12 Terza Lezione A

12.1 Programmazione a oggetti: problema N-body

Python è un linguaggio che permette la programmazione ad oggetti. Molto di Python stesso è scritto con programmazione orientata ad oggetti. Lo scopo è quello di illustrare un semplice esempio di utilizzo creando una piccola simulazione ad N corpi. Per programmare ad ogetti si utilizza quelle che sono chiamate classi; fondamentalmente creare una classe vuol dire definire un oggetto. All'interno della classe è possibile definire delle funzioni che verranno chiamati metodi. Alcuni metodi sono particolari e sono contrassegnati da doppi underscore, e.g: "__init__, __iter__, __next__, __call__"; qui ci limiteremo al primo, in quanto necessario, e l'ultimo poiché può essere utile. I due nel mezzo permettono di costruire un "iterabile", parola che non dovrebbe esservi nuova; così come l'ultimo permette di costruire un oggetto "chiamabile" ("callable" anche questo non dovrebbe sembrarvi nuovo). Cominciamo quindi, nell'ottica della nostra simulazione a N corpi, a definire la nostra classe che ci permetterà di costruire i protagonisti della nostra simulazione (ci limiteremo per semplicità a vincolare la dinamica su un piano):

```
import numpy as np
import random as rn
import matplotlib.pyplot as plt
from matplotlib import animation

class Body:
    """
    Classe che rappresenta una pallina
    intesa come ogetto puntiforme
    """

    def __init__(self, x, y, vx, vy, m=1):
        """
        costruttore della classe, verra' chiamato
        quando creeremo l'istanza della classe, (i.e. b=Body()
        b e' chiamata istanza).
        In input prende la posizione, la velocita' e massa
        che sono le quantita' che identificano il corpo
        che saranno gli attributi della classe;
        il costruttore e' un particolare metodi della
        classe per questo si utilizzano gli underscore.
        il primo parametro che passiamo (self) rappresenta
        l'istanza della classe (self e' un nome di default)
        questo perche' la classe e' un modello generico che
        deve valere per ogni corpo.
        """
        #posizione
        self.x = x
        self.y = y
        #velocita'
        self.vx = vx
        self.vy = vy
        #massa
        self.m = m

    #aggiornamento posizione e velocita' con eulero
    def n_vel(self, fx, fy, dt):
        """
        ad ogni metodo della classe viene passato
        come primo argomento self , quindi l ' istanza
        date le componenti della forza e il passo temporale
        aggiorno le componenti della velocita'
        """
        self.vx += fx*dt
        self.vy += fy*dt

    def n_pos(self, dt):
        """
        dato il passo temporale aggiorno le posizioni
        """
        self.x += self.vx*dt
        self.y += self.vy*dt
```

Non è propriamente necessario usare i metodi per cambiare gli attributi di una classe, si possono utilizzare cose quali le property e i setter, ma non ce ne cureremo. Dunque ora abbiamo qualcosa che ci permette di creare i nostri oggetti, ognuno con caratteristiche (valori degli attributi) diversi. Abbiamo poi due metodi che ci permetto di aggiornare le quantità fisiche e dare una dinamica a tutti i nostri corpi. Vediamo ora, sempre

grazie ad un altra classe come implementare la nostra simulazione. Useremo per evitare divergenze in caso di incontri ravvicinati (del terzo tipo) quella che è la tecnica del softening, cioè il potenziale kepleriano sarà:

$$V(r) = -\frac{1}{\sqrt{r^2 + \epsilon}} \tag{48}$$

con ϵ chiamato appunto parametro di softening. Così anche se i pianeti sono molto vicini La forza tra essi non diverge. Se vogliamo simulare una normale orbita possiamo sempre settare a zero questo parametro.

```
class Sistema:
    '''
    Classe per evoluzione del sistema.
    Viene utilizzata la tecnica del softening  per impedire
    divergenze nella foza, sp e' il parametro di softening

    Parameters
    ----------
    corpi : list
        lista di ogetti della classe Body
    G : float
        Costante di gravitazione universale (=1)
    sp : float, optional, default 0
        parametro di softening
    '''

    def __init__(self, corpi, G, sp=0):
        self.corpi = corpi
        self.G = G
        self.sp = sp

    def evolvo(self, dt):
        '''
        chimata ad ogni passo temporale, fa evolvere il sistema
        solo di uno step dt, la forza e' calcolata secondo la
        legge di gravitazione universale;
        '''

        for corpo_1 in self.corpi:

            fx = 0.0
            fy = 0.0

            for corpo_2 in self.corpi:
                if corpo_1 != corpo_2:

                    dx = corpo_2.x - corpo_1.x
                    dy = corpo_2.y - corpo_1.y

                    d = np.sqrt(dx**2 + dy**2 + self.sp)

                    fx += self.G * corpo_2.m * dx / d**3
                    fy += self.G * corpo_2.m * dy / d**3

            corpo_1.n_vel(fx, fy, dt)

        for corpo in self.corpi:
            corpo.n_pos(dt)
```

Fondamentalmente La nostra classe ci permette quindi di integrare le varie equazioni del moto. Importante notare il metodo con cui esse sono integrate: aggiorniamo prima tutte le velocità e poi tutte le posizioni. Questo è chiamato metodo di eulero simplettico. Ora mostriamo la parte conclusiva del codice per avviare la nostra simulazione:

```
#=========================================================================
# Creating bodies and the system and computational parameters
#=========================================================================

rn.seed(69420)
dt = 1/20000
T  = int(2/dt)
E  = np.zeros(T)
L  = np.zeros(T)
G  = 1

# Number of body, must be even
N = 10
```

```
C = []
for n in range(N//2):
    '''
    two bodies are created at a time
    with equal and opposite velocity
    to keep the total momentum of the system zero
    '''
    v_x = rn.uniform(-0.5, 0.5)
    v_y = rn.uniform(-0.5, 0.5)
    C.append(Body(rn.uniform(-0.5, 0.5), rn.uniform(-0.5, 0.5), v_x, v_y))
    C.append(Body(rn.uniform(-0.5, 0.5), rn.uniform(-0.5, 0.5), -v_x, -v_y))


X = np.zeros((2, T, N)) # 2 because the motion is on a plane

# Creation of the system
soft = 0.01
sist = System( C, G, soft)

#================================================================================
# Evolution
#================================================================================

start = time.time()

for t in range(T):
    sist.update(dt)
    for n, body in enumerate(sist.bodies):
        X[:, t, n] = body.x, body.y

print("--- %s seconds ---" % (time.time() - start))

#================================================================================
# Plot and animation
#================================================================================

fig = plt.figure(0)
plt.grid()
plt.xlim(np.min(X[::2, :])-0.5, np.max(X[::2, :])+0.5)
plt.ylim(np.min(X[1::2,:])-0.5, np.max(X[1::2,:])+0.5)
colors = ['b']*N#plt.cm.jet(np.linspace(0, 1, N))

dot  = np.array([]) # for the planet

for c in colors:
    dot  = np.append(dot,  plt.plot([], [], 'o', c=c))

def animate(i):

    for k in range(N):

        dot[k].set_data(X[0, i, k], X[1, i, k])

    return dot

anim = animation.FuncAnimation(fig, animate, frames=np.arange(0, T, 50), interval=1, blit=True
    , repeat=True)


plt.title('N body problem', fontsize=20)
plt.xlabel('X(t)', fontsize=20)
plt.ylabel('Y(t)', fontsize=20)

# Ucomment to save the animation, extra_args for .mp4
#anim.save('N_body.gif', fps=50)# extra_args=['-vcodec', 'libx264'])

plt.show()
```

Oltre ad n palline a caso, creiamo anche un sistema, un pianeta che orbita intorno a due stelle:

```
# creation of body
C1 = Body(0.5,  0, 0, 20,  int(1e3))
C2 = Body(-0.5, 0, 0, -20, int(1e3))
C3 = Body(-1.5, 0, 0, 40,  int(1e1))
C  = [C1, C2, C3]
N  = len(C)
```

Ora in linea di principio la simulazione finisce qui. Tuttavia risulta interessante calcolare l'energia e il momento angolare del nostro sistema e vedere se esse, come ci si aspetteremmo, sono effettivamente conservate durante l'evoluzione del sistema. Mostreremo i risultati ma senza mostrare il codice che comunque trovate disponibile. Tra l'altro questa analisi la vogliamo fare mostrando soltanto il caso di tre corpi che si orbitano attorno in quanto nei due casi che mostreremo (due integratori diversi) le differenze sono apprezzabili. Usando N corpi creati a caso la conservazione dell'energia non si manifesta mai, a differenza, e vedremo perché, della conservazione del momento angolare. Per quanto riguarda la conservazione dell'energia probabilmente ciò è dovuto al fatto che, quelle scelte, non sono delle buone condizioni iniziali: in genere l'inizializzazione è affrontata in maniera più delicata (a noi interessava solo vedere l'animazione carina). Da qui e dal tipo di integratore usato, che abbiamo citato prima nasce tutta un'interessante discussione sulla meccanica classica, che brevemente vogliamo trattare.

12.2 Breve compendio di meccanica Hamiltoniana

Allora come prima cosa diamo un paio di definizioni, cercando comunque di non essere troppo matematici (non dimostriamo nulla qui, lasciamo tutto all'eventuale curiosità dell'eventuale lettore):
Consideriamo inizialmente una 2-forma ω differenziale su una varietà liscia M. La 2-forma ω è chiamata simplettica se è chiusa e non degenere. Chiusa significa che la sua derivata esterna è nulla $\mathrm{d}\omega = 0$; non vogliamo perdere tempo a spiegare cosa sia la derivata esterna, ma la potete vedere come una derivata che sia antisimmetica (cfr. tensore elettromagnetico: dato A potenziale vettore $\mathrm{d}A = \partial_i A_j - \partial_j A_i = F_{ij}$). Non degenere significa invece che, per ogni punto $p \in M$, considerando lo spazio tangente al punto T_pM (che è uno spazio vettoriale), si ha che per ogni vettore x non nullo su questo spazio esiste un vettore y tale che: $\omega(x, y) = 0$ solo per $y = 0$. Dunque la coppia (M, ω) è una varietà simplettica.
Consideriamo ora una funzione liscia $H : M \to \mathbb{R}$ su una varietà simplettica (M, ω). Abbiamo allora che il campo vettoriale X_H definito da $\omega(X_H, Y) = \mathrm{d}H(Y)$ è chiamato campo vettoriale hamiltoniano generato da H. La terna (M, ω, H) definisce un sistema hamiltoniano.
Inoltre è importante menzionare il seguente teorema: Sia (M, ω, H) un sistema hamiltoniano e sia $\Phi : \mathbb{R} \times M \to M$ il flusso generato dal campo vettoriale X_H. Allora $\forall t \in \mathbb{R}$ Φ_t è simplettico. Questo significa che il flusso non solo preserva la struttura simplettica, in particolare esso preserva la forma volume, il che vuol dire che preserva il volume dello spazio delle fasi. Inoltre il flusso esatto della nostra hamiltoniana è reversibile $\Phi_t^{-1} = \Phi_{-t}$
Detto ciò vediamo di concretizzare un po'; supponiamo di avere una hamiltoniana della forma:

$$H(p, q) = T(p) + V(q) \quad . \tag{49}$$

Se definiamo $x = (p, q)$ l'insieme di tutte le nostre variabili allora sappiamo che l'equazione del moto sono date dalla parentesi di poisson:

$$\dot{x} = X_H = \{x, H(x)\} \quad , \tag{50}$$

la cui soluzione è:

$$x(\tau) = e^{\tau X_H} x(0) \quad . \tag{51}$$

Abbiamo detto che vogliamo dunque integrare le nostre equazioni del moto. Assumiamo che dato un certo n esistano certi coefficienti $c_1, \ldots, c_k$ e $d_1, \ldots, d_k$ tale che:

$$e^{\tau X_H} = e^{\tau(X_T + X_V)} = \prod_{i=1}^{k} e^{\tau c_i X_T} e^{\tau d_i X_V} + \mathcal{O}(\tau^{n+1}) \quad , \tag{52}$$

dove i c_i e d_i devono rispettare certi vincoli a seconda del valore di n, e a seconda del quale inoltre si ottengono diversi algoritmi: per $n = 1$ abbiamo eluero simplettico (quello implementato sopra) mentre per $n = 2$ abbiamo il leapfrog. Non ci metremo ora a calcolare questi coefficienti. Inoltre si dimostra, ma noi non lo facciamo, che trovare i c_i e d_i che soddisfano la (52) è equivalente a trovare dei coefficienti c_i e d_i tali che:

$$\Phi_\tau = \prod_{i=1}^{k} e^{\tau c_i X_T} e^{\tau d_i X_V} = e^{\tau(X_T + X_V) + \mathcal{O}(\tau^{n+1})} \quad . \tag{53}$$

É inoltre interessante notare che se consideriamo l'integratore di eulero: $\Phi_h = e^{hX_T} e^{hX_V}$ esso è soluzione esatta di un altro sistema hamiltoniano scritto come perturbazioni di H, che non è detto abbia le stessi simmetrie dell'hamiltoniana di partenza:

$$\overline{H} = H + hH_2 + h^2 H_3 + \ldots \tag{54}$$

dove gli H_i si calcolano tramite parentesi di poisson espandendo in serie con la formula di Baker-Campbell-Hausdorff (BCH):

$$\begin{aligned}\overline{X_H} &= \frac{1}{h}\ln(e^{hX_T}e^{hX_V}) \\ &= \underbrace{X_T + X_V}_{H} + h\underbrace{\frac{1}{2}[X_T, X_V]}_{H_2} + h^2\underbrace{\frac{1}{12}([[X_T, X_V], X_V] + [[X_V, X_T], X_T])}_{H_3} \quad .\end{aligned} \tag{55}$$

Più in generale per un integratore della forma: $\Phi_h = \prod_{i=1}^{k} e^{hc_i X_T} e^{hd_i X_V}$ esso è soluzione di:

$$\overline{H} = H + h^n H_{n+1} + \mathcal{O}(h^{n+1}) \quad , \tag{56}$$

dove ora i vari H_i saranno diversi ma il modo di calcolarli è sempre lo stesso. Tornando a noi è stato implementato per fare un confronto anche un medoto simplettico del 4 ordine noto con il nome di Yoshida; scrivendo in coordinate lagrangiane la regola iterativa è:

$$v_{i+1} = v_i + d_i a(x_i)dt \tag{57}$$
$$x_{i+1} = x_i + c_i v_{i+1}dt \tag{58}$$

dove i coefficienti valgono:

$$c_1 = c_4 = \frac{1}{2(2-2^{1/3})} \qquad c_2 = c_3 = \frac{1-2^{1/3}}{2(2-2^{1/3})}, \tag{59}$$
$$d_1 = d_3 = \frac{1}{2-2^{1/3}}, \qquad d_2 = -\frac{2^{1/3}}{2-2^{1/3}}, \quad d_4 = 0. \tag{60}$$

Ultima cosa da far notare, ma molto interessante è che ogni integratore simplettico conserva il momento angolare, o più in generale è conservato qualsiasi generatore di trasformazioni puntuali lineari che sia una simmetria di H (e.g. rotazioni, dilatazioni, traslazioni). Per trasformazioni del tipo:

$$q_i \to q_i + \epsilon A_i(q) = q_i + \epsilon(\Omega_{ij}q_j + n_i), \tag{61}$$
$$p_i \to p_i - \epsilon\Omega_{ij}p_j \quad , \tag{62}$$

dove Ω è una matrice $N \times N$ e n è un vettore di coefficienti costanti. L'invarianza di H si scrive come:

$$\frac{\partial H}{\partial q_i}\delta q_i + \frac{\partial H}{\partial p_i}\delta p_i = \epsilon\left(\frac{\partial H}{\partial q_i}n_i + \frac{\partial H}{\partial q_i}\Omega_{ij}q_j - \frac{\partial H}{\partial p_i}\Omega_{ij}p_j\right) = 0 \quad . \tag{63}$$

Potete divertirvi a sostituire l'espressione data per le nuove coordinate date dall'integratore e far vedere che il generatore $G(p,q) = A_i p_i = \Omega_{ij}p_i q_j + n_i p_i$ è conservato.
Un modo magari più familiare per vedere tutto quel che abbiamo detto è quello dell'utilizzo delle trasformazioni canoniche.
Ricordiamo intanto che: $Q(q,p,t)$ e $P(q,p,t)$ sono trasformazioni canoniche se per una qualsiasi hamiltoniana $H(q,p,t)$ si può trovare un'hamiltoniana $K(Q,P,t)$ tale per cui le equazioni del moto per p e q data da H, si trasformano in quelle date da K in termini di Q e P.
In termini di x, definito sopra, abbiamo che:

$$\dot{x} = \Gamma\frac{\partial H}{\partial x} \quad \text{con} \quad \Gamma = \begin{pmatrix} 0 & -\mathrm{Id}_{N\times N} \\ \mathrm{Id}_{N\times N} & 0 \end{pmatrix} \tag{64}$$

Facciamo ora una trasformazione $X(x)$ (si generalizza a anche a trasformazioni dipendenti dal tempo). Abbiamo quindi che:

$$\dot{X}_i = \frac{\partial X_i}{\partial x_j}\dot{x}_j = \frac{\partial X_i}{\partial x_j}\Gamma_{jl}\frac{\partial H}{\partial x_l} = \frac{\partial X_i}{\partial x_j}\Gamma_{jl}\frac{\partial H}{\partial X_k}\frac{\partial X_k}{\partial x_l} = J_{ij}\Gamma_{jl}J^T_{lk}\frac{\partial H}{\partial X_k} = (J\Gamma J^T)_{ik}\frac{\partial H}{\partial X_k} \tag{65}$$

Per cui si arriva a:

$$J\Gamma J^T = \Gamma, \tag{66}$$

equazione che, in termini di matrici, definisce il gruppo simplettico. Diciamo questo perché tra simpletticità e canonicità c'è un legame stretto: $X(x,t)$ è canonica se la sua matrice jacobiana è simplettica a tutti i tempi a meno di una costante cioè: $J\Gamma J^T = \alpha\Gamma$. Inoltre se abbiamo che:

$$\{A,B\}_x = \{A,B\}_X, \tag{67}$$

la nostra trasformazione è simplettica. Ma sappiamo anche che verificare le parentesi di Poisson ci dà la canonicità della trasformazione:

$$X_i = \{X_i, H\}_x = \{X_i, H\}_X = \Gamma_{ij} \frac{\partial H}{\partial X_j} \tag{68}$$

Facciamo quindi un esempio semplice per far vedere come il metodo di Eulero non sia simplettico ma quello da noi implementato sopra lo sia. Per semplicità consentitemi di considerare (piuttosto che N corpi) la seguente:

$$H = \frac{p^2}{2} - \frac{1}{q}. \tag{69}$$

Il metodo di Eulero prevede:

$$p(t + \delta t) = p(t) - \delta t \frac{\partial H(q(t), p(t))}{\partial q} \tag{70}$$

$$q(t + \delta t) = q(t) + \delta t \frac{\partial H(q(t), p(t))}{\partial p} \tag{71}$$

Nel nostro caso abbiamo:

$$p(t + \delta t) = p(t) - \delta t \frac{1}{q^2(t)} \tag{72}$$

$$q(t + \delta t) = q(t) + \delta t p(t) \tag{73}$$

Per verificare la canonicità va dunque calcolata la:

$$\begin{aligned} \{q(t + \delta t), p(t + \delta t)\} &= \{q(t), p(t)\} - \delta t\{q(t), q^{-2}(t)\} + \delta t\{p(t), p(t)\} - \delta t^2\{p(t), q^{-2}(t)\} \\ &= 1 - \delta t^2 2q^{-3}(t). \end{aligned} \tag{74}$$

Vediamo dunque che la trasformazione non è canonica a tutti gli ordini in δt. Il metodo di Eulero simplettico invece ci dice:

$$p(t + \delta t) = p(t) - \delta t \frac{\partial H(q(t), p(t + \delta t))}{\partial q} \tag{75}$$

$$q(t + \delta t) = q(t) + \delta t \frac{\partial H(q(t), p(t + \delta t))}{\partial p} \tag{76}$$

Che nel nostro caso diventa:

$$p(t + \delta t) = p(t) - \delta t \frac{1}{q^2(t)} \tag{77}$$

$$q(t + \delta t) = q(t) + \delta t p(t + \delta t) = q(t) + \delta t p(t) - \delta t^2 \frac{1}{q^2(t)} \tag{78}$$

Per cui abbiamo:

$$\begin{aligned} \{q(t + \delta t), p(t + \delta t)\} = &\{q(t), p(t)\} - \\ &- \delta t(\{q(t), q^{-2}(t)\} - \{p(t), p(t)\}) - \\ &- \delta t^2(\{p(t), q^{-2}(t)\} + \{q^{-2}(t), p(t)\} - \{q^{-2}(t), q^{-2}(t)\}) \\ &= 1 \end{aligned} \tag{79}$$

Si vede bene dunque che la trasformazione è canonica per ogni δt; dato che il termine che prima rompeva la canonicità ora si cancella con un altro termine, mentre gli altri sono tranquillamente nulli. Potete se volete verificare che Eulero simplettico è dato dalla seguente funzione generatrice:

$$F_2(q(t), p(t + \delta t)) = q(t)p(t + \delta t) + \delta t H(q(t), p(t + \delta t)). \tag{80}$$

Il problema di questo metodo è però che non è reversibile. Si può però combinare con la variante in cui si usa $q(t + \delta t)$ per aggiornare $p(t)$, dando così vita ad un metodo del secondo ordine che è il leapfrog citato prima. Il quale è reversibile. Vediamo ora i risultati, dal punto di vista delle conservazioni, di due integratori (Eulero simplettico e Yoshida del quarto ordine):

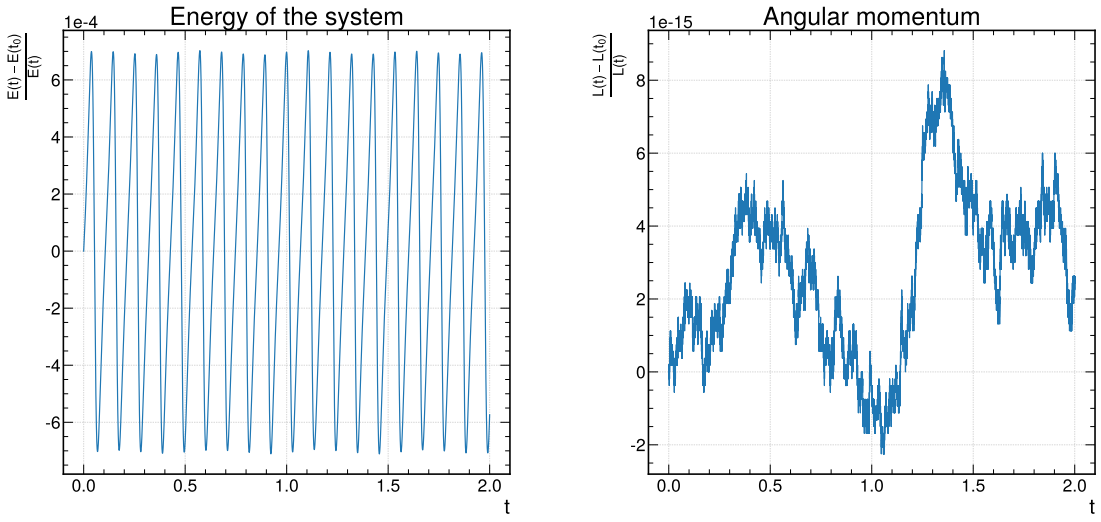

Figura 5: Conservazione di energia e momento angolare con l'integratore di Eulero simplettico.

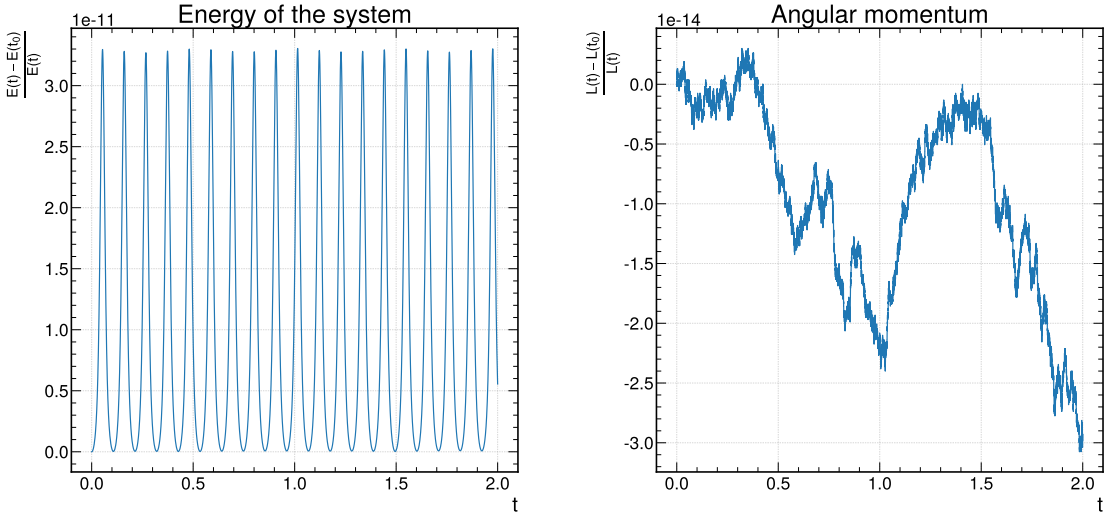

Figura 6: Conservazione di energia e momento angolare con l'integratore di Yoshida del quarto ordine.

Vediamo dunque come la conservazione dell'energia sia molto migliore nel caso di un integratore di ordine superiore, mentre la variazione di momento angolare è sempre dello stesso ordine circa. Avrete notato che lo stile dei plot è leggermente diverso, questo è dovuto al fatto che, per ottenere una migliore leggibilità, si è usato:

```
import mplhep
plt.style.use(mplhep.style.CMS)
```

Ok che è pallettaro ma onore al merito.

12.3 Più che una funzione

Le classi sono molto utili perché ci permettono di poter fare più cose che una funzione, ma possono anche essere chiamate come una funzione; ad esempio grazie al metodo "__call__" l'istanza che creiamo sarà una funzione. In una delle appendici è presente un codice che esegue un'interpolazione lineare; se andiamo a vederlo e ci fermiamo un attimo a pensare notiamo subito che quella funzione ogni volta che vine chiamata rifà tutto il conto necessario per trovare i coefficienti del polinomio in tutto l'intervallo. Con l'utilizzo di una classe possiamo calcolarci nel costruttore i coefficienti del polinomio e poi chiamare la funzione creata, la quale sa già i coefficienti del polinomio e quindi sarà effettivamente più veloce. Vediamo il caso, un po' più complicato ma più interessante, di una spline cubica (anche se a mali estremi un'interpolazione lineare funziona sempre, ordini più alti possono dare problemi):

```
class CubicSpline:
    '''
    1-D interpolating natural cubic spline

    Parameters
    ----------
    xx : 1darray
        value on x must be strictly increasing
    yy : 1darray
        value on y

    Example
    -------
    >>>import numpy as np
    >>>import matplotlib.pyplot as plt
    >>>x = np.linspace(0, 1, 10)
    >>>y = np.sin(2*np.pi*x)
    >>>F = CubicSpline(x, y)
    >>>print(F(0.2))
    0.9508316728694627

    >>>z = np.linspace(0, 1, 100)
    >>>plt.figure(1)
    >>>plt.title('Spline interpolation')
    >>>plt.xlabel('x')
    >>>plt.ylabel('y')
    >>>plt.plot(z, F(z), 'b', label='Cubic')
    >>>plt.plot(x, y, marker='.', linestyle='', c='k', label='data')
    >>>plt.legend(loc='best')
    >>>plt.grid()
    >>>plt.show()
    '''

    def __init__(self, xx, yy):

        self.x = xx                     # x data
        self.y = yy                     # y data will be the constant of polynomial
        self.N = len(xx)                # len of data
        alpha  = np.zeros(self.N-1)     # auxiliar array
        self.b = np.zeros(self.N-1)     # linear term
        self.c = np.zeros(self.N)       # quadratic term
        self.d = np.zeros(self.N-1)     # cubic term
        l      = np.zeros(self.N)       # auxiliar array
        z      = np.zeros(self.N)       # auxiliar array
        mu     = np.zeros(self.N)       # auxiliar array

        if not np.all(np.diff(xx) > 0.0):
            raise ValueError('x must be strictly increasing')

        dx = xx[1:] - xx[:-1]
        a = yy

        for i in range(1, self.N-1):
            alpha[i] = 3*(a[i+1] - a[i])/dx[i] - 3*(a[i] - a[i-1])/dx[i-1]

        l[0]  = 1.0
        z[0]  = 0.0
        mu[0] = 0.0

        for i in range(1, self.N-1):
            l[i]  = 2.0*(xx[i+1] - xx[i-1]) - dx[i-1]*mu[i-1]
            mu[i] = dx[i]/l[i]
            z[i]  = (alpha[i] - dx[i-1]*z[i-1])/l[i]

        l[self.N-1] = 1.0
        z[self.N-1] = 0.0
        self.c[self.N-1] = 0.0

        #Coefficient's computation
        for i in range(self.N-2, -1, -1):

            self.c[i] = z[i] - mu[i]*self.c[i+1]
            self.b[i] = (a[i+1] - a[i])/dx[i] - dx[i]*(self.c[i+1] + 2.0*self.c[i])/3.0
            self.d[i] = (self.c[i+1] - self.c[i])/(3.0*dx[i])


```

```
    def __call__(self, x):
        '''
        x : float or 1darray
            when we want compute the function
        '''
        n = self.check(x)

        if n == 1 :

            for j in range(self.N-1):
                if self.x[j] <= x <= self.x[j+1]:
                    i = j
                    break

            q = (x - self.x[i])
            return self.d[j]*q**3.0 + self.c[j]*q**2.0 + self.b[j]*q + self.y[j]

        else:
            F = np.zeros(n)
            for k, x1 in enumerate(x):

                for j in range(len(self.x)-1):
                    if self.x[j] <= x1 <= self.x[j+1]:
                        i = j
                        break

                q = (x1 - self.x[i])
                F[k] = self.d[j]*q**3.0 + self.c[j]*q**2.0 + self.b[j]*q + self.y[j]

            return F


    def check(self, x):
        try :
            n = len(x)
            x_in = np.min(self.x) <= np.min(self.x) and np.max(self.x) >= np.max(x)
        except TypeError:
            n = 1
            x_in = np.min(self.x) <= x <= np.max(self.x)

        # if the value is not in the correct range it is impossible to count
        if not x_in :
            errore = 'Value out of range'
            raise Exception(errore)

        return n
if __name__ == '__main__':

    x = np.linspace(0, 1,10)
    y = np.sin(2*np.pi*x)

    z = np.linspace(0, 1, 1000)
    G = CubicSpline(x, y)

    print(G(0.2))

    plt.figure(1)
    plt.title('Spline interpolation')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.plot(z, G(z), 'r', label='Cubic')
    plt.plot(x, y, marker='.', linestyle='', c='k', label='data')
    plt.legend(loc='best')
    plt.grid()
    plt.show()


    #==================================================
    # Cfr scipy and our spline
    #==================================================

    from scipy.interpolate import InterpolatedUnivariateSpline
    s3 = InterpolatedUnivariateSpline(x, y, k=3)
    plt.figure(2)
    plt.subplot(211)
    plt.title("Spline interpolation N = 10", fontsize=15);
```

```
    plt.ylabel("error", fontsize=15)
    plt.plot(z, G(z)-np.sin(2*np.pi*z), 'b', label='CubicSpline');
    plt.plot(z, s3(z)-np.sin(2*np.pi*z), 'r', label='Scipy');
    plt.grid();plt.legend(loc='best');
    plt.subplot(212);
    plt.title("Spline interpolation N = 30", fontsize=15)

    x = np.linspace(0, 1, 30)
    y = np.sin(2*np.pi*x)
    G = CubicSpline(x, y)
    s3 = InterpolatedUnivariateSpline(x, y, k=3)

    plt.ylabel("error", fontsize=15)
    plt.xlabel('x', fontsize=15)
    plt.plot(z, G(z)-np.sin(2*np.pi*z), 'b', label='CubicSpline');
    plt.plot(z, s3(z)-np.sin(2*np.pi*z), 'r', label='Scipy');
    plt.grid();plt.legend(loc='best')
    plt.show()

[Output]
0.9508316728694627
```

Fondamentalmente che succede: noi creiamo G, istanza della classe CubicSpline, nel farlo chiamiamo il costruttore il quale calcola i coefficienti dei polinomi interpolanti. Ora però grazie al metodo "__call__" G è un oggetto chiamabile. Avete presente quando in altri codici passando funzioni ad altre funzioni nella documentazione scrivevamo callable? Ecco è proprio questo, è come se la vostra funzione fosse il metodo "__call__" della classe. Quindi chiamando l'istanza viene eseguito il metodo "__call__" il quale, nel nostro caso, sa già i dati necessari e sa calcolarci la spline. Giusto per completezza precisiamo che questa spline cubica non è esattamente la stessa spline cubica di scipy che abbiamo visto sopra; questa si chiama spline naturale e ai bordi si comporta un pò diversamente rispetto a scipy (meglio anche). Per vederlo calcoliamo la differenza fra l'interpolazione calcolata in z (l'array nel codice) e i valori che restituisce "np.sin(2*np.pi*z)"; Facciamo due casi, interpoliamo prima 10 e poi 30 punti. Il codice sopra mostrato è riportato nella cartella interpolazioni, benché mostrato in questa sezione. Inoltre è riportato la stessa implementazione per il caso lineare.