BiliardoTriangolare

Pedrielli Chiara
Peroni Chiara
Zanella Giovanni

25 Agosto 2025

1 Introduzione

"Biliardo Triangolare" è un programma che presenta due funzionalità: la prima permette all'utente di simulare il lancio di una singola pallina all'interno di un biliardo costituito da due bordi inclinati, la seconda permette di simulare un numero programmabile N di lanci generando i dati di input da delle distribuzioni scelte dall'utente e restituendo i parametri delle distribuzioni dei valori output. Per la realizzazione del progetto in gruppo è stato utilizzando Git Hub, il lavoro è stato svolto su una repository condivisa.

2 Scelte progettuali e implementative

Il progetto è stato suddiviso in sei file:

- main.cpp: contiene il flusso logico di esecuzione del programma
- biliardo_test: continene i test scritti con la libreria DoctTest.
- biliardo.hpp: header file contenente le definizioni delle struct (Point, CollisionResult, Result) e class (Ball, Border) utilizzate, con relativa dichiarazione di metodi.
- biliardo.cpp: sono stati implementati i vari metodi delle classi e delle struct necessari per le funzionalità del programma.
- biliardo_statistica.hpp:header file nel quale è presente la struct specifica per svolgere la funzionalità statistica del programma (Statsresult).
- biliardo_statistica.cpp: è contenuta l'implementazione del metodo simulate_stats.

Il codice è stato inserito in un namespace, nominato pf, per evitare conflitti tra i nomi del programma e quelli delle librerie esterne. Seguendo ora l'organizzazione del programma in più file, verranno decritte più nel dettaglio le classi, le struct e le implementazioni dei relativi metodi.

2.1 Header file biliardo.hpp

In primo luogo, è stato adottato il metodo delle include guards allo scopo di prevenire eventuali problemi legati a inclusioni multiple e violazioni della "One Definition Rule".
All'interno del namespace denominato "pf" è presente la definizione di struct e class, e la relativa dichiarazione dei metodi. La struct Point consente di rappresentare un punto nel piano come un'unica entità: le sue variabili membro corrispondono infatti alle coordinate cartesiane del punto (ascissa e ordinata).
E' poi definita la classe Ball, che racchiude le specifiche della particella nel moto, ovvero le sue coordinate, espresse con un Point, e il suo angolo in radianti, calcolato rispetto alla direzione positiva dell'asse x. Nella parte pubblica della classe sono presenti il costruttore, i metodi getter, dichiarati const per consentire l'accesso in sola lettura alle variabili membro private, e i metodi move_to e set_angle che permettono di modificare la posizione o l'angolo della particella. Nel metodo move_to l'oggetto Point è passato by const reference mentre nella funzione successiva, per new_s, essendo un oggetto di un tipo primitivo come double viene applicato il

passato by value. Questo ragionamento è applicato anche ai metodi successivi, permettendo di distinguere in modo coerente quando passare i parametri by const reference, by reference oppure by value.
La struct CollisionResult è designata per contenere il risultato di una singola collisione. Il booleano has_hit risulta vero se è avvenuto un urto con uno dei due bordi, le coordinate dell'urto sono salvate nella variabile hit, mentre al booleano upper viene assegnato vero se l'urto è avvenuto con il bordo superiore.
Successivamente è definita la classe Border, le cui variabili membro sono i parametri caratteristici dei bordi del biliardo: rispettivamente l'ordinata dell'estremo sinistro, la cui ascissa è zero, l'ordinata e l'ascissa dell'estremo destro e la pendenza del bordo superiore. Nella sezione pubblica è presente il costruttore, in cui si impedisce di inizializzare la slope nel caso in cui sia stato inserito L=0, seguito dai metodi getter, dichiarati come const, e dal metodo move_border per modificarne le caratteristiche. La funzione membro initial_check prende in input i parametri by const reference e si occupa di verificare che i valori inseriti per i bordi e la particella siano validi e conformi alla geometria del problema.
Considerando che questa funzione prende in ingresso e opera su oggetti diversi svolgendo diversi controlli non aveva senso mantenerla dinamica e chiamarla su un'istanza in particolare, è stato dunque deciso di renderla static e di accedere ai parametri dei vari oggetti tramite i metodi getter.

Il metodo NewAngle, anch'esso static e i cui parametri sono passati by const reference, provvede a calcolare il nuovo coefficiente angolare della retta che descrive la traiettoria della particella dopo un urto. Infine i vari metodi set permettono di assegnare nuovi valori ai parametri del biliardo.
La struct Result rappresenta il formato in cui viene salvato il risultato della simulazione nella versione grafica. Salva al suo interno il numero di rimbalzi avvenuti, le caratteristiche della particella al momento dell'uscita quando x=L mentre l'std::vector trajectory è utile ai fini della rappresentazione grafica. Il metodo static BallSimulation è la funzione attraverso la quale viene simulato il movimento completo della particella fino ad ottnerne il risultato finale.

2.2 Source file biliardo.cpp

All'interno del file biliardo.cpp sono riportate le definizioni dei metodi dichiarati nell'header file, responsabili della simulazione del movimento di una particella.
Sono stati definiti i metodi getter per le variabili membro della Classe Ball. Successivamente, il metodo move_to permette di assegnare nuovi valori alle coordinate di un oggetto di tipo Ball, mentre il metodo set_angle consente di modifcarne l'angolo.

Metodo next_collision
E' poi definito il metodo next_collision, il cui scopo è quello di individuare le caratteristiche dell'urto successivo della particella. In particolare, ne determina le coordinate e il bordo con cui avviene, oppure lancia un eccezione nel caso in cui, per la geometria del sistema, la particella tornerebbe indietro. E' stato inserito un assert per accertarsi che, al momento dell'inizio del metodo, l'ascissa della particella sia in condizioni valide. In primo luogo, prima di calcolare l'ascissa dell'urto, vengono considerati due casi particolari, in cui la traiettoria della palla è parallela a uno dei due bordi. Questo comporterebbe l'insorgenza di un denominatore uguale a zero nel calcolo dell'intersezione con il bordo superiore o inferiore. Ad esempio, nel caso in cui la traiettoria della palla sia parallela al bordo superiore, se si calcolasse un'ipotetica intersezione con esso, il denominatore sarebbe uguale a zero. Dunque, nel blocco if si dà per scontato che non ci possa essere un'intersezione con questo bordo e la si calcola solo per il bordo inferiore. Se l'intersezione tra le due rette avviene ad un ascissa inferiore di L, si restituisce un oggetto della struct CollisionResult in cui viene assegnato valore "true" al primo booleano, le coordinate dell'urto all'oggetto di tipo Ball, e "false" all'ultimo in quando ad essere colpito è il bordo inferiore. Se, invece, l'intersezione avverrebbe solo oltre la fine del bordo, la particella esce, assegnando "false" al booleano has_hit della struct e calcolando l'ordinata della retta della particella in corrispondenza di x=L. Analogamente viene svolto il caso successivo in cui ad essere parallelo alla traiettoria della palla è il bordo inferiore.
Dopo la gestione del caso in cui i bordi sono verticali, si passa al calcolo di x_up, come intersezione tra la retta della particella in moto e quella a cui appartiene il bordo superiore.
Se x_up risulta (strettamente) maggiore dell'ascissa corrente della particella ma minore di L significa che l'urto è avvenuto col il bordo superiore. Viene quindi calcolata l'ordinata dell'urto e costruito il corrispondente oggetto CollisionResult.
Se x_up è maggiore di L (o analogamente x_down nel caso subito successivo) denota che la particella è uscita dal biliardo e se ne calcolano le caratteristiche quando x=L.
Se x_up risulta invece minore o uguale dell'ascissa corrente del punto allora l'urto può essere avvenuto solo con il bordo inferiore. Viene calcolata x_down, sempre come intersezione di rette, e valutato se la particella esce o collide effettivamente col bordo. In ognuno dei due casi, viene creato l'oggetto di tipo CollisionResult per conservare quanto appurato.

Metodo NewAngle

Il metodo in questione calcola la nuova pendenza della retta relativa alla traiettoria della particella dopo ogni urto. Il coefficiente angolare è stato calcolato in funzione della pendenza del bordo con cui urta (mb) e della retta della particella prima dell'urto (mp), tenendo conto del fatto che l'urto è perfettamente elastico e l'angolo di incidenza è uguale a quello di riflessione.
Per il caso in cui il denominatore risulta molto prossimo allo zero è stato predisposto il lancio di un'eccezione.
Sono presenti vari blocchi if per racchiudere e gestire le diverse possibili configurazioni in cui si può presentare il sistema. Per prima cosa viene gestito il caso dei bordi orizzontali.
In seguito, per calcolare la nuova pendenza della traiettoria della palla è stata utilizzata la formula corrispondente al rapporto tra le variabili numerator e denominator, per assicurarsi che il segno assegnato alla pendenza fosse quello giusto sono stati distinti i seguenti casi:

- Bordi inclinati verso l'asse x.
  Per questo caso vi sono due possibilità: la palla ha pendenza precedente all'urto uguale o maggiore della normale al bordo e dunque dopo il rimbalzo torna indietro, oppure ha pendenza minore e dopo il rimbalzo va avanti

    Bordi inclinati verso l'esterno.
    Se la palla durante l'urto ha angolo di riflessione pari all'angolo tra la normale e il piano orizzontale, allora la pallina dopo l'urto prosegue dritta, ovvero con pendenza pari a zero. Questo rappresneta il caso spartiacque per l'assegnazione del segno della slope: il programma calcola l'angolo di riflessione della pallina e lo confronta con l'angolo della normale, a questo punto decide il segno adeguato per la slope finale della palla.

Dato che tutti i ritorni si trovano all'interno degli if, i quali rappresentano e gestiscono i vari casi illustrati qui sopra, per evitare il rischio di raggiungere il termine della funzione senza un risultato, è stato aggiunto un return della nuova pendenza della palla utilizzando la formula generica senza forzare il segno.
Al termine della funzione viene restituito il valore della nuova slope calcolato.

Metodo initial check

Il metodo initial check consiste in una serie di controlli per accertarsi che i valori inseriti dall'utente nella fase di input siano validi e conformi alla geometria del sistema. Viene testato, nell'ordine, che la particella parta tra i bordi e non fuori da essi, che i bordi non si intersechino con l'estremo di destra sull'asse x, che i bordi verticali non ostruiscano la palla, che l'angolo di lancio sia compreso da -90°e 90°, ed infine che l'ascissa dell'estremo destro del bordo sia positiva.

Metodo BallSimulation

Nel metodo BallSimulation convergono i metodi precedentemente descritti. La variabile bounce quantifica il numero di rimbalzi compiuti prima di uscire, il vettore trajectory tiene conto di diverse posizioni della palla durante tutta la simulazione, queste posizioni saranno poi usate per mostrare nella finestra grafica i vari rimbalzi.
Nel ciclo for si stabilisce un numero massimo di un milione di rimbalzi, al raggiungimento del quale il loop termina. Il metodo prende in input un oggetto di tipo Ball e i due bordi su cui applica il metodi next collision.
A questo punto, se l'urto non è avvenuto, la palla viene spostata nelle coordinate finali; se l'urto è avvenuto la palla viene spostata nelle coordinate dell'impatto.
In entrabi i casi i cicli while servono a far sì che la palla grafica si muova lentamente e in maniera fluida: il suo percorso viene infatti diviso in tanti piccoli passi che vengono inseriti nel vettore trajectory e che saranno poi maneggiati nel main.
Una voltra determinate le coordinate successive della palla tramite next collision viene chiamato il metodo NewAngle per calcolare la nuova pendenza della traiettoria della pallina.

2.3 Header file biliardo_statistica.hpp

Il file header biliardo_statistica.hpp, anch'esso dotato di include guards, definisce la struct StatsResult le cui variabili membro sono i valori che verranno restituiti al termine della simulazione statistica. La funzione simulate_stast, qui solo dichiarata, prende in input i valori forniti dall'utente per la simulazione quali il numero di prove, i parametri caratteristici delle distribuzioni gaussiane da cui estrarre i dati e le caratteristiche del bordo.

2.4 Source file biliardo_statistica.cpp

All'interno del source file dedicato alla simulazione statistica è presente la definizione della funzione simulate_stats. Essa inizializza un generatore di numeri pseudocasuali (engine) che lavora con il seed fornito da rd(). Vengono cosi definite due distribuzioni gaussiane per valori double relative all'ordinata iniziale e all'angolo di lancio, i cui parametri media e deviazione standard sono scelti dall'utente. All'interno del ciclo for, iterato tante volte quanto è il numero di simulazioni da svolgere, viene applicata la funzione BallSimulation sui valori prodotti dalla generazioni di numeri casuali. Questo blocco di codice è stato inserito in un blocco try-catch per evitare che il programma venga interrotto da una pallina che torna indietro, come invece accade per i lanci singoli: nella simulazione statistica le palline che tornano indietro vengono semplicemente ignorate. Viene quindi salvato il risultato su un vettore dinamico mediante il push back.
Al termine della simulazione, viene applicata sui due vettori creati la lambda function parameters. Essa contiene le specifiche per il calcolo, lavorando su un range di oggetti tramite gli iteratori, di media, deviazione standard, coefficiente di simmetria e coefficiente di appiattimento. I risultati sono salvati in un oggetto della struct StatsResult e vengono poi restituiti.

2.5 main.cpp

Come in parte già discusso nella sezione input/output, nel main sono contenuti il flusso logico del programma e i comandi per l'interazione con l'utente da terminale. Il programma inizia chiedendo all'utente se desidera svolgere un lancio singolo o se vuole eseguire la simulazione statistica di N lanci.
Se l'utente sceglie di effettuare una simulazione statistica, viene chiesto di inserire i valori necessari, controllando subito eventuali inserimenti non validi che possono comportare il lancio di un'eccezione. Vengono costruiti i bordi per simmetria e poi creato un oggetto res di tipo StatsResult in cui vengono salvati i risultati ottenuti dalla funzione simulate_result. Infine i valori vengono stampati a schermo, ciascuno accompagnato da una breve stringa esplicativa.

Nel secondo caso, per la simulazione del moto di una singola particella, è stata predisposta l'interfaccia grafica utilizzando la versione 2.6.0 di SFML:
per prima cosa vengono dichiarati e inizializzati gli oggetti necessari per aprire la finestra grafica, aggiornarla e per stampare il testo di input e output.
Successivamente vengono inizializzati gli oggetti grafici (ball, broders, middle line) e gli oggetti logici.// A questo punto vengono preparati gli oggetti Text per stamapre sulla finestra grafica le domande del computer (questions), le risposte dell'utente (inputText) e la risposta finale del programma (response).
Per fare in modo che le domande venissero stampate a schermo una alla volta solo dopo che l'utente avesse inserito al risposta alla domanda precedente, è stato creato un vettore contenente le domande ordinate: la prima viene stampata all'apertura della finestra grafica, mentre la stampa delle domande successive è gestita, in seguito, da un ciclo for.
Subito dopo la creazione del vettore è stata implementata la lamba function updateQuestion che ci permetterà di aggiornare la domanda stamapta a schermo.
A questo punto si apre la finestra grafica: il primo if serve per aggiornare la finestra grafica: passati 0.016s quest'ultima viene ripulita e ogni oggetto viene ridisegnato.
In seguito tramite un ciclo while vengono gestiti gli eventi:

- Closed window: alla chiusura della finestra grafica il programma viene terminato.
- Backspace: si trova all'interno della condizione "Text entered", viene cancellata l'ultimo carattere inserito dall'utente.
- Invio: si trova all'interno della condizione "Text entered", la stringa digitata dall'utente viene salvata nella variabile apposita, quest'ultima viene determinata dal programma in base al valore della variabile step.
  Il resto dei comandi presenti all'interno dell'if che gestisce l'inserimento del testo, servono per la scrittura in tempo reale della risposta dell'utente.
- Spazio: a questo punto inizia la simulazione del lancio, per prima cosa si controlla che i valori delle variabili date in input siano accettabili per una simulazione, se questo controllo va a buon fine inizia la simulazione vera e propria tramite il metodo BallSimulation.
  Il ciclo for presente di seguito fa in modo che la palla grafica venga spostata nel punto dell'intersezione.
  Al termine della simulazione viene stampata a schermo la risposta finale del programma.

Infine, dopo la chiusura del blocco try, il main termina con un blocco catch adibito alla gestione di eventuali eccezioni lanciate nel corso del programma. I messaggi di errore vengono stampati a schermo dallo stream di

output cerr, mentre il metodo what() permette di visualizzare la spiegazione dell'errore specifica per il caso che si è verificato.

3 Librerie esterne da installare

Per quanto riguarda librerie o file esterni da installare, il file doctest.h, in quanto file headeronly, e quello relativo a clangformat sono contenuti nella cartella. In merito alla componente grafica del progetto, invece, risulta necessario installare SFML per poter visualizzare correttamente l'interfaccia. Per la realizzazione del progetto è stata adottata la versione 2.6.0 di SFML.
Se è sufficiente utilizzare una versione compatibile tipo 2.6.x e si possiede una distribuzione Ubuntu 22.04 o più recente, è possibile installare SFML e i pacchetti necessari con il comando: "sudo apt install -y build-essential

cmake pkg-config libsfml-dev fonts-dejavu-core".
In alternativa, se risulta necessaria esattamente la versione 2.6.0 si può procedere digitando da terminale:

"sudo apt install -y build-essential cmake pkg-config git
libfreetype-dev libx11-dev libxrandr-dev libxcursor-dev libxinerama-dev
libxcomposite-dev libxfixes-dev libglvnd-dev libopenal-dev
libflac-dev libogg-dev libvorbis-dev libudev-dev"
seguito da:

"git clone –branch 2.6.0 https //github.com/SFML/SFML.git
cd SFML
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release -
DSFML_INSTALL_PKGCONFIG_FILES=TRUE
cmake –build build –parallel
sudo cmake –install build
sudo ldconfig".

4 Come eseguire il programma

Si riportano di seguito i comandi per eseguire:
Compilazione del programma:

```
g++ -std=c++17 -Wall -Wextra biliardo.cpp main.cpp biliardo_statistica.cpp \
    -I/home/chiara-pedrielli24/progetto/SFML-2.6.0/include \
    -L/home/chiara-pedrielli24/progetto/SFML-2.6.0/build/lib \
    -lsfml-graphics -lsfml-window -lsfml-system -lsfml-audio -lsfml-network \
    -o biliardo
```

Esecuzione del programma:

```
./biliardo
```

Compilazione dei test:

```
g++ -std=c++17 -Wall -Wextra biliardo.cpp biliardo_test.cpp \
    -I/home/chiara-pedrielli24/progetto/SFML-2.6.0/include \
    -L/home/chiara-pedrielli24/progetto/SFML-2.6.0/build/lib \
    -lsfml-graphics -lsfml-window -lsfml-system -lsfml-audio -lsfml-network \
    -o biliardo
```

Esecuzione dei test:

```
./biliardo
```

5 Input e output

Per quanto riguarda la gestione del flusso di input e output del programma tramite il terminale, essa è quasi interamente implementata nel file main.cpp. A tal proposito è stata inclusa la libreria standard iostream.
In primo luogo, nel main il programma richiede all'utente di scegliere, mediante l'inserimento di una stringa corrispondente a una delle due opzioni proposte, se eseguire lo studio statistico di N particelle, la cui ordinata e angolo iniziali vengono estratti da una popolazione gaussiana di cui l'utente sceglie la media e la deviazione standard, oppure la simulazione grafica del movimento di una singola particella.

Nel primo caso, se l'utente sceglie di eseguire la simulazione statistica, viene chiesto, tramite messaggi stampati a schermo, di inserire le caratteristiche delle distribuzioni gaussiane da cui estrarre i dati. In particolare, sono richiesti: il numero di simulazioni da eseguire, mediante l'inserimento di un numero intero, il valore medio e la deviazione standard relativa alla distribuzione normale dell'ordinata di partenza y_0, e media e deviazione standard per la popolazione gaussiana della variabile θ, angolo iniziale di lancio.
Infine, viene chiesto di inserire i parametri relativi ai bordi del biliardo nell'ordine r1, r2 ed L. r1 costituisce l'ordinata dell'estremo sinistro del bordo, mentre L e r2 rispettivamente l'ascissa e l'ordinata dell'estremo destro. I due bordi vengono poi costruiti per simmetria rispetto all'asse x.
I risultati di ciascuna simulazione, eseguita sulla base valori forniti dall'utente, vengono salvati su due vettori, uno relativo alle ordinate di uscita e uno agli angoli. Dopo di ché, su di essi sono calcolati i principali parametri statistici, quali media, deviazione standard, coefficiente di simmetria e di appiattimento. Tali valori vengono memorizzati in oggetti della struct Stats e stampati a schermo al termine del processo. Il flusso di output restituisce, infatti, il valore di ciascuno di questi parametri, accompagnato da una stringa che ne esplicita il significato.

Nel secondo caso, vengono chieste all'utente le caratterstiche iniziali della particella, ovvero l'ordinata e l'angolo di partenza espresso in radianti, e i parametri dei bordi.
Vengono stampate a schermo, una alla volta, le domande presenti nel vettore questionText: ordinata della pallina, angolo di lancio, coordinate degli estremi dei bordi.
Il programma rileva quando l'utente preme invio dopo aver inserito la risposta alla domanda stampata sullo schermo e salva questa risposta nella variabile apposita: tramite un ciclo for viene infatti aggiornata la variabile step, la quale permette al programma di sapere a quale domanda sta rispondendo l'utente e in quale variabili salvare gli input.
Dopo aver applicato la funzione BallSimulation sui parametri inseriti in input, il risultato della simulazione viene salvato in un oggetto della struct Result, le cui variabili membro sono: un intero che tiene conto del numero di rimbalzi, un oggetto di tipo Ball in cui vengono salvate le coordinate finali della particella e l'angolo di uscita, e un std :vector necesario al fine di rappresentare graficamente la traiettoria. Al termine della simulazione, dunque, vengono stampate a schermo le caratteristiche dell'oggetto Result ottenuto. Inoltre nella finestra grafica è possibile osservare il movimento della particella fino alla sua uscita.
Al termine del blocco try presente nel main, è previsto un blocco catch destinato alla gestione dell'output degli errori tramite std::cerr. In particolare, nei vari file del programma sono state individuate alcune situazioni che, qualora si verificassero, comportano il lancio di un'eccezione mediante throw. Il blocco catch del main permette quindi di intercettare tali eccezioni e visualizzare a schermo un messaggio descrittivo. In aggiunta, è presente una riga di output nel file biliardo.cpp, destinata a segnalare il raggiungimento del numero massimo di rimbalzi.

6 Strategie di Test

Per il testing è stato utilizzato il framework C++ doctest, al fine di verificare che le funzioni si comportino come previsto nei casi considerati. Il file dedicato al testing è biliardo_test.cpp e in esso è stato incluso il file doctest.h contenuto nella repository del progetto.
La strategia seguita è stata quella di verificare il corretto lancio delle eccezioni e di accertarsi del corretto funzionamento alcuni casi "corretti".
I test sono stati suddivisi in due TEST_CASE distinti, rispettivamente dedicati al controllo del corretto funzionamento delle funzioni BallSimulation e initial_checks.
Ciascuno dei due test case si articola in più subcase separati per testare il programma fornendo molteplici diverse configurazioni iniziali.
Nel primo TEST_CASE, relativo all'analisi della funzione BallSimulation sono contenuti vari subcase volti a verificare che la funzione restituisca quanto previsto. In particolare, nei casi in cui si prevede che la palla giunga fino al termine del biliardo viene controllato il numero di rimbalzi compiuti, l'ordinata della palla in corrispondenza di x=L e/o l'angolo di uscita. Per verificare che l'ordinata della palla e l'angolo uscente siano quelli attesi sono stati utilizzati check con doctest::Approx, in modo da permettere una tolleranza nel confronto tra numeri double. Nelle configurazioni in cui è previsto che la palla, a seguito di uno o più rimbalzi, torni indietro, viene

invece testato il lancio di un'eccezione, come stabilito nella consegna del progetto.
L'altro TEST_CASE è dedicato al controllo del corretto funzionamento della funzione initial_checks. In particolare, in ciascuno dei subcase realizzati viene testato il lancio di un'eccezione, dovuta ad un inserimento dei parametri iniziali non valido.

7 Interpretazione risultati ottenuti

In questa sezione abbiamo analizzato l'effetto della lunghezza L del biliardo sul comportamento statistico delle traiettorie in uscita nelle simulazioni, mantenendo costanti le ascisse degli estremi dei bordi superiore e inferiore, rispettivamente r1 = 100 e r2 = 50. Sono state realizzate 15 simulazioni totali, ciascuna da 1000 lanci, variando la lunghezza da 750 a 50 con decrementi costanti di 50 ad ogni simulazione. Le condizioni iniziali delle distribuzioni normali dell'ascissa iniziale della pallina e dell'angolo iniziale di lancio sono rimaste invariate per tutte le prove:

- Ascissa iniziale y_0 = 0, con deviazione standard σ_y0 = 25
- Angolo iniziale di lancio θ_0 = 0, con deviazione standard σ_ θ = 0.3

Il numero di simulazioni valide è rimasto elevato e costante per tutti i valori della lunghezza, affermandosi intorno al 90%.
DISTRIBUZIONE DELL'ASCISSA FINALE y_f
La media di y_f non mostra un andamento proporzionale alla lunghezza del biliardo. Oscilla tra valori positivi e negativi nell'intorno di zero, non mostrando una correlazione significativa con la lunghezza.
La deviazione standard di y_f resta costante, con valori di circa 27-29, con una leggera riduzione per valori di L più piccoli (per l = 50 abbiamo σ = 24.3).
Il coefficiente di simmetria resta vicino allo zero per tutte le prove, indicando distribuzioni finali simmetriche.
Il coefficiente di appiattimento di y_f oscilla intorno a 2.5, indicando che le code sono meno popolate rispetto a quelle della distribuzione normale. In sintesi, la distribuzione di y_f non varia significativamente con il variare di L.
DISTRIBUZIONE DELL'ASCISSA FINALE $\mathbf{y}_\theta$
La media di θ_f rimane prossima allo zero per tutti i valori di L.
La deviazione standard invece evidenzia una tendenza decrescente al diminuire della lunghezza del biliardo, infatti per L = 750 abbiamo σ = 0.56 mentre per L = 50 abbiamo σ = 0.36. Quindi per biliardi più corti abbiamo un angolo d'uscita generalmente più ristretto.
Il coefficiente di simmetria di θ_f è prossimo allo zero per L grandi, mentre notiamo una decrescita per L piccoli (per L = 50 il coefficiente è pari a -1.28). Quindi per biliardi corti abbiamo una forte asimmetria nella distribuzione degli angoli d'uscita.
Il coefficiente di appiattimento di θ_f mostra una crescita progressiva al diminuire di L (per L = 750 il coefficiente è pari a 2.75, per L = 50 abbiamo il coefficiente uguale a 5.89).
Dunque le distribuzioni di θ_f diventano più appuntite e concentrate per biliardi corti.

In conclusione la variazione della lunghezza L del biliardo influisce prevalentemente sulle distribuzioni dell'angolo di uscita θ_f, mentre ha un impatto ridotto sulle distribuzioni dell'ascissa finale y_f. In particolare per L grandi otteniamo distribuzioni più simmetriche e piatte, mentre per L piccoli riduciamo la deviazione standard aumentando l'asimmetria e l'appuntimento della distribuzione.

Nella cartella consegnata per il progetto, è incluso un documento con il foglio di calcolo contenente i risultati delle simulazioni.

8 Intelligenza artificiale

Per lo sviluppo del progetto è stato deciso di disabilitare preventivamente l'estensione GitHub Copilot in Visual Studio Code fin dall'inizio, così da evitare la generazione automatica di codice.
La quasi totalità del codice è stata scritta integralmente dal gruppo, senza ricorso a strumenti di intelligenza artificiale per la produzione diretta del codice. Infatti non sono presenti nel progetto porzioni consistenti di codice copiate o generate automaticamente. Gli strumenti di AI (in particolare ChatGPT) sono stati utilizzati esclusivamente come supporto per ricevere suggerimenti puntuali e chiarimenti, o per la risoluzione di alcuni errori minori durante lo sviluppo.
Un utilizzo più rilevante dell'IA è stato necessario nella parte relativa all'utilizzo della libreria SFML e alla gestione della grafica. In questa fase, l'intelligenza artificiale è stata sfruttata come strumento di approfondimento teorico e di chiarimenti di alcuni concetti, ma la scrittura effettiva del codice è stata svolta sempre dai componenti del gruppo.