Sant'Anna
Scuola Universitaria Superiore Pisa

Complementi di Matematica: EDO e loro applicazioni

Studio analitico di un sistema di oscillatori attraverso le EDO e sistemi numerici per risoluzione numerica

---

Studenti:
Cairone Giuseppe
Rossi Gianmarco
Scuola Superiore di Studi Universitari
e di Perfezionamento Sant'Anna

20/13/2023

Indice

1 Introduzione

Per questo progetto si intende studiare un sistema composto da n pendoli, ciascuno con massa m e braccio di lunghezza l (non massivo), fissati ad un supporto di massa M.

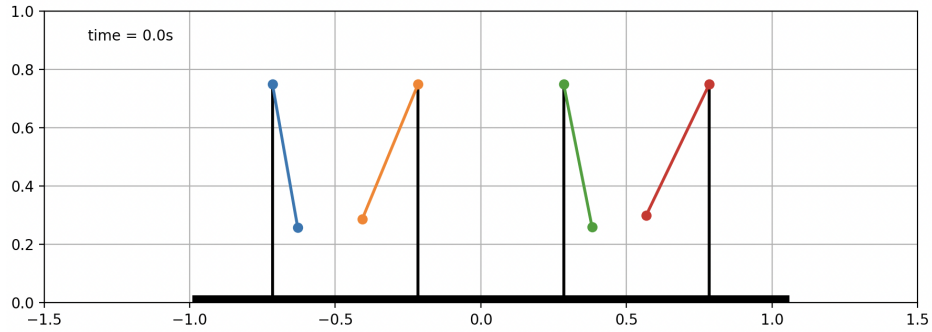

Figura 1: Condizione iniziale per 4 pendoli

Il sistema ha quindi $n+1$ gradi di libertà, uno legato al moto orizzontale del supporto e uno per ogni pendolo. Il primo è determinato dalla posizione $x(t)$ del supporto, i gradi di libertà associati ai pendoli sono determinati dall'angolo del pendolo rispetto alla verticale $\theta_i(t)$. Per ciò che si è interessati a studiare poniamo $x(0) = 0$, $\dot{x}(0) = 0$. Dunque lo stato iniziale è determinato da $\theta_i(0), \dot{\theta}_i(0)$ per ogni i-esimo pendolo con $1 \leq i \leq n$.

2 Oscillazioni libere

Per studiare il sistema si fa uso delle equazioni di Eulero-Lagrange, le quali ci permettono, a partire dall'energia cinetica e quella potenziale, di ricavare le equazioni del moto per il sistema.

2.1 Energia del Sistema

Si parte quindi andando a scrivere le equazioni per ricavare l'energia cinetica del sistema che risulta essere

$$K = \frac{1}{2}M\dot{x}^2 + \frac{1}{2}\sum_{i=0}^{n} m v_i^2$$

dove le varie v_i sono le velocità delle masse nel sistema di riferimento del laboratorio quindi:

$$v_i^2 = \left(l\dot{\theta}_i \sin\theta_i\right)^2 + \left(\dot{x} + l\dot{\theta}_i \cos\theta_i\right)^2$$

L'energia potenziale la calcoliamo ponendo lo zero del potenziale nel vertice di oscillazione dei pendoli in modo da semplificare l'espressione della stessa e quindi anche i calcoli. In definitiva si ottiene che:

$$U = -\sum_{i=0}^{n} mgl\cos\theta_i$$

Da notare che non viene considerata l'energia potenziale del supporto perché rimane costante nel tempo. Interessando a noi la differenza di energia potenziale tutti i termini costanti possono quindi essere omessi.

2.2 Lagrangiana del Sistema

Una volta scritte le equazioni delle energie possiamo procedere a scirvere la lagrangiana del sistema ovvero:

$$L\left(x, \dot{x}, \theta_i, \dot{\theta}_i\right) = K - U = \frac{1}{2}M\dot{x}^2 + \frac{1}{2}\sum_{i=0}^{n} m v_i^2 + \sum_{i=0}^{n} mgl\cos\theta_i$$

Per ottenere il sistema di equazioni differenziali del moto applico l'equazione di Eulero-Lagrange ad ogni coordinata

$$\frac{\partial L}{\partial q_j}(\mathbf{q}, \dot{\mathbf{q}}, t) - \frac{d}{dt}\frac{\partial L}{\partial \dot{q}_j}(\mathbf{q}, \dot{\mathbf{q}}, t) = 0$$

Si ottiene quindi il sistema

$$\begin{cases} \dfrac{\partial L}{\partial \theta_1} = & \dfrac{d}{dt}\dfrac{\partial L}{\partial \dot{\theta_1}} \\ \dots & \\ \dfrac{\partial L}{\partial \theta_n} = & \dfrac{d}{dt}\dfrac{\partial L}{\partial \dot{\theta_n}} \\ \dfrac{\partial L}{\partial x} = & \dfrac{d}{dt}\dfrac{\partial L}{\partial \dot{x}} \end{cases} \Rightarrow \begin{cases} \ddot{\theta}_1 = \dfrac{gsin(\theta_1) - \ddot{x}cos(\theta_1)}{l} \\ \dots \\ \ddot{\theta}_n = \dfrac{gsin(\theta_n) - \ddot{x}cos(\theta_n)}{l} \\ \ddot{x} = \displaystyle\sum_{i=1}^{n} \dfrac{ml\dot{\theta_i^2}\sin(\theta_i) - mg\sin(2\theta_i)}{M + m\sin^2(\theta_i)} \end{cases}$$

Andando a risolvere per le coordinate $x(t)$ e $\theta_i(t)$ si ottiene l'evoluzione del sistema

3 Risoluzione numerica

Una volta analizzato analiticamente il sistema dinamico si procede a risolvere le equazioni. Dato che non esiste una funzione esplicita che risolva le equazioni differenziali ottenute. Si usa quindi un sistema numerico per ottenere un risultato.

3.1 Metodo numerico

Per risolvere numericamente le equazioni differenziali abbiamo scelto di usare il metodo numerico di Runge-Kutta di ordine 4. Questo metodo garantisce la simpletticità del sistema ovvero che l'energia totale del sistema rimanga quanto più costante e limitata nel tempo. Proprio quest'ultimo fattore è importante in quanto il sistema iniziale prevede una conservazione dell'energia totale, aspetto che deve rispecchiarsi anche nel metodo numerico.

3.2 Implementazione

3.2.1 Premesse

Per semplicità abbiamo scelto di implementare il metodo numerico in Python: linguaggio che offre ottime prestazioni ma al contempo permette di facilitare l'analisi dati. Per rendere il processo di

integrazione ancora più prestante abbiamo utilizzato anche la libreria *Cython* che permette di una tipizzazione delle variabili come in C. Con queste accortezze siamo riusciti a dimezzare il tempo di calcolo per il nostro algoritmo. Abbiamo poi deciso di usare *MatPlotLib* per graficare i nostri risultati in modo da avere un riscontro visivo di ciò che l'integratore è in grado di fare.

3.2.2 Algoritmo di Integrazione

L'algoritmo di integrazione, come già accennato, si basa sul metodi di Runge-Kutta si ordine 4. Abbiamo quindi bisogno di una funzione che data la condizione iniziale del sistema y_n ci restituisca la sua derivata rispetto al tempo. Questo l'abbiamo fatto usando le equazioni per le accelerazioni trovate con la Lagrangiana del sistema, che trascritte e poste in una funzione sono:

```
def dSdt(cnp.ndarray S):
    # S   di tipo [ang1, ome1, ..., angk, omek, pos, vel]
    # res di tipo [ome1, acc1, ..., omek, acck, vel, acc]
    # inizializzo il vettore dei risultati a 0
    cdef cnp.ndarray res = np.zeros(2 * self.N + 2, dtype=DTYPE)

    # calcolo l'accelerazione del supporto
    cdef double acc = self.m *  sum(self.l * (S[k + 1] ** 2 * sin(S[k])) - g * sin(2 * S[k])
                                    for k in range(0, 2 * self.N, 2))
    acc /= self.M + self.m * sum(sin(S[k]) ** 2
                                 for k in range(0, 2 * self.N, 2))

    # calcolo omega e acc ang
    cdef double acc_ang
    for k in range(0, 2 * self.N, 2):
        # sposto le velocità dal vettore iniziale a quello dei risultati
        res[k] = S[k + 1]
        # calcolo l'acc ang in funzione della velocità e posizione
        acc_ang = (g * sin(S[k]) - acc * cos(S[k])) / self.l
        # controllo se è attivo il damping e effettuo le modifiche necessarie all'accelerazione
        if damping: acc_ang -= mi * S[k + 1] * ((S[k] / teta0) ** 2 - 1)
        # aggiungo l'acc ang ai risultati
        res[k + 1] = acc_ang

    # sposto la velocità del supporto dal vettore iniziale a quello dei risultati
    res[2 * self.N] = S[2 * self.N + 1]
    # aggiungo l'accelerazione al vettore dei risultati
    res[2 * self.N + 1] = acc
    return res
```

Data la funzione che regola le accelerazioni e le velocità del sistema si passa alla parte di integrazione vera e propria con la prima funzione che viene chiamata con argomenti la funzione che regola il sistema, le condizioni iniziali del sistema e il *timestep* che si vuole utilizzare.

```
def integrate_all(fun: callable, y0: cnp.ndarray, double h) -> cnp.ndarray:
    # calcola per ogni istante di tempo l'integrazine in base alle condizioni di partenza

    # iniziallizza il vettore dei risultati a 0
    cdef cnp.ndarray res = np.zeros(shape=(self.n + 1, 2 * self.N + 2), dtype=DTYPE)
    # il primo istante è quello delle condizioni iniziali
    res[0] = y0

    # assegna al tempo da valutare il tempo iniziale
    t_eval = self.t

    # intera lungo il tempo
    for t, index in zip(t_eval, range(1, len(t_eval))):
```

```
        # per ogni istante di tempo aggiunge al risultato l'initegrazione corrispondente che riporta quindi
        ↪  posizione e velocità
        res[index] = integrate_t(fun, res[index - 1], h)
        # print(res[index], end="\n")

    return res
```

La funzione `integrate_all` fa una chiamata a `integrate_t` per ogni timestep nel tempo totale e per ogni nuovo step chiama la funzione `integrate_t` con lo stato del sistema nell'istante precedente. In questo modo iteriamo per tutto il tempo dato e troviamo la soluzione numerica al moto del sistema.

```
def integrate_t(fun: callable, cnp.ndarray yn , float h):
    # calcola uno step di integrazione date le condizioni iniziali, lo step h e la funzione che regola il moto
    cdef cnp.ndarray k1 = h*fun(yn)
    cdef cnp.ndarray k2 = h*fun(yn + (1/2)*k1)
    cdef cnp.ndarray k3 = h*fun(yn + (1/2)*k2)
    cdef cnp.ndarray k4 = h*fun(yn + k3)
    # calcola la nuova posizione e la restituisce per l'iterazione successiva
    return yn + (1/6)*k1 + (1/3)*k2 + (1/3)*k3 + (1/6)*k4
```