Simulatore del movimento dei pianeti in un sistema solare

Gioele Mancino, Federico Mastroforti, Luca Tesei, Nico Tortorici

7 luglio 2023

Indice

1 Introduzione

1.1 Obiettivo del simulatore

Lo scopo del programma quello di simulare un sistema di N corpi planetari o stellari secondo la fisica newtoniana. L'obiettivo principale era quello di creare un sistema solare stabile, in grado cioè di preservare le orbite nel tempo, senza che queste degenerino.

Oltre all'obiettivo principale, si sono volute implementare altre funzionalità utili all'utente, tra cui:

- finestra grafica per visualizzare i corpi, con relativa interfaccia grafica;
- possibilità di inserire nuovi corpi con una massa e direzione a scelta;
- possibilità di avviare, interrompere o ripristinare la simulazione;
- possibilità di scegliere tra diverse configurationi iniziali.

1.2 Istruzioni per la compilazione

Piattaforma di riferimento: Ubuntu 20.04.

Eseguire il pull della repository con il comando:

```
$ git clone https://github.com/NexganGH/gravity_simulator
```

Installare SFML v2.5.1 con il comando:

```
$ sudo apt-get install libsfml-dev
```

Installare TGUI, libreria estensione di SFML per create le interfacce utenti:

```
$ sudo add-apt-repository ppa:texus/tgui
$ sudo apt-get update
$ sudo apt-get install libtgui-1.0-dev
```

Assicurarsi di avere CMake installato, da installare altrimenti con:

```
$ sudo apt-get install cmake
```

Il programma può essere compilato, *per la release*, con:

```
$ cmake -S. -B build -DBUILD_TESTING=Off -DCMAKE_BUILD_TYPE=Release
$ cmake --build build
```

Questo creerà un eseguibile `gravity` nella cartella `build`, da eseguire con il comando:

```
$ build/gravity
```

1.3 Funzionalità

Il programma consiste in uno spazio in cui sono presenti diversi corpi stellari, i quali sono soggetti a gravità reciproca.

La schermata iniziale prevede una simulazione del sistema solare fino a Marte. L'utente può da qui avviare la simulazione con il tasto in alto a sinistra.

Tra le varie funzionalità:

- l'utente può inserire un nuovo corpo con click destro del mouse. Questo tasto apre una finestra che richiede di inserire la massa **in unità terrestri** (1 = massa della Terra). Successivamente è richiesto scegliere la direzione verso cui il corpo verrà lanciato con una velocità di modulo 30 km/s.
- l'utente può mettere in pausa la simulazione
- l'utente può ripristinare la simulazione con il tasto "reset", in alto a sinistra.
- l'utente può scegliere tra varie configurazioni predefinite in alto a sinistra.

Altre caratteristiche:

- la grandezza dei pianeti è proporzionale in maniera esponenziale alla massa (con esponente < 1);
- il colore dei pianeti è casuale; il colore delle stelle varia da giallo a rosso a seconda della massa.
- la finestra è sempre quadrata e dipende dalla grandezza dello schermo dell'utente.

Da notare che l'interfaccia grafica non scala con la grandezza dello schermo dell'utente.

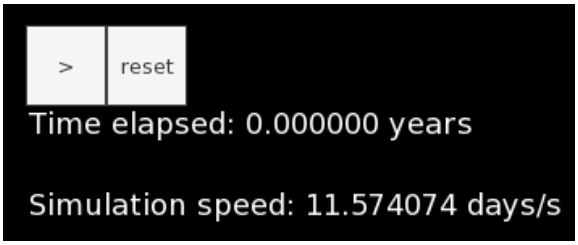

Figura 1: Il tasto (sinistra) Play/Pause permette di far partire e fermare la simulazione in ogni momento, mentre il tasto (destra) reset riporta la simulazione allo stato iniziale (i corpi già presenti ritornano nel punto in cui erano a inizio simulazione mentre quelli generati dall'utente vengono eliminati)

Figura 2: Il menù presente nella parte alta a sinistra dell'interfaccia serve a selezionare delle simulazioni prefabbricate. Cliccandone una essa comparirà nello schermo.

1.4 La fisica dietro al programma

La legge di gravitazione universale afferma che due punti materiali si attraggono con una forza di intensità direttamente proporzionale al prodotto delle masse dei singoli corpi e inversamente proporzionale al quadrato della loro distanza. Questa legge, espressa vettorialmente, diventa:

$$\mathbf{F}_{2,1}(\mathbf{r}) = -\frac{Gm_1m_2}{r^2}\hat{\mathbf{u}}_r \tag{1}$$

dove $\mathbf{F}_{2,1}$ è la forza con cui l'oggetto 1 è attratto dall'oggetto 2, G è la costante di gravitazione universale, che vale circa $6,67 \cdot 10^{-11} \frac{Nm^2}{kg^2}$, m_1 e m_2 sono le masse dei due corpi, $\mathbf{r} = \mathbf{r}_1 - \mathbf{r}_2$ è il vettore

congiungente i due corpi (supposti puntiformi), r è il suo modulo e $\hat{\mathbf{u}}_r = \frac{\mathbf{r}}{r}$ rappresenta il versore che individua la retta congiungente i due punti materiali.

Il programma utilizza questa relazione per calcolare istante per istante le forze che agiscono su ogni corpo, e di conseguenza le accelerazioni:

$$\mathbf{F}_{j,tot} = \sum_{i=1, i\neq j}^{N_{bodies}} \mathbf{F}_i(\mathbf{r}) = m\ddot{\mathbf{r}}_{j,tot} \tag{2}$$

dove $\mathbf{F}_{j,tot}$ è la forza risultante che agisce sul corpo j.

Integrando poi la precedente equazione si ottiene la legge oraria $\mathbf{r}(t)$ del corpo j. Non essendo risolvibile analiticamente si deve ricorrere a metodi di integrazione numerica.

2 Trattazione matematica

Siccome la forza gravitazionale è una grandezza vettoriale (vedi 1.4) è possibile scomporla nelle sue due componenti indipendenti[1]: nel caso del progetto si è preferito fare utilizzo delle coordinate cartesiane, così da poter ottenere, a partire dalla 2, le equazioni differenziali qui sotto riportate, le cui soluzioni descrivono rispettivamente la proiezione sull'asse x ed y del moto dei corpi.

$$F_{j,x,tot} = \sum_{i=1, i\neq j}^{N_{bodies}} -\frac{Gm_j m_i}{d_{j,i}^2}\frac{x_j - x_i}{d_{j,i}} = m_j\ddot{x_j} \tag{3}$$

$$F_{j,y,tot} = \sum_{i=1, i\neq j}^{N_{bodies}} -\frac{Gm_j m_i}{d_{j,i}^2}\frac{y_j - y_i}{d_{j,i}} = m_j\ddot{y_j} \tag{4}$$

dove $F_{j,x,tot}$ e $F_{j,y,tot}$ sono le componenti cartesiane della forza totale che agisce sul corpo j, x_j, x_i e y_j, y_i sono rispettivamente le ascisse e le ordinate dei corpi j e i, $\ddot{x_j}$ e $\ddot{y_j}$ sono le componenti cartesiane dell'accelerazione del corpo j, $d_{j,i}$ è la distanza tra i due corpi, così definita:

$$d_{j,i} = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2} \tag{5}$$

Le equazioni 3 e 4 possono essere scritte in forma vettoriale, ottenendo:

$$\mathbf{F}_{j,tot} = \sum_{i=1, i\neq j}^{N_{bodies}} -\frac{Gm_j m_i}{d_{j,i}^2}\left(\frac{x_j - x_i}{d_{j,i}}; \frac{y_j - y_i}{d_{j,i}}\right) = m_j(\ddot{x_j}; \ddot{y_j}) \tag{6}$$

dove $(\frac{x_j - x_i}{d_{j,i}}; \frac{y_j - y_i}{d_{j,i}})$ è il versore del vettore distanza tra i due corpi[2].

Nel progetto si farà uso di questa equazione in forma vettoriale, alla quale verranno appliacati i metodi di integrazione numerica che ci proponiamo di trattare di seguito.

2.1 Metodo LeapFrog

Come precedentemente menzionato, la soluzione dell'equazione differenziale 6 per ogni singolo corpo risulta impossibile da ricavare analiticamente. Si deve quindi ricorrere a metodi numerici di integrazione per equazioni differenziali. Nel presente progetto, per ottenere la posizione $\mathbf{r}_j$ e la velocità $\dot{\mathbf{r}}_j$ del corpo j([3]), che risente dell'attrazione gravitazionale simultanea degli altri N-1 corpi,è stato adottato il metodo

[1] Il progetto infatti tratta solo il caso bidimensionale.

[2] Si veda la trattazione matematica in appendice A.1

[3] Il sistema di riferimento utilizzato è quello della finestra grafica nel quale l'origine è posta nel vertice in alto a sinistra, l'asse delle ascisse positivo verso destra e quello delle ordinate positivo verso il basso.

Leapfrog. È un sistema ampiamente utilizzato per ricavare numericamente le soluzioni di equazioni differenziali scalari del tipo:

$$\ddot{x} = \dot{v} = A(x) \tag{7}$$

L'algoritmo permette di calcolare la posizione del corpo in esame allo step $i+1$ x_{i+1} nota la positione allo step i x_i e la velocità del corpo allo step $v_{i+\frac{1}{2}}$.

Le equazioni per aggionare la posizione e la velocità del corpo al rispettivo step sono:

$$v_{i+\frac{1}{2}} = v_i + \frac{1}{2} a_i \Delta t \tag{8}$$

$$x_{i+1} = x_i + v_{i+\frac{1}{2}} \Delta t \tag{9}$$

$$v_{i+1} = v_{i+\frac{1}{2}} + \frac{1}{2} a_{i+1} \Delta t \tag{10}$$

dove $a_i = A(x_i)$.

Il suddetto metodo integrativo ha la proprietà di conservare l'energia meccanica totale, risultando in una completa stabilità delle orbite dei corpi (vedi confronto in appendice con altri metodi testati per l'implementazione A.2) e per questo motivo viene specialmente utilizzato per simulare, come nel presente caso, le traiettorie coniche dei pianeti.

3 Implementazione

3.1 Strumenti utilizzati

È stato utilizzato C++ 17, compilator tramite GCC.

Si sono utilizzate le librerie:

- SFML 2.5.1 per l'implementazione grafica.
- TGUI 1.0 per l'interfaccia utente.

Per il linking dei file e la creazione degli executable, come i test, è stato usato CMake 3.16.

Per lo sviluppo è stato utilizzato l'IDE Visual Studio Code, usato in ambiente WSL con Ubuntu 20.04. È stato adottato, per la formattazione, un file .clang-format basato sullo stile di Google.

Per effettuare il controllo delle versioni, è stato utilizzato Git 2.5.1; come repository remota è stato adottato GitHub. La repository è presente al link: `https://github.com/NexganGH/gravity_simulator/tree/master`

3.2 Struttura dei file

La directory principale contiene è divisa nelle seguenti sotto-directory:

- `docs/` contiene questo file, il suo sorgente in Latex e altri file necessari alla compilazione.
- `include/` contiene gli header (`.hpp`).
- `src/` contiene i file `.cpp` con le definizioni.
- `test/` contiene i file di testing.

3.3 Struttura generale delle classi

L'obiettivo di design del progetto è stato quello di sfruttare al massimo la OOP per garantire la massima riusabilità delle classi, in egual modo aumentando la mantenibilità e scalabilità del codice.

La classe cardine del programma è `SimulationState`, la quale rappresenta lo stato corrente della simulazione. Essa contiene:

- la lista dei corpi presenti nello spazio (vettore di `Body`);
- un'istanza di `Configuration`, che rappresenta la configurazione iniziale attualmente in uso;
- un'istanza di `PhysicsEngine`, classe che si occupa di effettuare i calcoli circa le posizioni dei pianeti;
- un'istanza di `Renderer`, il cui scopo è disegnare a schermo i pianeti.

`Body` è una classe che rappresenta un corpo nello spazio, dotato di una posizione, velocità, forza e massa. Tale classe è astratta poiché alcuni metodi dipendono dal tipo del corpo, distinzione eseguita tramite ereditarietà (attualmente le classi figlie sono `Planet` e `Star`).

`Configuration` rappresenta uno specifico stato iniziale della simulazione: è quindi caratterizzato da una lista di corpi con le loro posizioni iniziali. A tal propositio, la funzione `getConfigurations` fornisce una lista di configurazioni, le quali sono mostrate all'utente in alto a sinistra nell'interfaccia.

`PhysicsEngine` è la classe che esegue i calcoli: il suo metodo pubblico più importante è `evolve`, il quale permette di evolvere una lista di corpi nell'arco di tempo `dt`. Esso implementa il metodo di integrazione LeapFrog.

`Renderer` è la classe che permette di disegnare a schermo i corpi e i componenti dell'interfaccia. Nel caso dei corpi, è effettuata un automatico una conversione dalle coordinate dei corpi, espressi in coordinate reali (metri), alle coordinate sullo schermo (pixel). Tale conversiene avviene, ovviamente, in base alla larghezza dell'universo in visione (variabile a seconda della configurazione).

`GuiManager` è una classe che crea e gestisce l'interfaccia utente.

3.4 PhysicsEngine

Il metodo principale, `evolve`, ha come parametro la lista dei corpi da evolvere e l'intervallo di tempo `dt`. Tale intervallo deve corrispondere all'intervallo di tempo che è trascorso *nella realtà* dalla scorsa chiamata di `evolve`.

Al fine di simulare eventi che avvengono in periodi estesi, come anni, è necessario però utilizzare una `timescale`, ovvero un valore che velocizzi la simulazione. Tale valore è moltiplicato per il `dt` del metodo evolve. Si incorre nella seguente relazione:

$$T_{simulazione} = n \cdot dt \cdot timescale$$

dove $T_{simulazione}$ è il tempo trascorso, in secondi, nella simulazione durante n chiamate di `evolve` di un intervallo di tempo dt ciascuno.

Se $n \cdot dt = 1s$, cioè è passato 1 secondo nella realtà, si ha che $T_{simulazione} = timescale$; da ciò si conclude che timescale, espresso in secondi, rappresenta il numero di secondi passato nella simulazione durante un secondo nella realtà.

Si noti inoltre che, siccome dt dipende dalla velocità con cui viene eseguito il ciclo, quindi dall'hardware dell'utente, il tempo della simulazione è normalizzato: *la velocità della simulazione è costante in tutti gli ambienti/dispositivi.*

Questo implica il fatto, però, che un utente con un hardware lento, a parità di timescale, avrà degli intervalli di calcolo maggiori, diminuendo l'accuratezza dei calcoli. A tal proposito può essere ridotto il timescale.

3.5 Renderer

La classe `Renderer` ha lo scopo di disegnare i corpi e l'interfaccia a schermo. Poiché le posizioni dei corpi sono espressi in coordinate reali (metri), è necessario che prima di disegnarli a schermo le loro coordinate sono convertite. A tal proposito, il metodo statico `fromUniverseWidth` permette di calcolare una scala a partire dalla grandezza dell'universo e della finestra:

$$scale = \frac{universe\ width}{window\ width}$$

Per convertire una coordinata `x` da coordinate reali a coordinate sullo schermo, si usa la relazione:

$$x_{schermo} = \frac{x_{reale}}{scale}$$

3.6 Configuration

La classe `Configuration` contiene lo stato iniziale del progetto - tra cui la lista dei corpi. Lo scopo della classe è quello di fornire gli elementi per inizializzare `SimulationState`. La lista dei corpi è fornita tramite copia, in modo che la lista all'interno della classe rimanga non modificata possa essere riutilizzata per il tasto "reset".

Sono state create diverse istanze di `Configuration` per proporre diverse configurazioni iniziali all'utente. Queste sono istanziate nel metodo `getConfigurations`.

3.7 Altre classi

Tra le altre classi del programma ci sono:

- `GuiManager`, utilizzata per la creazione e gestione dell'interfaccia.
- `Vector`, utilizzata per rappresentare un vettore nello spazio 2D. Sono disponibili diverse operazioni.
- `OrbitDrawer`, utilizzata per disegnare le orbite e cancellare i punti dopo un intervallo di tempo.

3.8 Ciclo principale

Il ciclo principale si svolge nel `main`. Un ciclo permette di utilizzare il metodo `evolve` di PhysicsEngine per evolvere la simulazione per un tempo indefinito. Il valore `dt`, da fornire a `PhysicsEngine` (si veda Sezione (3.4)), è misurato tramite un cronometro.

3.9 Utilizzo dei puntatori

Al fine di evitare memory leaks, sono stati utilizzati gli smart pointer. Ove possibile, è stato adottato lo `unique_ptr`.

A titolo esemplificativo si consideri la lista di puntatori a `Body`:

```
std::vector<std::unique_ptr<Body>> _bodies;
```

Il vettore contiene dei puntatori a `Body` poiché questa è una classe astratta, si tratta quindi di polimorfismo dinamico.

`Configuration` ne crea una copia che viene passata a `SimulationState`. Per la copia si utilizza il metodo astratto di `clone` che ritorna uno `unique_ptr` all'oggetto `Body` clonato: a questo punto viene

creato un nuovo vettore con questi puntatori. L'ownership è poi passata a `SimulationState`; tale principio è applicato anche agli altri puntatori all'interno di `SimulationState`, la quale ne detiene l'ownership.

Per accedere ad un oggetto detenuto da `SimulationState`, questa fornisce come interfaccia una serie di metodi che ritornano referenze agli `unique_ptr`.

Per istanziare i puntatori si utilizzano i metodi `std::make_unique` e `std::make_shared`.

4 Testing e debugging

4.1 Tecnica di testing

È stato utilizzata la libreria `doctest` per effettuare i testing. CMake permette di effettuare i linking e creare un eseguibile per ogni file di test.

4.2 Eseguire un test

I test sono contenuti in executable separati con il nome della classe.t. Ad esempio, per effettuare il test della classe `PhysicsEngine`:

```
$ cmake -S. -B build -DBUILD_TESTING=On
$ cmake --build build --target physics_engine.t
```

Per eseguire i test, eseguire `physics_engine.t` nella cartella `build`:

```
$ build/physics_engine.t
```

4.3 Memory Leak

Durante i test non sono stati riscontrati memory leak provenienti dal codice originale. In determinati test si sono evidenziati, però, dei leak provenienti da SFML oppure TGUI la cui causa è sconosciuta.

5 Risultati e problemi riscontrati

Tramite il metodo Leapfrog, si è ottenuta con successo una simulazione in scala del sistema solare (fino a Marte oppure fino Nettuno a seconda della configurazione). Si riscontra, con successo:

- che le orbite sono stabili e non degenerano nel tempo: ricoprono sempre lo stesso percorso;
- che la Terra rivoluziona attorno al Sole in 1 anno, come atteso;
- che le funzionalità implementate, come la creazione di un nuovo pianeta, il reset, la pausa funzionano correttamente;
- che la simulazione risulta anche in altri casi coerente con ciò che approssimativamente ci si aspetta.

Tra i problemi riscontrati, tuttavia, si evidenzia:

- un peggioramento delle performance nel tempo, dovuto alla creazione dei punti delle orbite (anche se questi vengono cancellati dopo un certo periodo);
- dopo un periodo di simulazione elevato, il tempo di rivoluzione della Terra aumenta la proprio durata (> 1 anno). Si ipotizza che tale problema sia dovuto alle approssimazioni numeriche effettuate.

A Calcoli e approssimazioni

A.1 Versore distanza tra due corpi

Dati due corpi p_1 e p_2, si definisce il vettore distanza tra di essi come $\mathbf{d}$:

$$\mathbf{d} = (x_1 - x_2; y_1 - y_2) \tag{11}$$

Per calcolare il suo versore è sufficiente riscalare lo stesso per l'inverso della sua norma:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{12}$$

Di conseguenza la formula del versore distanza sarà:

$$\hat{\mathbf{d}} = (\frac{x_1 - x_2}{\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}};\ \frac{y_1 - y_2}{\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}}) \tag{13}$$

Ecco perchè nel testo ci si riferisce alle sue componenti nella forma:

$$\hat{\mathbf{d}}_x = \frac{x_1 - x_2}{\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}};\ \hat{\mathbf{d}}_y = \frac{y_1 - y_2}{\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}} \tag{14}$$

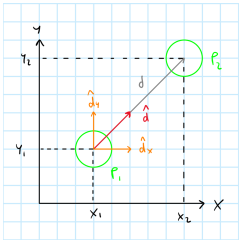

Figura 3: Scomposizione in coordinate del versore distanza.

A.2 Altri metodi di approssimazione

Tra i vari metodi di integrazione numerica testati, il metodo LeapFrog si è rivelato il più accurato, mostrando una stabilità delle orbite notevolmente superiore a quella ottenuta con gli altri metodi.

La prima versione del progetto presentava un'implementazione del metodo di Eulero, basato sullo sviluppo in serie di Taylor, fino a terzo e quarto ordine. Le orbite erano instabili e divergevano.

L'implementazione del metodo Runge-Kutta del secondo e quarto ordine risultava più precisa nella stablità delle orbite, presentando comunque approssimazioni notevoli sull'energia, che non si conservava e continuava a crescere.

Il metodo LeapFrog, a dispetto degli altri, mantiene costanti le energie, approssimando leggermente le traiettorie (a livello trascurabile date distanze così elevate tra i corpi celesti), mantenendo però orbite costanti e stabili nel tempo.