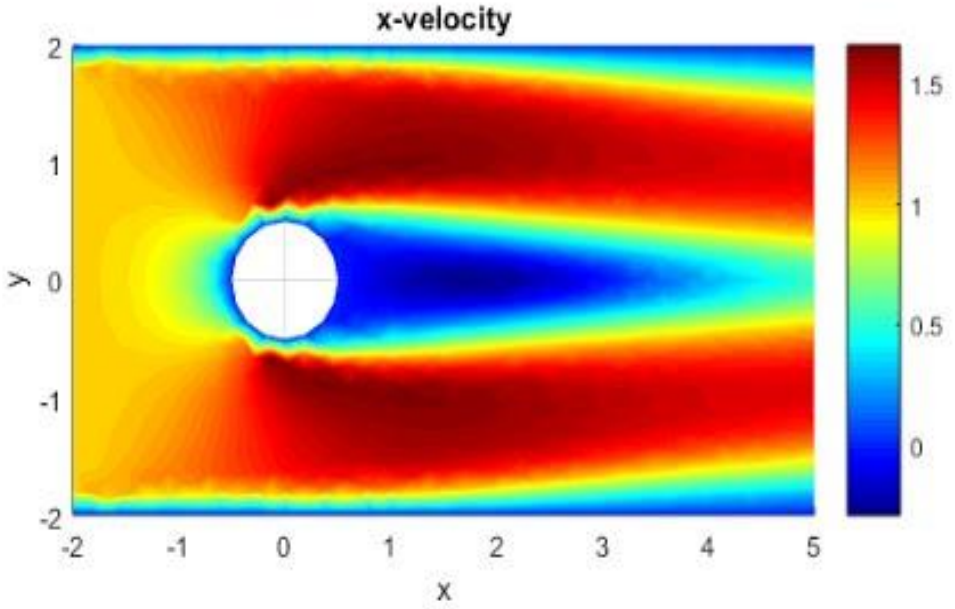
Fisica Computazionale 101

analisi del problema degli n-corpi e delle equazioni di Navier-Stokes attraverso i metodi computazionali



Liceo Scientifico G. A. Cavazzi-Sorbelli, Pavullo nel Frignano (Modena)

Patrick Uguzzoni 5^B Liceo, *Fisica Computazionale*

ABSTRACT

La fisica computazionale è una branca relativamente recente della fisica ma che, con il passare del tempo e l'aumento della complessità delle scoperte fisiche, è considerata sempre più essenziale. La mia lezione si concentrerà quindi su questa disciplina e soprattutto sui suoi metodi; del modo in cui il suo studio venga sempre più diffuso anche come corso base in molte università al mondo. In particolare porterò l'approccio di questa fisica ai problemi multi-corpo e alla modellizzazione dei flussi in contesti di fluidodinamica, anche attraverso l'utilizzo di software specifici programmati, ad esempio, attraverso python.

Nel caso dei problemi multi-corpo parlerò di come una soluzione sia spesso introvabile secondo metodi analitici, ma che attraverso una simulazione si possa approssimare la soluzione stessa con una precisione che aumenta all'aumentare della qualità della simulazione (sia a livello di software con l'ottimizzazione del codice, che a livello hardware con l'aumento della potenza di calcolo).

Mentre per la questione dei flussi mi concentrero' soprattutto sul problema delle equazioni di Navier-Stokes (che sono ciò che originano la principale necessita' della computazione all'interno della fluidodinamica), di come attraverso lo studio in 3 parti di una simulazione (una prima parte di preparazione alla simulazione, in cui vengono ad esempio elaborati i volumi, una seconda parte che rappresenta la simulazione vera e propria e un'ultima parte di analisi dei dati) basata sulla discretizzazione del modello reale possa produrre un risultato sufficientemente affidabile delle equazioni la cui soluzione analitica rappresenta tutt'oggi uno dei problemi del millennio.

INDICE

0. README.TXT

Questa prima parte è utile solo se si desidera riprodurre ciò che viene indicato nelle parti successive, e serve per definire gli ambienti di lavori e alcuni tecnicismi che vengono altrimenti dati per scontato nelle sezioni seguenti, similmente questa parte non viene considerata parte effettiva della trattazione; a differenza di ciò considero una parte integrante del lavoro i codici con i relativi commenti che lascio in allegato.

0.1. Linguaggi ed environment

Definiamo innanzitutto il linguaggio di programmazione utilizzato, ovvero python e perchè ho scelto di utilizzarlo.

È bene partire premettendo che non vi è un linguaggio di programmazione in assoluto migliore di tutti gli altri e che la scelta del linguaggio avviene per questo in base a ciò che si ha bisogno di fare; in questo caso python funziona particolarmente bene perché:

- È un linguaggio definito di quarta generazione ed è quindi particolarmente *user-friendly* rispetto a quelli di prima, seconda o terza generazione;
- Possiede alle proprie spalle una comunità particolarmente vivace che permette un facile riscontro online;
- Tra tutti i linguaggi è forse quello più propenso allo sviluppo di *librerie* ovvero file che permettono di ridurre la complessità e la lunghezza dei codici introducendo delle più facili ed intuitive scorciatoie.

Le librerie saranno particolarmente utili al nostro progetto, questo perchè python è un linguaggio che lavora senza avere necessariamente bisogno di un obiettivo; a differenza di programmi come arduino, il cui obiettivo è chiaramente quello di programmare la scheda fisica perché possa muovere o accendere qualche altra parte, python non ha reale bisogno di questo riscontro, motivo per cui avremo bisogno di environment particolari come *Jupyter Notebook* che ci permettono di visualizzare in modo effettivo ciò che programmiamo.

In questo caso, il fatto che dobbiamo realizzare una simulazione fisica rende python particolarmente versatile per la sua facilità di uso e il vasto numero di librerie già esistenti, è tuttavia utile notare come in realtà, nell'effettivo sviluppo di queste simulazioni, sia molto più comune l'utilizzo di C++, un linguaggio di terza generazione che permette una velocità di calcolo indubbiamente maggiore.

0.2. Librerie

Nel nostro caso vengono utilizzate tre librerie, di cui il problema multi-corpo ne userà soltanto due.

La prima è Numpy, definitivamente la libreria più utilizzata per le simulazioni, che permette di avere comodamente sotto mano un vasto numero di costanti fisiche e di facili comandi per controllare semplici calcoli che avvengono molto di frequente negli ambienti, appunto, fisici. Questa è una libreria sicuramente essenziale per il nostro scopo e che rende il codice sicuramente più facile da realizzare e da leggere.

La seconda è MatPlotLib ovvero la versione più gratuita ed *open-source* di MatLab, un altro linguaggio di programmazione di quarta generazione usato più spesso per visualizzare i dati ottenuti da simulazioni python.

La terza infine è utilizzata, come già detto, solo per la simulazione di Navier-Stokes, ovvero tqdm che permette semplicemente di avere la barra di progresso della simulazione e che torna utile per

controllare che non succeda nulla di strano nel mentre; invece per il problema multi-corpo ho ritenuto fosse più interessante avere il progressivo sviluppo dei grafici anche per poter osservare l'effettivo andamento delle traiettorie.

0.3. Ma quindi cosa devo fare?

Passiamo ad un po' di pratica, ovvero come passare ad avere un effettivo *environment* in cui poter inserire e programmare il proprio codice.

Dichiaro subito che attualmente lavoro su un MacBook, per cui la procedura descritta successivamente resterà senza effettivi link ed è quella seguita da me, tuttavia per PC Windows la procedura ***dovrebbe*** essere simile, se non uguale, mentre per Linux (o altri) mi spiace, ma non è un ambiente che conosco particolarmente bene. Altra pratica che tornerà sicuramente utile è quella di consultare quelle che vengono chiamate *documentation* ovvero siti web sviluppati per essere veri e propri libretti di istruzioni per un determinato linguaggio, per trovarli vi basta cercare su internet *nome linguaggio* + *documentation* e solitamente il primo link (che dovrebbe seguire qualcosa sulle tracce del *NomeLinguaggio.docs.org*) è quello giusto. Ora, se vi basta fare ciò che viene fatto in questa guida allora bene, la procedura è assai semplice, altrimenti se desiderate *smanettare* un po' di più sulla cosa allora è probabile che dobbiate fare qualche ricerca in più sull'argomento; ad ogni modo dovrebbe essere sufficiente scaricare *Anaconda* (raggiungibile ad *anaconda.com*), ovvero un software che offre un ambiente con già preconfezionati sia python, che tutte le librerie citate, che *Jupyter Notebook* ovvero il software che useremo per eseguire e visualizzare i codici con i loro grafici.

Nel mentre che scaricate anaconda, è buona norma creare una cartella ad hoc nel proprio desktop o dove preferite (ma ricordate dove la mettete!). A questo punto potete aprire anaconda e aprire *Jupyter Notebook*, vi aprirà una finestra web e qui potete selezionare la cartella che avete creato in precedenza. In alto a destra cliccate *New* e selezionate *Python 3*, a questo punto vi genererà un file

.ipynb ovvero una versione modificata di un *.py* (formato nativo di python) che conserva un paio di variabili in più, comunque questo non è particolarmente interessante.

A questo punto potete riscrivere i codici a mano nel file appena aperto oppure inserire nella cartella i file che lascio in allegato (c'è bisogno di una cartella per ciascun file). Per eseguire il codice sarà sufficiente aprirlo (sempre in Jupyter Notebook), selezionare il blocco di codice cliccando con il tasto sinistro del mouse e premere il piccolo triangolo nella barra in alto, questo dovrebbe avviare il codice; per usarlo nuovamente è necessario prima fermarlo con il quadratino a fianco del triangolo e poi tornare ad eseguirlo.

1. L'APPROCCIO COMPUTAZIONALE

Ma perché è così crescente l'importanza dell'approccio computazionale in fisica? A livello molto pratico la risposta sta nel fatto che alcune equazioni sono di difficile risoluzione analitica e, ovviamente, un pc difficilmente potrà ottenere la soluzione analitica di un problema ancora irrisolto, allora si utilizza lo stesso pc per calcolare l'equazione per una quantità altissima di numeri fino ad ottenere una buona approssimazione della soluzione.

1.1. Fisica Computazionale

Ma cos'è la fisica computazionale? La fisica computazionale è considerata una disciplina che unisce fisica, per lo studio della realtà, matematica, per i metodi, e informatica per gli strumenti.

Come disciplina è sufficientemente recente ma la sua importanza è sempre crescente poiché è in grado di fornire un nuovo approccio alla soluzione di quei problemi per cui ci sia bisogno altrimenti di una soluzione analitica ancora non trovata. Tuttavia è estremamente difficile ottenere una soluzione precisa di un problema attraverso la computazione poiché spesso richiede l'utilizzo di

---

metodi a forza bruta (il classico metodo *a tentativi* ma fatto per migliaia e migliaia di numeri) o di radici (il cui valore è quasi sempre approssimato) che generano errori piccoli ma che ripetuti per le migliaia di calcoli necessari ad ottenere la simulazione rendono la soluzioni inutilizzabile.

Un altro grosso problema verso cui si va spesso incontro è la realizzazione degli algoritmi (gli schemi di calcolo) grazie ai quali ottenere l'effettiva simulazione, sia per un elevato numero di variabili o una difficoltà matematica eccessiva.

1.2. Simulazioni

L'approccio computazionale è usato quindi soprattutto per effettuare delle simulazioni di situazioni reali per la cui predizione servirebbe la soluzione di equazioni senza soluzione analitica e dalla difficile approssimazione, e straordinariamente funzionano straordinariamente bene.

Il concetto alla base si può dire essere lo stesso degli esperimenti mentali condotti ad esempio da Galileo; si prende una situazione che si vuole analizzare, né si riducono le variabili rispetto alla realtà e la si risolve per approssimazione, nel nostro caso. Viene utilizzato un computer per ottenere queste grandi approssimazioni che richiederebbero altrimenti ben più tempo per essere calcolate. Tuttavia anche un computer ha i suoi limiti e i due più grandi limiti sono quello a livello software e quello a livello hardware:

- A livello software le limitazioni agiscono su più livelli, partendo ad esempio dalla scelta stessa del linguaggio di programmazione, che come è stato spiegato sopra può essere più limitato di un altro nell'eseguire un compito rispetto ad un altro; la grossa limitazione tuttavia sta nel codice in sé per sé, in particolare nel processo logico che si sceglie per giungere alla soluzione. Ad esempio, se volessimo assegnare il valore "2" ad una variabile ci sono modi infiniti per farlo, ma è ovvio che utilizzare stringhe come `x=2` è estremamente più efficace di usare stringhe come `x=(sqrt(2^2)+(2^3-2^4)/2)*(-1)`; calcolare la

prima richiede sicuramente meno tempo (in questo caso si parla di qualche frazione di secondo, ma ricordiamo che di questi conti in una simulazione se ne fanno parecchi e dunque questo piccolo scarto alla fine può fare la differenza tra una simulazione calcolabile in tempi umani e una che non lo è).

- A livello hardware invece la cosa è un po' meno intuitiva vista la forma estremamente compatta che hanno assunto i computer ai giorni nostri, ma questi rimangono comunque dei dispositivi fisica dotati di capacità limitate, e *la potenza di calcolo* è, detta in modo molto terra-terra, la quantità di calcoli elementari che un sistema è in grado di calcolare al secondo. Quindi un computer con un disco più lento, meno memoria RAM o con una cpu con una bassa capacità di distribuire il calcolo sarà ovviamente più lento nei conti ma anche limitato nella loro complessità.

2. PROBLEMA degli N-CORPI

In fisica si parla del problema degli n-corpi per riferirsi allo studio e alla predizione dei moti di più corpi celesti che interagiscono tra di loro gravitazionalmente.

Ora alcuni casi particolari (come il problema a 2 corpi o alcuni casi ancora più particolari del problema a 3 corpi) sono stati risolti, sebbene non abbiamo ancora una soluzione analitica per il problema generale, questo a causa della crescente difficoltà matematica. Il punto sta nel fatto che, ad esempio, possiamo studiare il sistema solare come più problemi a 2 corpi ma questo tipo di soluzione avrà un errore non particolarmente trascurabile e varrà per un periodo di tempo estremamente limitato (più è grande l'errore iniziale minore sarà il tempo per il quale si potrà predire un evento poichè l'errore tende ovviamente ad accumularsi nel tempo).

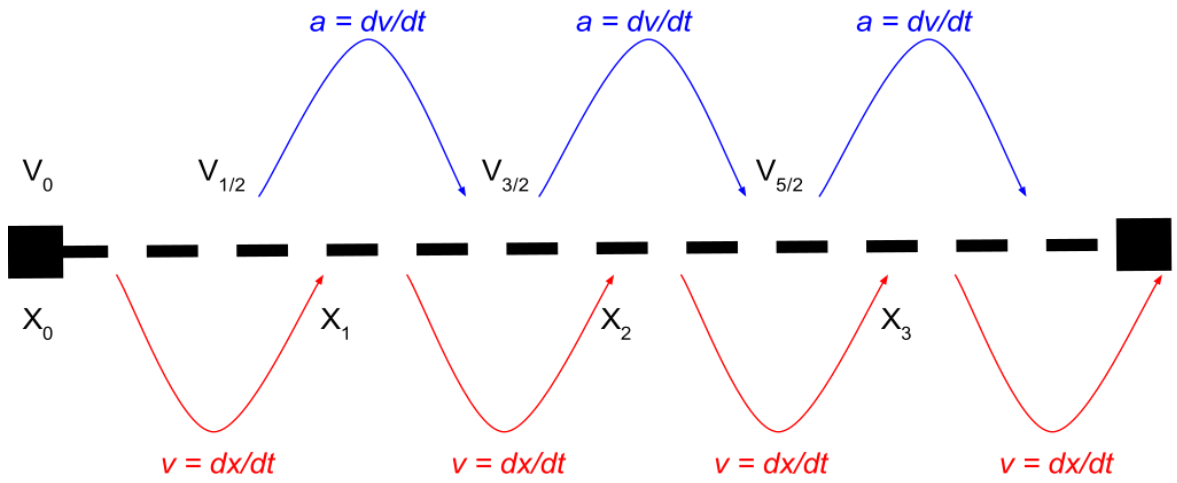
Il codice utilizzato è *relativamente* banale e l'ho commentato a sufficienza perché chi desidera comprenderlo riga per riga possa farlo con sufficiente semplicità; i commenti si occupa di spiegare il funzionamento delle stringhe di codice e sorvolano particolarmente sulla matematica e i metodi propri matematici utilizzati perché banalmente mi sono stati consigliati da quella vivace comunità che programma in python già nominata.

2.1. Simulazione: metodi generali

La simulazione che ho utilizzato per il problema degli n-corpi riduce il problema in un piano cartesiano, che è ovviamente già un'estrema semplificazione della realtà ma che per il nostro scopo funziona più che bene. Ora il movimento non è ovviamente fluido e ovviamente il *movimento* degli oggetti non è altro che in realtà dei piccolissimi spostatamenti messi in fila; per ognuno di questi singoli movimenti il computer si occupa di ricalcolare ogni singolo valore che descrive la posizione dei corpi e ne ricalcola tutte le interazioni.

Le variabili che vengono assegnate ad inizio di ogni simulazione arbitrariamente ad ogni particella sono una massa (il cui valore totale nel sistema è 20), e una posizione e una velocità casuali.

Il metodo che ho seguito quindi semplicemente calcola attraverso l'uso di matrici e vettori la posizione e l'accelerazione che i corpi subiscono a causa dell'interazione che avviene tra di loro, dettata dalla formula di gravitazione universale di Newton. Per il calcolo invece di quei minimi spostamenti utilizzo un caso particolare del *metodo Leapfrog*, che prende il nome di *kick-drift-kick* che funziona calcolando ogni volta parte della velocità, l'accelerazione, e la restante parte di velocità, ottenendo così una buona simulazione dei moti delle particelle.



Infine altro metodo utilizzato è quello di calcolare l'energia totale del sistema, data da energia cinetica e potenziale; questo è un conto semplicissimo da realizzare e da integrare nel codice e in termini di risorse non costa praticamente nulla, è infatti buona norma utilizzarlo per verificare che la simulazione mantenga un certo equilibrio ed è solitamente indicatore di una buona simulazione (ricordo che nel nostro caso, essendo un sistema chiuso, l'energia totale del sistema deve essere costante).

2.2. Simulazione: metodi di programmazione

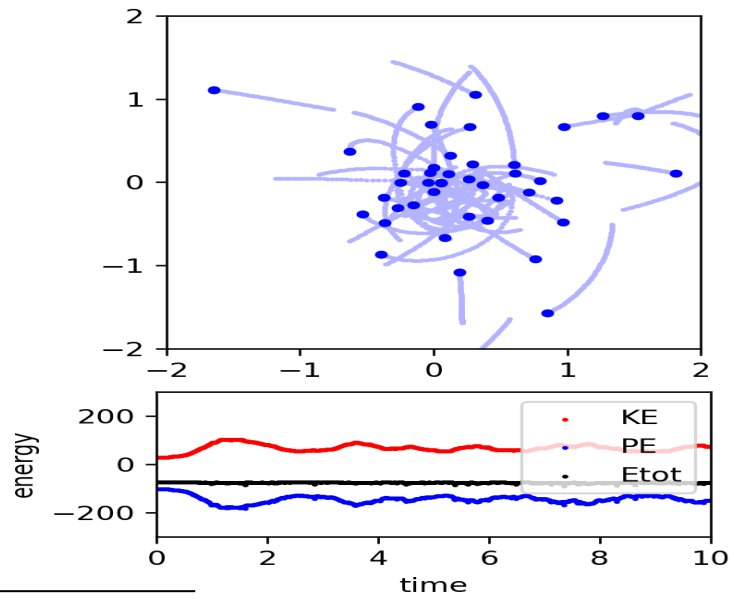
Python è un linguaggio che *vuole* essere leggibile e per cui i rientri a capo linee non sono solo buona norma, ma sono essenziali perché il codice venga compilato (ovvero perchè il computer non rilevi parti di codice illeggibili); utilizziamo dunque i rientri a capo per orientarci all'interno del codice, in cui i rientri, assieme all'ordine in cui vengono scritte le stringhe di codice, definiscono il percorso logico completo che è stato utilizzato per realizzare il codice. È ovviamente diviso in 4 sezioni principali, definite da `import` e `def`:

- Nel nostro caso gli `import` sono i comandi che servono per segnalare quali librerie vogliamo utilizzare nel nostro codice (e nel nostro caso gli assegno anche un nome più breve per semplicità).
- I `def` invece vanno appunto a definire una funzione, nel nostro caso sono 3 le funzioni che definiamo, ovvero quelle chiamate `getAcc`, `getEnergy` e `main`.

Ora, sebbene l'impressione sia che il computer esegua tutte le stringhe contemporaneamente, in realtà ne esegue una alla volta e dunque anche l'ordine è importante; per questo le librerie è necessario che vengano dichiarate all'inizio, per poi dichiarare la funzione che ci permette di ottenere l'accelerazione, quella dell'energia e infine quella che utilizza entrambe per realizzare la simulazione e inserirla in un grafico.

Altro concetto molto importante da tenere a mente è che il grafico è uno strumento che serve solamente a noi che guardiamo, è bene infatti ricordare che la simulazione in sé per sé è in realtà soltanto un enorme numero di dati, un qualcosa che andrebbe posto in un foglio di calcolo (come excel) per poterne ricavare il grafico. In questo caso il grafico è realizzato in automatico ma voglio sia chiaro che il grafico si inserisce come uno step successivo e non necessario alla realizzazione della simulazione.

L'altro grande scoglio logico è nelle variabili utilizzate nella funzione `main`. Alcune si spiegano da sole, ma è bene notare come *G* sia stato approssimato ad 1 (approssimazione fatta per evitare di dover rendere tutto in scala planetaria; come il *dt* sia quel periodo minimo di tempo al cui diminuire aumenta la precisione e la complessità di calcolo della simulazione; e infine come vi sia ciò che chiamo *softening length* in assenza di una buona traduzione. Quest'ultimo valore serve perchè nella nostra simulazione utilizziamo delle particelle puntiforme, e da questo deriva il problema per niente banale che, all'avvicinarsi dei due corpi, la forza di gravitazione di Newton tende all'infinito; nella realtà il problema non si pone poiché due pianeti non possono essere abbastanza vicini ed avere abbastanza massa da causare un problema, ma nella nostra simulazioni le particelle puntiformi possono sovrapporsi mantenendo la loro massa, che causa sbalzi nell'accelerazione. Questa softening length quindi è proprio il valore minimo per definizione per cui due particelle possano avvicinarsi e che evita il problema appena descritto. Le ultime parti della funzione `main` servono invece per generare il grafico e per salvarne l'immagine nella medesima cartella del codice[1].



[1] L'immagine è quella prodotta dall'esecuzione del codice al termine della simulazione, si noti che viene calcolata una simulazione di 10 secondi, ma la cui computazione richiede molto di più.

3. EQUAZIONI di NAVIER-STOKES

Le equazioni di Navier-Stokes sono un sistema di tre equazioni di bilancio (moto, energia e materia) che descrivono un fluido viscoso lineare; con il termine fluido si fa riferimento ad un materiale che non abbia forma propria (quindi può essere, ad esempio, sia gassoso che liquido) mentre la linearità fa riferimento al rapporto tra sforzo fatto sul materiale e deformazione dello stesso (il fluido si dice lineare quando lo sforzo è lineare alla velocità di deformazione del fluido). Di base queste equazioni descrivono il modo in cui si comportano i fluidi in ambienti più o meno complessi e la loro soluzione analitica segnerebbe un grande passo nella modellizzazione di sistemi fluidodinamici, infatti rappresentano uno dei sette problemi del millennio.

Il problema della loro soluzione nasce dalla differenza tra sistemi continui e discreti; se consideriamo un sistema discreto la loro soluzione è spesso semplice perché possono essere divisi in "pezzi" individuali, dei "pacchetti" singoli, mentre nel caso di un sistema continuo questo non accade e possiamo quindi già dedurre che le stesse equazioni di Navier-Stokes non saranno altro che una semplificazione della realtà dei fluidi che si comportano come sistemi continui. Si può inoltre semplificare la questione ed ammettere che queste equazioni non siano altro che la seconda legge di Newton scritta in una forma applicabile anche ai sistemi continui; se consideriamo infatti la seconda legge di Newton come $\mathbf{F}=m\mathbf{a}$ diamo già per scontato che la massa del corpo sia costante, possiamo quindi scriverla più correttamente come:

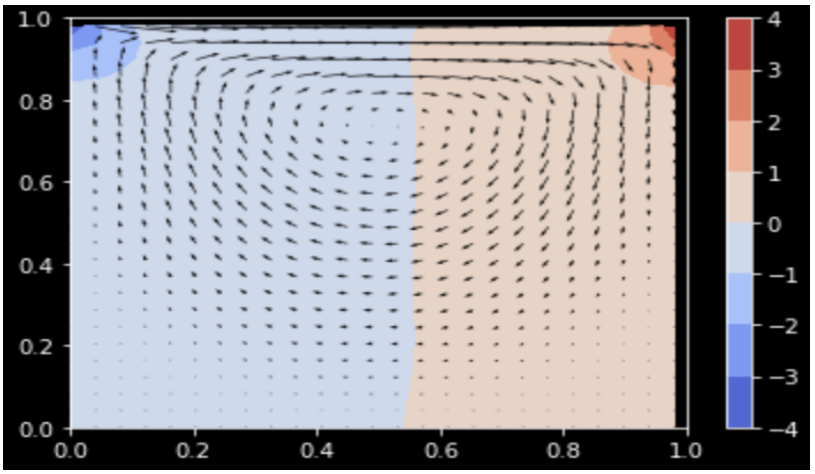
$$\vec{F} = \frac{d}{dt}(m\vec{v})$$

A questo punto se consideriamo un fluido piuttosto che un corpo ovviamente non ci interesserà più la sua massa quanto la sua densità ρ, e otterremo così la forza infinitesima f fatta su di una parte infinitesima di fluido, per la questione di continuità e discrezione di cui prima:

$$\vec{f} = \rho\frac{d\vec{v}}{dt}$$

Ora, riscrivendo questa formula in notazione vettoriale (la velocità di un fluido sarà in funzione di spazio e tempo) e considerando un paio di altre forze (pressioni e viscosità) si ottiene infine l'equazione di Navier-Stokes[2].

$$\frac{d\vec{v}}{dt} + (\vec{v}\cdot\vec{\nabla})\,\vec{v} = -\frac{1}{2}\vec{\nabla}P + \vec{v}\nabla^2 + \frac{1}{\rho}\vec{f}_{ext}$$



3.1. Simulazione: metodi generali

In questa simulazione viene utilizzato il metodo di proiezione di Chorin, che essenzialmente divide il conto dell'equazione calcolando prima le forze viscose e poi le pressioni (basti sapere che serve per dividere e rimettere insieme l'equazione).

[2] L'immagine è anche qui quella prodotta dall'esecuzione del codice, che tuttavia non richiede così tanto tempo quanto l'altro, seppur sia più complesso a livello logico ma ben più ottimizzato a livello di metodi e codice.

Il resto della matematica dietro a questo codice in realtà consiste nel dividere nuovamente i vettori nelle loro componenti e nel calcolo matriciale, mi limito a considerare il resto dei metodi per dati poiché sono vari e non sarei in grado di spiegarli particolarmente meglio; mi limito a dire che il metodo di Chorin lo descrivo meglio nei metodi di programmazione, che le derivate parziali vengono risolte usando uno schema a differenza centrale e che l'operatore di Laplace viene risolto mediamente uno stencil a 5 punti.

Per concludere questa sezione la simulazione è condotta considerando le condizioni di pressione di tutti i bordi come stabili e di Neumann tranne che per il sopra che sono di Dirichlet, mentre tutte le condizioni di bordo di pressione sono omogenee di Dirichlet tranne che per il sopra la cui velocità è definita; i bordi definiti in questa maniera simulano il bordo fisico del sistema e definiscono il limite letterale del sistema che analizziamo, lasciando tuttavia che il bordo superiore possa variare simulando un flusso sulla superficie.

3.2. Simulazione: metodi di programmazione

Il codice è sicuramente più esaustivo grazie ai commenti inseriti, rispetto a questa sezione che coprirà soltanto le parti più ardue; questo a causa del fatto che sarebbe altrimenti estremamente lungo descrivere tutti i passaggi logici del codice, soprattutto considerato che buona parte della matematica resta relativamente oscura.

Il metodo di Chorin innanzitutto viene utilizzato per poter considerare il fluido incomprimibile pur tuttavia calcolando un gradiente di pressione al suo interno, in particolare agisce in tre diversi momenti:

- Calcola una velocità intermedia utilizzando le equazioni del moto senza considerare il gradiente di pressione;
- Calcola una pressione intermedia risolvendo l'equazione della pressione usando Poisson;

- Utilizza la velocità intermedia e la pressione intermedia per ottenere una velocità finale e un gradiente di pressione pur considerando il fluido incomprimibile.

È anche interessante notare come questo codice segua uno schema logico completamente diverso: se nell'altro erano presenti tre "blocchi" di codici principali in questo è tutto racchiuso nella funzione main. Questo accade perché il codice è stato sottoposto ad una fase di *ottimizzazione* molto più lunga e quindi le funzioni tendono a essere condensate per aumentare l'efficienza del codice stesso.

In questo caso il grafico è anch'esso assolutamente diverso e rappresenta la sezione di un contenitore il cui unico lato "aperto" è quello superiore in cui viene eseguito lo sforzo sul fluido. Nel grafico viene così rappresentato sia il flusso tramite i vettori, che il gradiente di pressione tramite il colore.

4. CODICE degli N-CORPI

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

def getAcc( pos, mass, G, softening ):
        """
       Calcola l'accelerazione di ogni particella dovuta alla forza di Newton
        pos  è una matrice del tipo N x 3 di posizioni
        mass è un vettore di tipo N x 1 di masse
        G è la costante di gravitazione di Newton
        softening è la softening length
        a è una matrice del tipo N x 3 di accelerazioni
        """
        # posizioni r = [x,y,z] per tutte le particelle:
        x = pos[:,0:1]
        y = pos[:,1:2]
        z = pos[:,2:3]

        # matrice che contiene tutte le separazioni di particelle a coppie: r_j - r_i
        dx = x.T - x
        dy = y.T - y
        dz = z.T - z

        # matriche che contiene 1/r^3 per tutte le separazioni di particelle a coppie:
        inv_r3 = (dx**2 + dy**2 + dz**2 + softening**2)
        inv_r3[inv_r3>0] = inv_r3[inv_r3>0]**(-1.5)

        ax = G * (dx * inv_r3) @ mass
        ay = G * (dy * inv_r3) @ mass
        az = G * (dz * inv_r3) @ mass

        # riunisce le componenti della velocità:
        a = np.hstack((ax,ay,az))

        return a

def getEnergy( pos, vel, mass, G ):
        """
        Calcola l’energia cinetica (KE) e potenziale (PE) del sistema
        """
        # Energia cinetica:
        KE = 0.5 * np.sum(np.sum( mass * vel**2 ))


        # Energia potenziale:
        x = pos[:,0:1]
        y = pos[:,1:2]
        z = pos[:,2:3]

        dx = x.T - x
        dy = y.T - y
        dz = z.T - z

        # matrice che contine 1/r per tutte le separazioni delle particelle a coppie:
        inv_r = np.sqrt(dx**2 + dy**2 + dz**2)
        inv_r[inv_r>0] = 1.0/inv_r[inv_r>0]

        # la somma avviene in modo che ogni interazione venga contata una sola volta
        PE = G * np.sum(np.sum(np.triu(-(mass*mass.T)*inv_r,1)))

        return KE, PE;


def main():
        """ Simulazione principale """

        # Variabili della simulazione
```

```
N         = 50     # numero di particelle
t         = 0      # istante di inizio
tEnd      = 10.0   # istante di fine
dt        = 0.01   # minor dt possibile
softening = 0.1    # softening length
G         = 1      # costante di gravitazione di Newton
plotRealTime = True # fornisce i grafici in tempo reale

# Genera le restanti condizioni iniziali per le particelle
np.random.seed(17)            # imposta il seme per il generatore casuale

mass = 20.0*np.ones((N,1))/N  # la massa totale nel sistema è 20
pos  = np.random.randn(N,3)   # seleziona casualmente posizioni e velocità
vel  = np.random.randn(N,3)

# calcola uno schema a centro di massa
vel -= np.mean(mass * vel,0) / np.mean(mass)

# calcola le accelerazioni gravitazionali iniziali
acc = getAcc( pos, mass, G, softening )

# calcola l'energia del sistema
KE, PE  = getEnergy( pos, vel, mass, G )

# numero di dt
Nt = int(np.ceil(tEnd/dt))

# salva posizioni ed energie per realizzare le scie nel grafico
pos_save = np.zeros((N,3,Nt+1))
pos_save[:,:,0] = pos
KE_save = np.zeros(Nt+1)
KE_save[0] = KE
PE_save = np.zeros(Nt+1)
PE_save[0] = PE
t_all = np.arange(Nt+1)*dt

# preparare il grafico impostando le assi e il resto
fig = plt.figure(figsize=(4,5), dpi=80)
grid = plt.GridSpec(3, 1, wspace=0.0, hspace=0.3)
ax1 = plt.subplot(grid[0:2,0])
ax2 = plt.subplot(grid[2,0])

# loop principale in cui si utilizza il metodo kick-drift-kick
for i in range(Nt):
          # primo mezzo kick
          vel += acc * dt/2.0

          # drift
          pos += vel * dt

          # aggiorna le accelerazioni
          acc = getAcc( pos, mass, G, softening )

          # secondo mezzo kick
          vel += acc * dt/2.0

          # aggiorna il tempo
          t += dt

          # ottieni l'energia del sistema
          KE, PE  = getEnergy( pos, vel, mass, G )

          # salva le energie e le altre variabili per creare le scie sul grafico
          pos_save[:,:,i+1] = pos
          KE_save[i+1] = KE
          PE_save[i+1] = PE

          # realizza i grafici in tempo reale
          if plotRealTime or (i == Nt-1):
                    plt.sca(ax1)
                    plt.cla()
                    xx = pos_save[:,0,max(i-50,0):i+1]
                    yy = pos_save[:,1,max(i-50,0):i+1]
```

```
                plt.scatter(xx,yy,s=1,color=[.7,.7,1])
                plt.scatter(pos[:,0],pos[:,1],s=10,color='blue')
                ax1.set(xlim=(-2, 2), ylim=(-2, 2))
                ax1.set_aspect('equal', 'box')
                ax1.set_xticks([-2,-1,0,1,2])
                ax1.set_yticks([-2,-1,0,1,2])

                plt.sca(ax2)
                plt.cla()
                plt.scatter(t_all,KE_save,color='red',s=1,label='KE' if i == Nt-1 else "")
                plt.scatter(t_all,PE_save,color='blue',s=1,label='PE' if i == Nt-1 else "")
                plt.scatter(t_all,KE_save+PE_save,color='black',s=1,label='Etot' if i == Nt-1 else "")
                ax2.set(xlim=(0, tEnd), ylim=(-300, 300))
                ax2.set_aspect(0.007)

                plt.pause(0.001)


        # aggiunge la legenda
        plt.sca(ax2)
        plt.xlabel('time')
        plt.ylabel('energy')
        ax2.legend(loc='upper right')

        # salva l'immagine
        plt.savefig('nbody.png',dpi=240)


        return


if __name__== "__main__":
  main()
```

5. CODICE di NAVIER-STOKES

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

N_POINTS = 50
DOMAIN_SIZE = 1.0
N_ITERATIONS = 500
TIME_STEP_LENGTH = 0.001
KINEMATIC_VISCOSITY = 0.1
DENSITY = 1.0
HORIZONTAL_VELOCITY_TOP = 1.0

N_PRESSURE_POISSON_ITERATIONS = 50
STABILITY_SAFETY_FACTOR = 0.5

def main():
    element_length = DOMAIN_SIZE / (N_POINTS - 1)
    x = np.linspace(0.0, DOMAIN_SIZE, N_POINTS)
    y = np.linspace(0.0, DOMAIN_SIZE, N_POINTS)

    X, Y = np.meshgrid(x, y)

    u_prev = np.zeros_like(X)
    v_prev = np.zeros_like(X)
    p_prev = np.zeros_like(X)

    def central_difference_x(f):
        diff = np.zeros_like(f)
        diff[1:-1, 1:-1] = (
            f[1:-1, 2:  ]
            -
            f[1:-1, 0:-2]
        ) / (
            2 * element_length
        )
        return diff

    def central_difference_y(f):
        diff = np.zeros_like(f)
        diff[1:-1, 1:-1] = (
            f[2:  , 1:-1]
            -
            f[0:-2, 1:-1]
        ) / (
            2 * element_length
        )
        return diff

    def laplace(f):
        diff = np.zeros_like(f)
```

```
    diff[1:-1, 1:-1] = (
        f[1:-1, 0:-2]
        +
        f[0:-2, 1:-1]
        -
        4
        *
        f[1:-1, 1:-1]
        +
        f[1:-1, 2:  ]
        +
        f[2:  , 1:-1]
    ) / (
        element_length**2
    )
    return diff


maximum_possible_time_step_length = (
    0.5 * element_length**2 / KINEMATIC_VISCOSITY
)
if TIME_STEP_LENGTH > STABILITY_SAFETY_FACTOR * maximum_possible_time_step_length:
    raise RuntimeError("Stabilità non garantita")

for _ in tqdm(range(N_ITERATIONS)):
    d_u_prev__d_x = central_difference_x(u_prev)
    d_u_prev__d_y = central_difference_y(u_prev)
    d_v_prev__d_x = central_difference_x(v_prev)
    d_v_prev__d_y = central_difference_y(v_prev)
    laplace__u_prev = laplace(u_prev)
    laplace__v_prev = laplace(v_prev)

    # Prima parte di Chorin in cui calcola la velocità intermedia (u) senza
    # il gradiente di pressione
    u_tent = (
        u_prev
        +
        TIME_STEP_LENGTH * (
            -
            (
                u_prev * d_u_prev__d_x
                +
                v_prev * d_u_prev__d_y
            )
            +
            KINEMATIC_VISCOSITY * laplace__u_prev
        )
    )
    v_tent = (
        v_prev
        +
        TIME_STEP_LENGTH * (
            -
            (
                u_prev * d_v_prev__d_x
                +
```

```
            v_prev * d_v_prev__d_y
        )
        +
        KINEMATIC_VISCOSITY * laplace__v_prev
    )
)

# Condizioni di contorno di velocità: Omogenee di Dirichlet ovunque
# tranne che per il sopra la cui velocità è definita.
u_tent[0, :] = 0.0
u_tent[:, 0] = 0.0
u_tent[:, -1] = 0.0
u_tent[-1, :] = HORIZONTAL_VELOCITY_TOP
v_tent[0, :] = 0.0
v_tent[:, 0] = 0.0
v_tent[:, -1] = 0.0
v_tent[-1, :] = 0.0

d_u_tent__d_x = central_difference_x(u_tent)
d_v_tent__d_y = central_difference_y(v_tent)

# Calcola la pressione risolvendo con Poisson (seconda parte di Chorin)
rhs = (
    DENSITY / TIME_STEP_LENGTH
    *
    (
        d_u_tent__d_x
        +
        d_v_tent__d_y
    )
)

for _ in range(N_PRESSURE_POISSON_ITERATIONS):
    p_next = np.zeros_like(p_prev)
    p_next[1:-1, 1:-1] = 1/4 * (
        +
        p_prev[1:-1, 0:-2]
        +
        p_prev[0:-2, 1:-1]
        +
        p_prev[1:-1, 2:  ]
        +
        p_prev[2:  , 1:-1]
        -
        element_length**2
        *
        rhs[1:-1, 1:-1]
    )

    # Condizioni di contorno di pressione: condizioni di contorno omogeneo di Neumann
    # ovunque tranne che per il sopra,dove è di Dirichlet
    p_next[:, -1] = p_next[:, -2]
    p_next[0,  :] = p_next[1,  :]
    p_next[:,  0] = p_next[:,  1]
    p_next[-1, :] = 0.0
```

```
            p_prev = p_next


        d_p_next__d_x = central_difference_x(p_next)
        d_p_next__d_y = central_difference_y(p_next)

        # ultimo step di Chorin in cui si assicura l'incomprimibilità del fluido.
        u_next = (
            u_tent
            -
            TIME_STEP_LENGTH / DENSITY
            *
            d_p_next__d_x
        )
        v_next = (
            v_tent
            -
            TIME_STEP_LENGTH / DENSITY
            *
            d_p_next__d_y
        )

        # Condizioni di bordo di velocità: omogenee di Dirichlet ovunque
        # tranne che sopra che è definita
        u_next[0, :] = 0.0
        u_next[:, 0] = 0.0
        u_next[:, -1] = 0.0
        u_next[-1, :] = HORIZONTAL_VELOCITY_TOP
        v_next[0, :] = 0.0
        v_next[:, 0] = 0.0
        v_next[:, -1] = 0.0
        v_next[-1, :] = 0.0

        # Advance in time
        u_prev = u_next
        v_prev = v_next
        p_prev = p_next

    plt.style.use("dark_background")
    plt.figure()
    plt.contourf(X[::2, ::2], Y[::2, ::2], p_next[::2, ::2], cmap="coolwarm")
    plt.colorbar()

    plt.quiver(X[::2, ::2], Y[::2, ::2], u_next[::2, ::2], v_next[::2, ::2], color="black")
    plt.xlim((0, 1))
    plt.ylim((0, 1))
    plt.show()

if __name__ == "__main__":
    main()
```

BIBLIOGRAFIA

A. Sitiweb

A.1. Linguaggio di Programmazione e simili

Anaconda - https://www.anaconda.com

MatPlotLib - https://matplotlib.org

MatPlotLib documentation - https://matplotlib.org/stable/users/index

Numpy - https://numpy.org

Numpy documentation - https://numpy.org/doc/stable/

Python - https://www.python.org

Python documentation - https://docs.python.org/3/

Tqdm - https://tqdm.github.io

A.2. Pagine utili per i metodi

Problema degli n-corpi - https://it.wikipedia.org/wiki/Problema_degli_n-corpi

Equazioni di Navier-Stokes - https://it.wikipedia.org/wiki/Equazioni_di_Navier-Stokes

Metodo Chorin - https://en.wikipedia.org/wiki/Projection_method_(fluid_dynamics)

Metodo Leapfrog - https://en.wikipedia.org/wiki/Leapfrog_integration