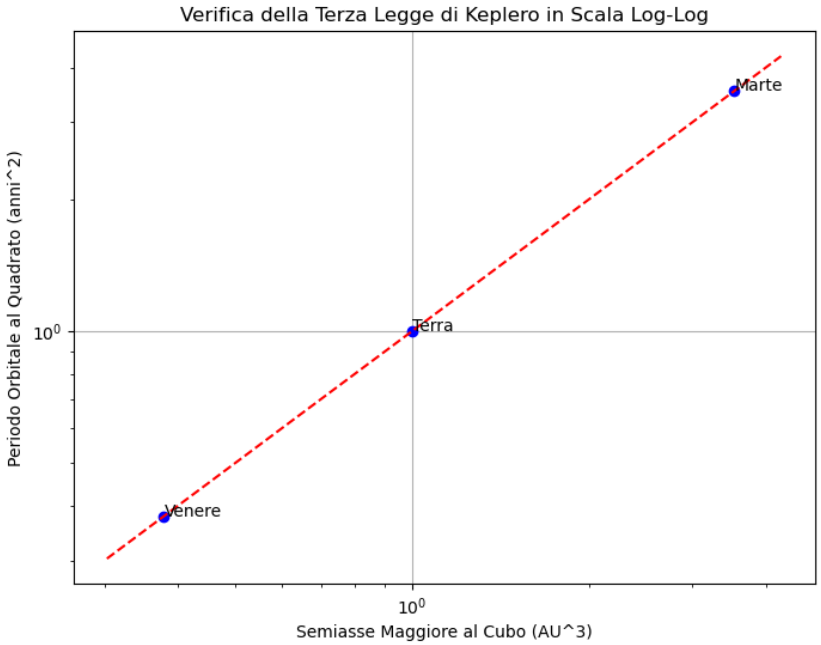

Figura 6: Grafico

Allegato *Questi script son presente anche in allegato come formato"assignment1abc.py" nella consegna. Son seguiti in sequenza.*

Assignment n.2– ODE: problema planetario a tre corpi

-

0.4 2a

Dato un sistema planetario a 3 corpi costituito da 3 massi planetarie uguali alla massa solare, trovare le condizioni inizali (posizione e velocità) che generano traiettorie stabili. Suggerimento: considerare le soluzioni proposte da Eulero e Lagrange.

Introduzione

In questo esercizio, affrontiamo il problema dei tre corpi, un classico problema della meccanica celeste in cui si studiano le traiettorie di tre corpi sottoposti alla reciproca interazione gravitazionale. L'obiettivo è trovare condizioni iniziali che generano traiettorie stabili per un sistema di tre corpi di massa uguale alla massa solare.

Soluzioni di Eulero e Lagrange

0.4.1 Soluzioni di Eulero

Leonhard Euler scoprì una famiglia di soluzioni al problema dei tre corpi, dove i tre corpi sono sempre in una configurazione colineare. Questo implica che i corpi si muovono lungo una linea retta, mantenendo sempre la stessa disposizione relativa.

Soluzioni di Lagrange

Joseph-Louis Lagrange trovò una famiglia di soluzioni più stabili, dove i corpi formano un triangolo equilatero che ruota mantenendo la forma del triangolo. Queste soluzioni sono utilizzate per analizzare i punti di Lagrange nel sistema Terra-Sole.

Formulazione Matematica

Le equazioni del moto per ciascun corpo, in assenza di altre forze oltre alla gravità, sono descritte dall'equazione differenziale:

$$\ddot{\vec{r}}_i = G \sum_{\substack{j=1 \\ j \neq i}}^{3} m_j \frac{\vec{r}_j - \vec{r}_i}{|\vec{r}_j - \vec{r}_i|^3} \tag{2}$$

dove $\vec{r}_i$ è la posizione del corpo i, m_j è la massa del corpo j, e G è la costante gravitazionale universale.

Simulazione Numerica

Utilizziamo Python per implementare e risolvere numericamente le equazioni del moto. La simulazione è basata su una discretizzazione temporale dell'intervallo da 0 a 100 unità di tempo, suddiviso in 10000 punti.

Definizione dei Parametri

Impostiamo i parametri della simulazione, inclusi i valori delle masse, le posizioni iniziali, e le velocità iniziali dei corpi.

```
G = 6.67408e-11  # Costante di gravitazione universale
m1 = m2 = m3 = 1.989e+30  # Masse dei corpi, uguali alla massa del sole
```

Equazioni del Moto

Le equazioni del moto sono implementate nella funzione `EquazioniTreCorpi`, che calcola le derivazioni temporali delle posizioni e delle velocità per i tre corpi:

```
def EquazioniTreCorpi(w, t, G, m1, m2, m3):
    # Implementazione delle equazioni del moto
    return derivs
```

Risoluzione delle Equazioni

Utilizziamo la funzione `odeint` per risolvere le equazioni del moto a partire dalle condizioni iniziali fino al tempo finale:

```
soluzione = scipy.integrate.odeint(EquazioniTreCorpi, init_params, time_span, args
```

Configurazioni Iniziali

Diverse configurazioni iniziali sono state sperimentate per esplorare la stabilità e la dinamica del sistema:

Configurazioni Simil-Eulero Queste configurazioni prevedono che i corpi siano inizialmente allineati e si muovano in modo da creare traiettorie complesse che possono portare due corpi verso l'infinito mentre il terzo compie un percorso isolato:

- Prima configurazione: Due corpi orbitano verso l'infinito, il terzo descrive un percorso che li annoda stabilmente.
- Seconda configurazione: Posizioni simmetriche che causano tre nodi e scambi, dopodiché due corpi vanno verso l'infinito e uno rimane isolato.

Configurazioni Triangolo Equilatero (Lagrange) In queste configurazioni, i corpi sono posizionati a formare un triangolo equilatero. Sono state esplorate diverse condizioni di velocità:

- Velocità nulle: I corpi si muovono direttamente verso il centro e collidono.
- Velocità iniziali verso il centro: Simile alla precedente, con i corpi che si incontrano al centro.
- Velocità ortogonali: Questa configurazione mantiene una buona stabilità fino a circa 3000 passi, dopodiché due corpi si allontanano verso l'infinito.

Simulazione Numerica e Animazione

Preparazione dell'Animazione

Per visualizzare le traiettorie dei tre corpi nel sistema, utilizziamo la libreria Matplotlib in Python, che offre un robusto supporto per la creazione di grafici 3D e animazioni. La preparazione dell'animazione comprende diverse fasi:

- **Configurazione della Figura e degli Assi:** Creiamo una figura e un asse 3D con dimensioni specificate per garantire che l'animazione sia chiaramente visibile.

```
fig = plt.figure(figsize=(12, 12))
ax = fig.add_subplot(111, projection='3d')
```

  Impostiamo inoltre l'angolo di vista in modo che l'animazione offra la migliore visuale possibile delle traiettorie.

```
ax.view_init(elev=8, azim=-60, roll=0)
```

- **Inizializzazione di Linee e Punti:** Inizializziamo le linee che tracciano le orbite e i punti che rappresentano le posizioni attuali dei corpi. Questo ci permette di aggiornare dinamicamente le posizioni durante l'animazione.

```
lines = [ax.plot([], [], [], '-', color="darkblue")[0], ...]
points = [ax.plot([], [], [], 'o', color="darkblue")[0], ...]
```

- **Impostazione dei Limiti degli Assi:** Definiamo i limiti degli assi per garantire che tutti i corpi rimangano visibili durante l'intera simulazione, indipendentemente dalla loro posizione nel sistema.

```
dim = 10
ax.set_xlim(-dim, dim)
ax.set_ylim(-dim, dim)
ax.set_zlim(-dim, dim)
```

- **Aggiunta di Testi e Etichette:** Aggiungiamo testi per migliorare la comprensione delle dinamiche osservate, includendo informazioni su posizioni e configurazioni iniziali.

```
init_text = ax.text2D(...)
pos_texts = [...]
```

Funzione di Aggiornamento

La funzione di aggiornamento `update` è chiamata ad ogni frame dell'animazione. Aggiorna le posizioni delle linee e dei punti in base ai dati della soluzione del problema dei tre corpi.

```
def update(num, r1_sol, r2_sol, r3_sol, lines, points, pos_texts):
    # Aggiornamento delle posizioni delle linee e dei punti
    return lines + points + [num_points_text]
```

Esecuzione dell'Animazione

Utilizziamo `FuncAnimation` per creare l'animazione, specificando il numero di frame e l'intervallo tra essi. L'intervallo è particolarmente importante per regolare la velocità dell'animazione e assicurare che il movimento dei corpi sia fluido e comprensibile.

```
ani = animation.FuncAnimation(...)
```

Questo approccio non solo permette di visualizzare in modo efficace le traiettorie e le interazioni dinamiche dei corpi celesti, ma fornisce anche un potente strumento didattico per comprendere meglio la complessità dei sistemi gravitazionali a più corpi, infatti abbiamo scelto di utilizzare l'osservazione dell'animazione come approccio per comprendere intuitivamente la stabilità delle orbite che differiscono di caso in caso a ogni minimo cambiamento!

Di seguito il codice completo settato con la configurazione a triangolo isoscele di Lagrange.

```
import numpy as np
import scipy.integrate
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import animation

# Definizione dei parametri per time_span
tempo_iniziale = 0
tempo_finale = 100
numero_punti_temporali = 10000

# Definizione di time_span utilizzando i parametri
time_span = np.linspace(tempo_iniziale, tempo_finale,
    numero_punti_temporali)

# Costanti
G = 6.67408e-11  # Costante di gravitazione universale
m_nd = 1.989e+30  # Massa del sole, kg
r_nd = 5.326e+12  # Distanza  esempio tra le stelle in
    Alpha Centauri, m
v_nd = 30000  # Velocità relativa della Terra intorno al
     sole, m/s
t_nd = 79.91 * 365 * 24 * 3600 * 0.51  # Periodo
    orbitale di Alpha Centauri
```

```
21 K1 = G * t_nd * m_nd / (r_nd**2 * v_nd)
22 K2 = v_nd * t_nd / r_nd
23
24 # Velocità dell'animazione
25 v_animazione = 1
26
27 # Masse
28 m1, m2, m3 = 1.989, 1.989, 1.989  # Masse senza u.m.
29 # Masse planetarie
30 m4 = m5 = m6 = 1.989e+30  # Massa solare
31 ################################  simil~Euler
32 # Segue condizioni iniziali di alpha centauri con una
      massa aggiunta, 2 masse orbitano verso l'infinito la
      terza fa il giro poi si annodano stabilmente
33 # r1, r2, r3 = np.array([-0.5, 0, 0]), np.array([0.5, 0,
       0]), np.array([0, 1, 0])
34 # v1, v2, v3 = np.array([0.01, 0.01, 0]), np.array
      ([-0.05, 0, -0.1]), np.array([0, -0.01, 0])
35 # Posizioni a logica simmetrica,fan 3 nodi e degli
      scambi poi 2 orbitano verso l'infinito e 1 isolato va
       verso l'infinito
36 # r1, r2, r3 = np.array([0.5, 0, 0]), np.array([-0.5, 0,
       0]), np.array([0, -1, 0])
37 # v1, v2, v3 = np.array([-0.01, 0.01, 0]), np.array
      ([0.05, 0, 0.1]), np.array([0, 0.01, 0])
38 # Posiziona i corpi su un piano e distribuisce le
      velocita' in modo che il terzo corpo mantenga una
      posizione piu' centrale rispetto agli altri due,
      curva 2 orb + 1 isolata come gli altri
39 # r1, r2, r3 = np.array([-0.5, 0, 0]), np.array([0.5, 0,
       0]), np.array([0, 1, 0])
40 # v1, v2, v3 = np.array([0.01, 0.01, 0]), np.array
      ([-0.02, 0, -0.04]), np.array([0, -0.01, 0])
41
42 ################################ Lagrange
43 # Posizioni a triangolo equilatero e velocità iniziali(
      nulle) , si scontrano perfettamente al centro tutti e
       3
44 # r1, v1 = np.array([1, 0, 0]), np.array([0, 0, 0])
45 # r2, v2 = np.array([-0.5, np.sqrt(3)/2, 0]), np.array
```

```
    ([0, 0, 0])
# r3, v3 = np.array([-0.5, -np.sqrt(3)/2, 0]), np.array
    ([0, 0, 0])
# Posizioni a triangolo equilatero e velocità iniziali(
    verso il centro), si scontrano perfettamente al
    centro tutti e 3
# r1, v1 = np.array([1, 0, 0]), np.array([-0.1, 0, 0])
# r2, v2 = np.array([-0.5, np.sqrt(3)/2, 0]), np.array
    ([0, -0.1, 0])
# r3, v3 = np.array([-0.5, -np.sqrt(3)/2, 0]), np.array
    ([0, 0.1, 0])
# Posizioni a triangolo isoscele e velocità iniziali(
    ortogonali) , buono fino a 3k passi poi vanno 2 in
    orbita verso l'infinito e uno isolato verso l'
    infinito
r1, v1 = np.array([1, 0, 0]), np.array([0, 0.1, 0])
r2, v2 = np.array([0, 1, 0]), np.array([0, 0, 0])
r3, v3 = np.array([0, -1, 0]), np.array([0, -0.1, 0])

def EquazioniTreCorpi(w, t, G, m1, m2, m3):
    r1, r2, r3 = w[:3], w[3:6], w[6:9]
    v1, v2, v3 = w[9:12], w[12:15], w[15:18]
    r12 = np.linalg.norm(r2 - r1)
    r13 = np.linalg.norm(r3 - r1)
    r23 = np.linalg.norm(r3 - r2)

    dv1bydt = K1 * m2 * (r2 - r1) / r12**3 + K1 * m3 * (
        r3 - r1) / r13**3
    dv2bydt = K1 * m1 * (r1 - r2) / r12**3 + K1 * m3 * (
        r3 - r2) / r23**3
    dv3bydt = K1 * m1 * (r1 - r3) / r13**3 + K1 * m2 * (
        r2 - r3) / r23**3
    dr1bydt = K2 * v1
    dr2bydt = K2 * v2
    dr3bydt = K2 * v3
    derivs = np.concatenate([dr1bydt, dr2bydt, dr3bydt,
        dv1bydt, dv2bydt, dv3bydt])
    return derivs

# Risoluzione delle equazioni
```

```
73 init_params = np.concatenate([r1, r2, r3, v1, v2, v3])
74 soluzione = scipy.integrate.odeint(EquazioniTreCorpi,
       init_params, time_span, args=(G, m1, m2, m3))
75
76 # Estrazione delle soluzioni
77 r1_sol = soluzione[:, :3]
78 r2_sol = soluzione[:, 3:6]
79 r3_sol = soluzione[:, 6:9]
80
81 # Preparazione dell'animazione
82 fig = plt.figure(figsize=(12, 12))  # Aumentato il
       valore della dimensione della figura
83 ax = fig.add_subplot(111, projection='3d')
84 # Impostazione angolo di vista
85 ax.view_init(elev=8, azim=-60, roll=0)
86
87 # Inizializzazione linee e punti per l'animazione
88 lines = [ax.plot([], [], [], '-', color="darkblue")[0],
89          ax.plot([], [], [], '-', color="tab:red")[0],
90          ax.plot([], [], [], '-', color="magenta")[0]]
               # Cambiato il colore giallo in magenta
91 points = [ax.plot([], [], [], 'o', color="darkblue")[0],
92           ax.plot([], [], [], 'o', color="tab:red")[0],
93           ax.plot([], [], [], 'o', color="magenta")[0]]
94
95 # Limiti e label del plot
96 dim = 10
97 ax.set_xlim(-dim, dim)  # Ingranditi gli assi di dim
       volte
98 ax.set_ylim(-dim, dim)
99 ax.set_zlim(-dim, dim)
100 ax.set_xlabel("X")
101 ax.set_ylabel("Y")
102 ax.set_zlabel("Z")
103 ax.set_title("Simulazione Problema dei Tre Corpi")
104
105 # Visualizzazione delle condizioni iniziali e
       aggiornamento delle posizioni correnti
106 init_text = ax.text2D(0.01, 0.99, '', transform=ax.
       transAxes, verticalalignment='top', fontsize=6)
```

```
107 pos_texts = [
108     ax.text2D(0.99, 0.92, '', transform=ax.transAxes,
        verticalalignment='top', horizontalalignment='
        right', color="darkblue", fontsize=6),
109     ax.text2D(0.99, 0.89, '', transform=ax.transAxes,
        verticalalignment='top', horizontalalignment='
        right', color="tab:red", fontsize=6),
110     ax.text2D(0.99, 0.86, '', transform=ax.transAxes,
        verticalalignment='top', horizontalalignment='
        right', color="magenta", fontsize=6)
111 ]
112
113 # Testo per il numero di punti temporali
114 num_points_text = ax.text2D(0.01, 0.86, f'Punti
    temporali: {numero_punti_temporali}', transform=ax.
    transAxes, verticalalignment='top', fontsize=6)
115
116 # Uniamo il testo dell'intervallo con il testo delle
    posizioni
117 interval_text = ax.text2D(0.99, 0.82, f'[Velocità
    animazione]: {v_animazione} ms', transform=ax.
    transAxes, verticalalignment='top',
    horizontalalignment='right', fontsize=3.8)
118
119 # Aggiornamento delle posizioni
120 def update(num, r1_sol, r2_sol, r3_sol, lines, points,
    pos_texts):
121     for line, point, r_sol, pos_text in zip(lines,
        points, [r1_sol, r2_sol, r3_sol], pos_texts):
122         line.set_data(r_sol[:num+1, 0:2].T)
123         line.set_3d_properties(r_sol[:num+1, 2])
124         point.set_data(r_sol[num, 0:2].T)
125         point.set_3d_properties(r_sol[num, 2])
126         pos_text.set_text(f'Pos: {np.round(r_sol[num],
            2).tolist()}')
127     num_points_text.set_text(f'Punti temporali: {num+1}'
        )  # Aggiorna il numero di punti temporali
        durante l'animazione
128     return lines + points + [num_points_text]
129
```

```
130 # Creazione dell'animazione
131 ani = animation.FuncAnimation(fig, update, frames=len(
        time_span), fargs=(r1_sol, r2_sol, r3_sol, lines,
        points, pos_texts),
132                               interval=v_animazione,
                                      blit=False)
133
134 plt.show()
```

Mantenendo la stessa configurazione del codice sopra otteniamo i seguenti risultati entro i 3000 step:

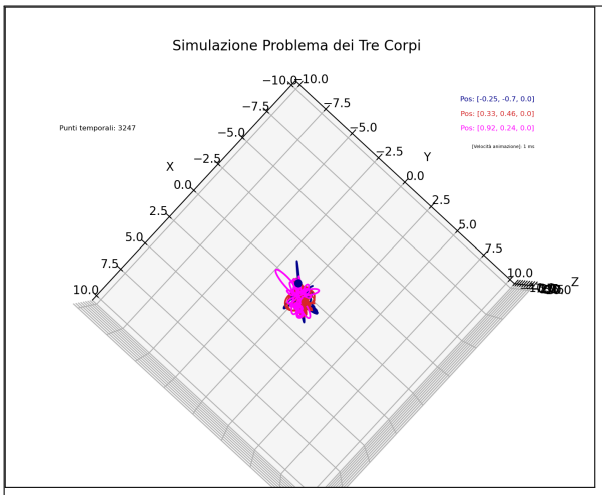

Figura 7: Velocità ortogonali e distanze a triangolo isoscele

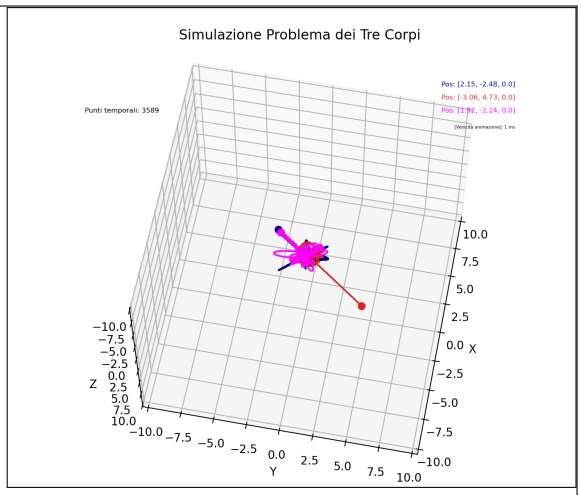

Figura 8: Vel. ortogonali e dist. a triangolo isoscele dopo 3k punti

Figura 9: Confronto delle velocità ortogonali con distanze a triangolo isoscele, prima e dopo 3000 punti temporali.

La maggior parte delle simulazioni perde stabilità quando un terzo corpo entra in 2 orbite stabili e funge da fionda gravitazionale creando comportamenti distinti, condivido diverse immagini ottenuti da configurazioni generali del codice sopra. ***Allegato*** *Questo script è presente anche in allegato come "assignment2a.py" nella consegna, richiede diversi minuti per arrivare a 10k step, attenzione, potrebbe salire la tentazione di voler velocizzare la simulazione abbassando a minor numero di step ma questo rovina la smulazione aumentando le instabilità*

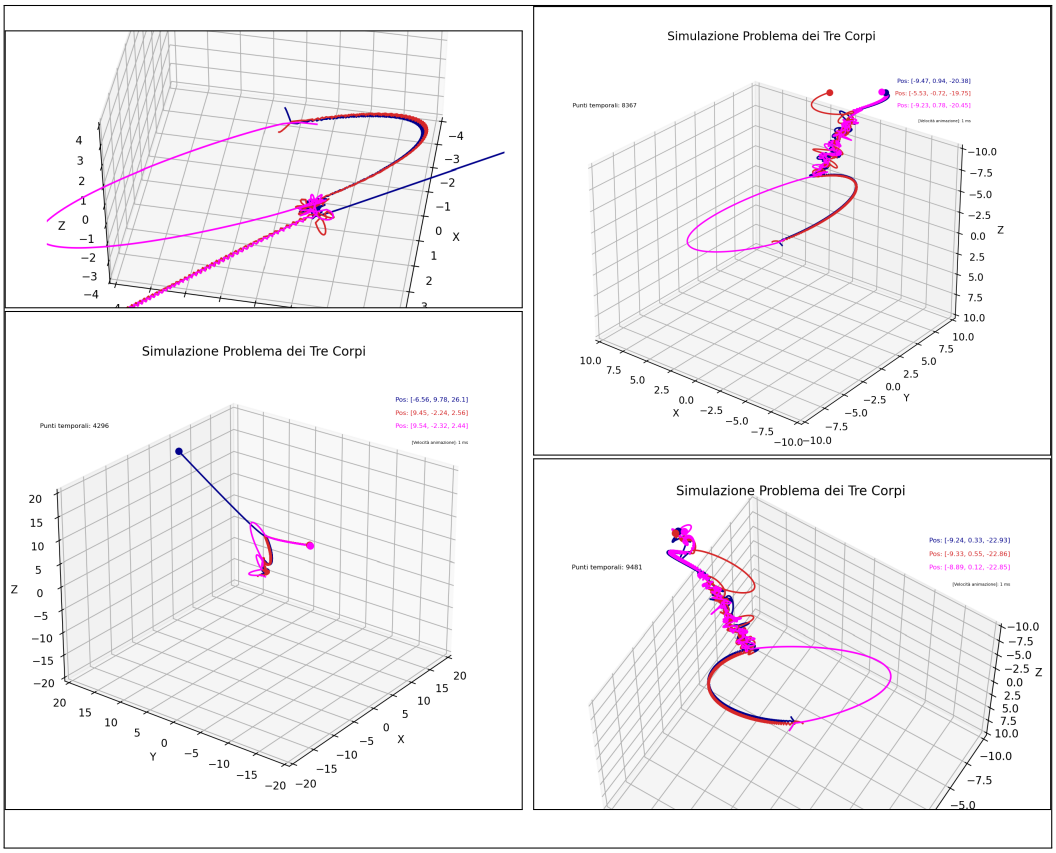

Figura 10: Queste condizioni interessanti son ottenute dalle prime configurazioni definite simil-Eulero

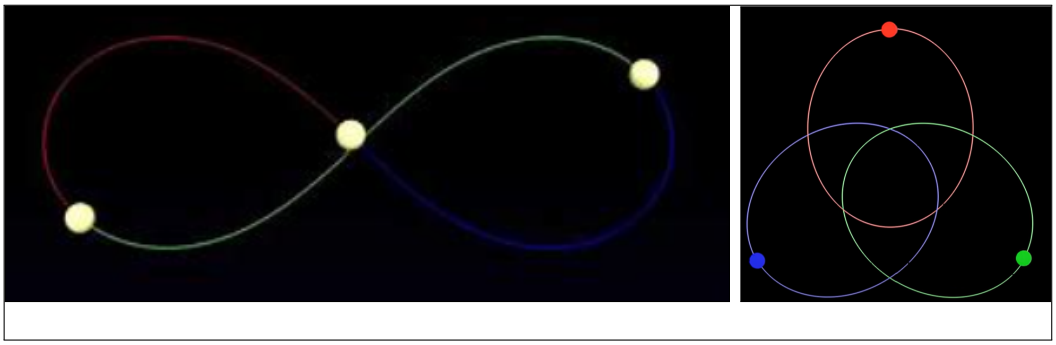

Figura 11: Condizioni iniziali per sistemi ideali perfetti, non reali, idee interessanti che fanno riflettere

Assignment n.3– PDE: equazione del calore

-

0.5 3a

Studiare l'evoluzione temporale (utilizzando la soluzione numerica FTCS dell'equazione parabolica del calore) di un profilo di temperatura sinusoidale in un sistema unidimensionale di lunghezza $L = 100$ m. T$(x;0)$=$T_0 - A_0 \cos(2x/\lambda)$ con:

- $T_0 = 500$ K: temperatura media
- $A_0 = 20$ K: ampiezza iniziale del profilo
- $\lambda = L/4$: lunghezza d'onda del profilo di temperatura

Rappresentare graficamente l'evoluzione temporale del profilo imponendo le condizioni al contorno di Neumann.

Introduzione

Per implementare la simulazione numerica dell'equazione del calore con un profilo iniziale sinusoidale in Python usando il metodo FTCS (Forward Time, Centered Space), seguiremo i passaggi descritti di seguito. Imposteremo le condizioni al contorno di Neumann, che implicano che la derivata della temperatura rispetto allo spazio ai bordi è zero. Questo significa che non c'è flusso di calore attraverso i confini del sistema.

Teoria e Formulazione Matematica

L'equazione del calore in una dimensione è espressa come:

$$\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial x^2}$$

dove α è la diffusività termica del materiale. Utilizzando il metodo FTCS, discretizziamo questa equazione nel tempo e nello spazio.

Data una griglia spaziale con passo h e un intervallo temporale τ, l'approssimazione numerica diventa:

$$T_i^{n+1} = T_i^n + \frac{\alpha\tau}{h^2}(T_{i-1}^n - 2T_i^n + T_{i+1}^n)$$

Questo schema richiede che la condizione di stabilità $\frac{\alpha\tau}{h^2} \leq \frac{1}{2}$ sia soddisfatta per evitare instabilità numerica.

Setup Iniziale e Discretizzazione

- Lunghezza del dominio: $L = 100$ m.
- Temperatura media: $T_0 = 500$ K.
- Ampiezza iniziale del profilo: $A_0 = 20$ K.
- Lunghezza d'onda del profilo di temperatura: $\lambda = \frac{L}{4}$.
- Discretizzazione spaziale h e temporale τ.
- Numero di punti nel dominio spaziale N calcolato in base a h e L.

Condizione Iniziale

Il profilo iniziale di temperatura è dato da:

$$T(x, 0) = T_0 - A_0 \cos\left(\frac{2\pi x}{\lambda}\right)$$

Metodo FTCS e Visualizzazione

Il metodo FTCS è applicato per aggiornare la temperatura nel tempo, mantenendo le condizioni al contorno di Neumann per assicurare che non ci sia flusso di calore attraverso i confini del sistema. L'evoluzione temporale del profilo di temperatura viene poi graficata per visualizzare come la temperatura si stabilizza verso un equilibrio. Quindi applicare il metodo FTCS per aggiornare la temperatura nel tempo e gestire le condizioni al contorno di Neumann aggiornando i valori di temperatura agli estremi del dominio in modo da mantenere la derivata spaziale nulla.

Visualizzazione

Graficare l'evoluzione temporale del profilo di temperatura.

Codice

```python
import numpy as np
import matplotlib.pyplot as plt

# Costanti e condizioni iniziali
L = 100.0  # lunghezza del dominio in metri
T0 = 500.0  # temperatura media in Kelvin
A0 = 20.0  # ampiezza del profilo di temperatura in Kelvin
lambda_wave = L / 4  # lunghezza d'onda del profilo di temperatura

N = 100  # numero di punti spaziali
h = L / (N - 1)  # dimensione del passo spaziale
tau = 0.1  # passo temporale in secondi
kappa = 0.1  # diffusività termica in m^2/s

# Verifica della condizione di stabilità per il metodo FTCS
assert tau <= h**2 / (2 * kappa), "Condizione di stabilità non soddisfatta!"

# Numero di passi temporali da simulare
time_steps = 2000

# Profilo di temperatura iniziale
x = np.linspace(0, L, N)
T = T0 - A0 * np.cos(2 * np.pi * x / lambda_wave)

# Preparazione alla simulazione
T_new = np.copy(T)
results = [np.copy(T)]

# Simulazione con schema FTCS
```

```
for _ in range(time_steps):
    for i in range(1, N-1):
        T_new[i] = T[i] + kappa * tau / h**2 * (T[i+1] - 2*T[i] + T[i-1])
    # Condizioni al contorno di Neumann
    T_new[0] = T_new[1]
    T_new[-1] = T_new[-2]
    T = np.copy(T_new)
    results.append(np.copy(T))

# Creazione del grafico dei risultati
plt.figure(figsize=(10, 6))
for i, temp_profile in enumerate(results[::200]):
    plt.plot(x, temp_profile, label=f'Tempo = {i*200*tau:.1f} s')
plt.title('Evoluzione Temporale del Profilo di Temperatura')
plt.xlabel('Posizione (m)')
plt.ylabel('Temperatura (K)')
plt.legend()
plt.show()
```

Risultati

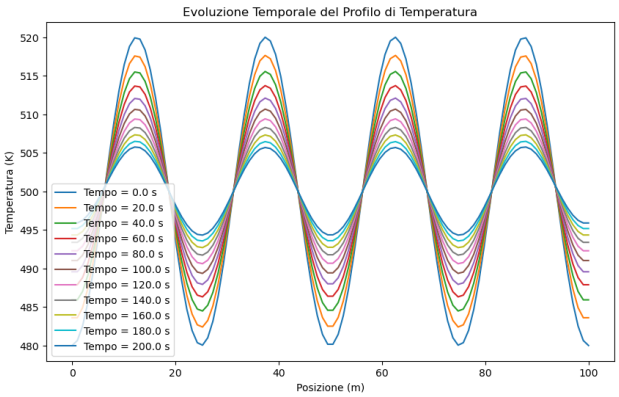

Figura 12: Evoluzione temporale del profilo di temperatura.

Discussione

Il grafico evidenzia come le ampiezze delle oscillazioni di temperatura diminuiscano nel tempo, indicando la tendenza del sistema a raggiungere uno stato di equilibrio termico. Inizialmente, i massimi e i minimi locali del profilo mostrano una derivata seconda spaziale (d'') positiva nei massimi e negativa nei minimi. La derivata temporale (dT/dt), risulta essere negativa nei massimi, dove la temperatura tende a diminuire, e positiva nei minimi, dove la temperatura tende ad aumentare.

Col passare del tempo, come mostrato nel grafico, l'ampiezza delle variazioni di temperatura si riduce significativamente. Questo comportamento riflette il processo di omogeneizzazione: le regioni più calde cedono calore alle regioni più fredde, riducendo le disomogeneità iniziali e spingendo il sistema verso uno stato uniforme di temperatura media.

Conclusione

Le osservazioni confermano la natura diffusiva del processo termico governato dall'equazione parabolica del calore, dimostrando come la diffusione agisce per "appiattire" il profilo di temperatura, riducendo le disomogeneità e portando il sistema verso l'equilibrio termico.

-

0.6 3b

Dimostrare che l'ampiezza del profilo sinusoidale decresce esponenzialmente nel tempo secondo la legge: $A(t) = A_0 \exp(-t/\tau) + c$.

Teoria e Derivazione

Consideriamo l'equazione del calore in una dimensione, dove la temperatura $T(x, t)$ segue la legge di diffusione:

$$\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial x^2}$$

dove α è la diffusività termica del materiale.

Per un profilo di temperatura iniziale sinusoidale, $T(x,0) = T_0 - A_0 \cos\left(\frac{2\pi x}{\lambda}\right)$, la soluzione dell'equazione del calore può essere espressa come una somma di modi che si attenuano esponenzialmente nel tempo. In particolare, il primo modo, che corrisponde alla frequenza fondamentale della condizione iniziale, evolve come segue:

$$T(x,t) = T_0 - A(t) \cos\left(\frac{2\pi x}{\lambda}\right)$$

dove

$$A(t) = A_0 \exp\left(-\frac{4\pi^2 \alpha t}{\lambda^2}\right)$$

Definendo $\tau = \frac{\lambda^2}{4\pi^2\alpha}$, riscriviamo la legge di attenuazione come:

$$A(t) = A_0 \exp\left(-\frac{t}{\tau}\right)$$

Questo mostra che l'ampiezza $A(t)$ decresce esponenzialmente nel tempo con una costante di tempo τ, che è inversamente proporzionale alla diffusività termica e al quadrato della lunghezza d'onda.

Simulazione Numerica

Per verificare la validità di questa analisi, si può implementare una simulazione numerica utilizzando il metodo FTCS e tracciare il decadimento dell'ampiezza nel tempo. I risultati della simulazione dovrebbero confermare la dipendenza esponenziale di $A(t)$ dal tempo come descritto dalla formula teorica.

Codice Python

```
import numpy as np
import matplotlib.pyplot as plt

# Parametri del modello
L = 100.0  # lunghezza del dominio (m)
```

```
N = 500    # numero di punti nel dominio spaziale
x = np.linspace(0, L, N)
dx = L / (N - 1)
kappa = 0.1  # diffusività termica (m^2/s)
dt = 0.01  # intervallo di tempo (s)
lambda_wave = L / 4  # lunghezza d'onda del profilo di temperatura
T0 = 500   # temperatura media (K)
A0 = 20    # ampiezza iniziale (K)

# Condizione iniziale
T = T0 - A0 * np.cos(2 * np.pi * x / lambda_wave)

# Tempo di simulazione e array per l'analisi dell'ampiezza
time_steps = 1000
amplitudes = []

# Simulazione numerica usando il metodo FTCS
for t in range(time_steps):
    T_new = np.copy(T)
    for i in range(1, N-1):
        T_new[i] = T[i] + kappa * dt / dx^2 * (T[i-1] - 2*T[i] + T[i+1])

    # Condizioni al contorno di Neumann
    T_new[0] = T_new[1]
    T_new[-1] = T_new[-2]

    T = np.copy(T_new)

    # Calcolo dell'ampiezza attuale
    if t % 50 == 0:  # Estrarre l'ampiezza ogni 50 passi temporali
        current_amplitude = (np.max(T) - np.min(T)) / 2
        amplitudes.append(current_amplitude)

# Calcolo della teoria dell'ampiezza decadimento esponenziale
```

```
times = np.arange(0, time_steps, 50) * dt
tau = lambda_wave^2 / (4 * pi^2 * kappa)
theoretical_amplitudes = A0 * exp(-times / tau)

# Grafico dei risultati
plt.figure(figsize=(10, 6))
plt.plot(times, amplitudes, 'bo-', label='Ampiezza Simulata')
plt.plot(times, theoretical_amplitudes, 'r--', label='Ampiezza Teorica $A_0 e^{-t/\\tau}$')
plt.title('Decadimento dell\'ampiezza del profilo di temperatura')
plt.xlabel('Tempo (s)')
plt.ylabel('Ampiezza (K)')
plt.legend()
plt.grid(True)
plt.show()
```

Risultati e Discussione

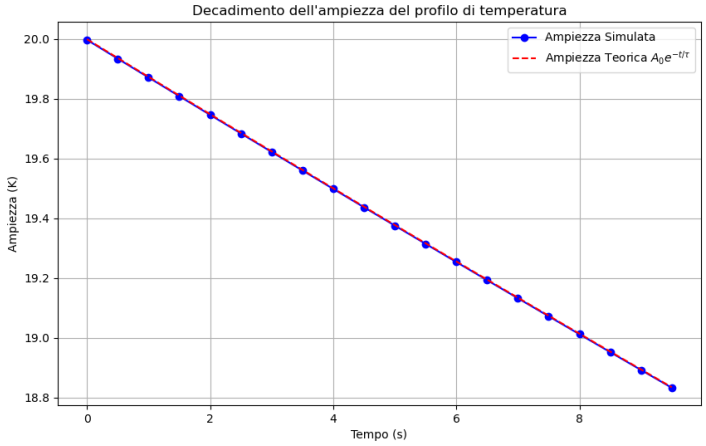

Figura 13: Confronto tra l'ampiezza simulata e quella teorica.

La simulazione numerica conferma la teoria prevista. Come mostrato nel grafico, l'ampiezza simulata segue da vicino il decadimento espo-

nenziale previsto dalla formula teorica $A(t) = A_0 \exp(-t/\tau)$. Questo risultato ci permette di notare quindi l'efficacia del modello matematico e del metodo numerico utilizzato per descrivere il raffreddamento in un sistema termico soggetto alla diffusione.

-

0.7 3c

Eseguire 3 simulazioni con $\alpha = 0.5, 5$ e 50 m^2/sec e λ fissato. Eseguire 3 simulazioni con $\lambda = L, L/2$ ed $L/4$ e α fissato. Dimostrare per tutti i casi precedenti che il tempo di rilassamento τ è legato alla diffusività termica α dalla seguente relazione approssimata: $\alpha \approx (\lambda^2/4\pi^2\tau)$.

Teoria e Formulazione Matematica

La diffusività termica α e la lunghezza d'onda λ sono parametri chiave nell'equazione del calore:

$$\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial x^2}$$

L'ampiezza del profilo di temperatura sinusoidale si attenua nel tempo seguendo una legge esponenziale, dove il tempo di rilassamento τ, per una data λ, è definito come:

$$\tau = \frac{\lambda^2}{4\pi^2 \alpha}$$

Simulazioni Numeriche

Utilizziamo un approccio di discretizzazione spaziale e temporale per risolvere numericamente l'equazione del calore e osservare il decadimento dell'ampiezza termica. Il codice è implementato in Python e utilizza il metodo di differenze finite esplicite per approssimare le derivate.

Codice

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
```

```
def heat_equation_simulation(L, alpha, lambda_wave, dx, A0=20, T0=500):
    N = int(L / dx) + 1
    x = np.linspace(0, L, N)
    T = T0 - A0 * np.cos(2 * np.pi * x / lambda_wave)
    kappa = alpha

    dt = (dx**2) / (2 * kappa)  # Condizione di stabilità
    time_steps = 10000  # Numero elevato di passi temporali
    amplitudes = []

    for _ in range(time_steps):
        T_new = np.copy(T)
        for i in range(1, N-1):
            T_new[i] = T[i] + kappa * dt / dx**2 * (T[i-1] - 2*T[i] + T[i+1])
        T = np.copy(T_new)
        current_amplitude = (np.max(T) - np.min(T)) / 2
        amplitudes.append(current_amplitude)

    return amplitudes, dt

def compute_tau(amplitudes, dt, reduction_factor=1):
    initial_amplitude = amplitudes[0]
    target_amplitude = initial_amplitude / np.exp(reduction_factor)
    for i, amplitude in enumerate(amplitudes):
        if amplitude < target_amplitude:
            return i * dt
    return None

def plot_amplitude_decay(fig, ax, amplitudes, dt, alpha, lambda_wave, index, reduction_factor=1):
    times = np.linspace(0, len(amplitudes) * dt, len(amplitudes))
    ax.plot(times, amplitudes, label=f'Alpha = {alpha}, Lambda = {lambda_wave} m')
```

```
    ax.axhline(y=amplitudes[0] / np.exp(reduction_factor), color='r', linestyle='--', label=f'Target: $e^{{-{reduction_factor}}}$ Amplitude')
    ax.set_title(f'Decay for Alpha = {alpha}, Lambda = {lambda_wave}')
    ax.set_xlabel('Time (s)')
    ax.set_ylabel('Amplitude (K)')
    ax.legend()
    ax.grid(True)

# Parametri
L = 100.0
dx = 0.1
reduction_factor = 0.5

# Setup della figura e degli assi
fig, axs = plt.subplots(2, 3, figsize=(18, 12))
axs = axs.flatten()

# Simulazioni con alpha variabile e lambda fissato
lambda_wave = L / 4
alphas = [0.5, 5, 50]
for i, alpha in enumerate(alphas):
    amplitudes, dt = heat_equation_simulation(L, alpha, lambda_wave, dx)
    tau = compute_tau(amplitudes, dt, reduction_factor)
    print(f'Alpha: {alpha}, Tau: {tau} seconds')
    plot_amplitude_decay(fig, axs[i], amplitudes, dt, alpha, lambda_wave, i, reduction_factor)

# Simulazioni con lambda variabile e alpha fissato
alpha_fixed = 5
lambdas = [L, L / 2, L / 4]
for i, lambda_wave in enumerate(lambdas):
    amplitudes, dt = heat_equation_simulation(L, alpha_fixed, lambda_wave, dx)
    tau = compute_tau(amplitudes, dt, reduction_factor)
    print(f'Lambda: {lambda_wave}, Tau: {tau} seconds')
    plot_amplitude_decay(fig, axs[i + 3], amplitudes, dt, alpha_fixed, lambda_wave, i + 3,
```

```
        reduction_factor)

fig.tight_layout()
plt.show()
```

Risultati e Discussione

Dai risultati delle simulazioni, osserviamo come il tempo di rilassamento τ vari con α e λ, confermando la relazione teorica. Di seguito, le figure illustrano i risultati ottenuti per le diverse configurazioni testate.

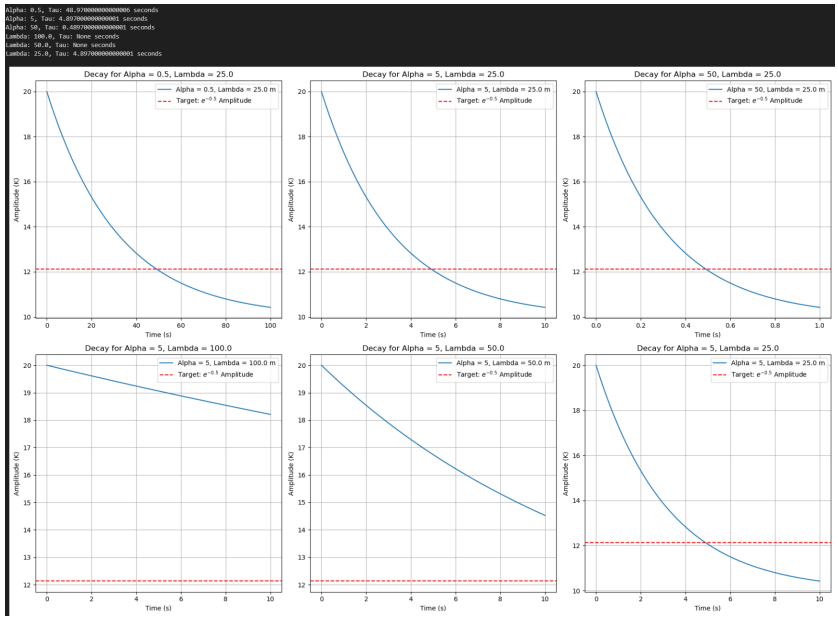

Figura 14: Decadimento dell'ampiezza per diversi valori di α e λ.

Risultati del tempo di rilassamento:

- Alpha: 0.5, Tau: 48.97 seconds
- Alpha: 5, Tau: 4.897 seconds
- Alpha: 50, Tau: 0.490 seconds
- Lambda: 100.0 m, Tau: None

- Lambda: 50.0 m, Tau: None
- Lambda: 25.0 m, Tau: 4.897 seconds

Conclusione

I risultati confermano la relazione teorica tra α, λ, e τ.

Allegato *Questo script è presente anche in allegato come "3c.ipynb"*