Biliardo 2D

Simone Cervellera e Francesco Colombo

Aprile 2025

1 Scelte progettuali e implementative

1.1 Introduzione

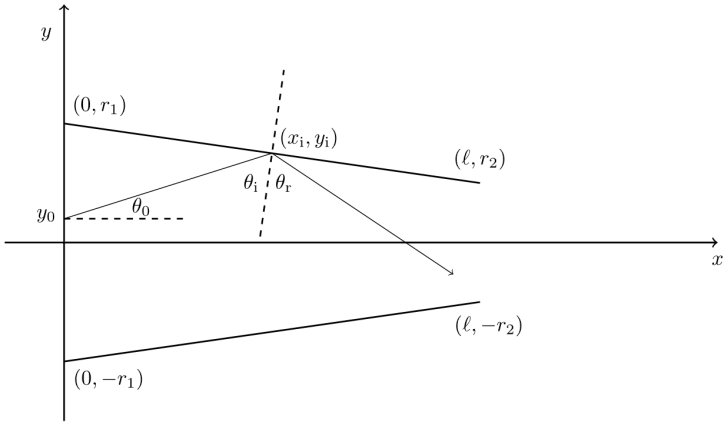

Figure 1: Sistema fisico sul quale si basa il programma.

L'obiettivo del programma è simulare un biliardo bidimensionale nel quale avvengono unicamente urti di tipo elastico, per poi calcolare, in base alle condizioni iniziali, l'esito del lancio; inoltre vogliamo simulare una quantità imprecisata di lanci con condizioni iniziali gaussianamente distribuite per mostrare all'utente la distribuzione della posizione e dell'angolo finali.
L'idea alla base della creazione del nostro biliardo 2D è di lavorare con le rette ed i loro coefficienti angolari.

$$m = \frac{m_i m_T^2 - m_i + 2|m_T|}{1 - m_T^2 + 2m_i|m_T|} \tag{1}$$

La formula 1.1, che abbiamo utilizzato per calcolare il coefficiente angolare della retta che rappresenta la traiettoria del rimbalzo m verrà ricavata esplicitamente in appendice. Al momento al lettore sarà sufficiente sapere che m_T indica il coefficiente angolare della retta che rappresenta la sponda superiore mentre m_i si riferisce a quello della traiettoria della pallina prima dell'urto. Al fine di poter fare ciò, abbiamo utilizzato cinque custom-type; una classe, Setup, composta da cinque float che rappresentano i parametri in ingresso del programma, ossia l, r_1, r_2, y_0 e θ_0 (come in fig.1) e le seguenti struct:

- **Point**, definita da due double che indicano ascisse e ordinate;

- **Line**, composta da due double che rappresentano il coefficiente angolare e l'intercetta di una retta;
- **System**, che è la rappresentazione del setup ma con tre elementi di tipo Line, ossia le due sponde ed il lancio iniziale;
- **Angle_and_point**, struct necessaria per poter avere in output i dati finali, composta da un double corrispondente ad un angolo e un Point.

1.2 Calcolo della traiettoria

La logica dietro al programma è quella di individuare dove si svolge il primo urto con le sponde, se avviene, per poi sequenzialmente calcolare i seguenti in funzione del precedente. In altre parole, tramite una serie di condizioni, il programma riconosce che se la pallina urta la sponda superiore, il prossimo urto deve avvenire con quella inferiore (salvo nel caso specifico in cui non ci siano urti successivi) e viceversa. Cerchiamo di approfondirne il funzionamento.
Ci serviamo di diverse funzioni:

- **findInterception**, la quale riceve come input due linee e resistuisce in output la loro intersezione sotto forma di un Point;
- **calculateFirstHit**, la quale, ricevendo in input un System e un double (l) restituisce dove avviene il primo urto; in caso esso non avvenga trova l'intersezione tra la traiettoria del lancio e la retta $x = l$;
- **getFinalPoint** la quale, riceve in input due Point, un double (l) e un System (assieme ad altri oggetti necessari per la resa grafica di cui parleremo più avanti). La funzione consiste fondamentalmente in un ciclo while all'interno del quale sono trattati, tramite vari if, tutti i possibili risultati di un lancio. Il ciclo fondamentalmente funziona finchè la pallina va avanti. I due Point vengono chiamati new e last interception e sono rispettivamente l'urto precedente ed il successivo nell'ottica di una possibile successione di urti. Quando viene effettuato un lancio, calculateFirstHit trova il primo urto, che è uno dei Point in entrata alla funzione (new_interception, appunto), mentre last_interception è inizializzato come il punto da dove parte il lancio. La funzione svolge il suo compito finchè la x di new_interception è maggiore di quella di last_interception. Ciò è possibile inizializzando all'inizio della funzione una Line di nome path che rappresenta il lancio iniziale (facendo uso della Line di System col medesimo compito). A ogni iterazione il ciclo trova la prossima intersezione intersecando path con una delle due sponde (a seconda di quella con cui ha urtato in precedenza) e riaggiorna path usando l' eq.1.1, last_interception e new_interception di conseguenza. Inoltre, la funzione include la possibilità che la pallina torni indietro.

Il lettore più attento potrebbe chiedersi cosa succeda in caso l'ultimo rimbalzo generi una traiettoria dalla pendenza minore (in valore assoluto) rispetto a quella delle sponde, poiché in questo caso new_interception si troverebbe sì sulla sponda corretta ma ad ascisse minori rispetto a quella di last_interception. getFinalPoint prevede anche questa casistica. Difatti a sponde divergenti corrisponde sempre una pallina che si muove solamente nel verso positivo delle x, perciò è sufficiente realizzare un if che tiene conto della divergenza delle linee per effettuare i suoi calcoli. In caso le sponde siano divergenti, infatti, se new_interception ha l'ascissa minore di last_interception la funzione pone new_interception come l'intersezione tra path e la retta $x = l$.

1.3 Resa grafica e interfaccia utente

La resa grafica del programma ruota attorno alle librerie SFML e TGUI. L'interfaccia utente consiste in una serie di EditBoxSlider (ossia delle caselle di testo con slider associato) e due bottoni, uno per lanciare la pallina con annessa animazione ed uno per lanciarla n volte generando casualmente (ma con distribuzione gaussiana, deviazione standard e media fissate dall'utente) le condizioni iniziali y_0 e θ_0. Le funzioni principali che governano questa parte di programma sono:

- **fillVector**, che, come dice il nome, datogli due punti riempie un vector (anche questo in input) con una serie di Point compresi tra essi sulla retta che li connette. Chiaramente la quantità e distanza di questi punti dipende dalla velocità che vogliamo dare alla pallina, che nel nostro caso viene riadattata alle dimensioni del sistema;
- **Setup::getNormalDistribution**, che esegue dei comandi simili a quelli di getFinalPoint ma che ignora le animazioni. Tuttavia, questa genera, utilizzando ROOT, un'immagine con gli istogrammi dei punti e degli angoli finali;

fillVector viene chiamato all'interno di ogni if di getFinalPoint in modo tale che alla chiusura del while di quest'ultima esista un vector che contiene fondamentalmente tutta la traiettoria della pallina. Disegnare il tutto diventa a questo punto molto semplice dal momento che basta avanzare di una posizione sul vector a ogni iterazione del ciclo while principale della finestra SFML. Sia Setup::run (la funzione che impacchetta tutti i metodi della sezione precedente) che Setup::getNormalDistribution vengono chiamati all'interno del ciclo while una volta per click sul bottone associato, essendo essi legati ciascuno ad un booleano corrispondente. Una volta premuti i pulsanti, il programma si premurerà di stampare a schermo i valori all'interno delle EditBox associate. Al fine di testare a dovere il programma, Setup::run restituisce un oggetto di tipo Angle_and_point.

Resta a questo punto da chiarire come vengano mostrati gli istogrammi. Essi sono legati ad uno sprite la cui texture viene cambiata a ogni click sul bottone della gaussiana; ciò è reso molto semplice dal fatto che ROOT genera l'immagine sempre con lo stesso nome, quindi basta dire alla texture di trovare il file col nome da noi dato ed aggiornarsi.

Notiamo infine come SFML abbia le ordinate invertite, più precisamente l'asse y è diretto verso il basso. Questo potrebbe causare problemi dal momento che lavoriamo direttamente con le coordinate di SFML, quindi piuttosto che traslare il sistema trasliamo l'origine all'interno della finestra e costruiamo tutto in funzione della sua posizione; i problemi sono risolti capovolgendo il Setup in fase di costruzione: i suoi costruttori sono infatti entrambi programmati per cambiare il segno di y_0 e θ_0 e quindi per capovolgere il sistema rispetto a quello che l'utente fornisce, al fine di dare in resa grafica un sistema coerente con i dati inseriti.

2 Istruzioni di compilazione, testing ed esecuzione

Assumendo che l'utente abbia già installato ROOT, egli potrà scaricare il programma dalla repository https://github.com/Feynman77/biliardo con il comando:

```
git clone --depth 1 https://github.com/feynman77/biliardo.git
```

Dopo aver clonato (o decompresso, se lo si ha ricevuto da altre fonti in formato zip) il file relativo al programma, l'utente dovrà scaricare le librerie necessarie alla sua esecuzione in questo modo:

```
sudo apt update && sudo apt upgrade -y && sudo apt install cmake libsfml-dev -y
sudo apt install ninja-build
sudo add-apt-repository ppa:texus/tgui
sudo apt install libtgui-1.0-dev
```

Una volta fatto ciò, sarà sufficiente eseguire i seguenti comandi in ordine:

```
cd biliardo
mkdir build
cmake -S . -B build
cmake --build build
cd build
./biliardo
```

Consigliamo in fase di esecuzione di generare una gaussiana prima di aver effettuato il primo lancio, poiché l'operazione di simulare il tutto per la prima volta è più pesante delle altre (ipotizziamo che ciò sia dovuto alla creazione della Tcanvas); da lì in poi, tenendo un numero di iterazioni ragionevolmente basso (attorno alle 10^4) si potrà contemporaneamente simulare e lanciare la pallina non avendo interruzioni all'animazione; siano le iterazioni di ordini di grandezza maggiori, consigliamo di svolgere le due azioni separatamente per poter apprezzare l'animazione in maniera fluida. Ricordiamo inoltre che nel momento in cui si preme il pulsante "Normal distribution" questo simula utilizzando i dati nella sezione sottostante, non quelli dell'ultima volta che si è premuto "Throw".
Segnaliamo inoltre come possa capitare, su alcune macchine, che venga mostrato a terminale un errore di questo tipo:

```
/usr/include/c++/9/bits/shared_ptr_base.h:155:6: runtime error:
member call on address 0x608000056c20 which does not point to an
object of type '_Sp_counted_base' 0x608000056c20: note: object
has invalid vptr
```

Crediamo questo possa dipendere da ROOT, tuttavia la causa ci è ignota dal momento che ci succede su una macchina su due, ergo non riusciamo a riprodurre l'errore al fine di trovarne la fonte. Questo comunque non dovrebbe interferire con l'utilizzo del programma, così come nemmeno:

```
    ==2805==ERROR: LeakSanitizer: detected memory leaks
```

anch'esso probabilmente dipendente da ROOT.
Portiamo all'attenzione del lettore che, nonostante il codice sia costruito per aprire la finestra nel centro dello schermo, la prima volta che il programma viene eseguito questa appare nella parte bassa del display; non conosciamo la ragione di questa anomalia, supponiamo dipenda da SFML.
Di seguito, invece, il comando per l'utente che volesse effettuare il testing del programma (sempre all'interno della cartella build):

```
./biliardo_test
```

3 Strategia di testing

Dal momento che la GUI, così come la generazione delle gaussiane, non è testabile in maniera efficace, il testing si è concentrato su findInterception, calculateFirstHit e getFinalPoint. Essendo queste estremamente dipendenti dalle altre funzioni di calcolo, abbiamo giudicato sufficiente mettere alla prova solo queste tre.

- il testing di findInterception è semplicemente verificare una serie di intersezioni tra rette (la cui correttezza è stata verificata tramite il software Geogebra);
- calculateFirstHit è stato testato con tre sistemi, uno in cui non si colpiscono sponde, uno in cui il primo urto è con la sponda superiore e uno in cui è con quella inferiore;
- abbiamo verificato che getFinalPoint calcoli correttamente gli esiti di svariati lanci, anche questi confrontati con i risultati ottenuti disegnando il sistema sul software GeoGebra;

4 Risultati

Eseguendo il programma, possiamo osservare diverse situazioni interessanti, con diversi gradi di complessità.

4.1 Lancio dritto

Un caso interessante, nella sua scolasticità, è quello di un lancio dritto. Inserendo limitata deviazione standard in entrambi i parametri (in modo tale che non si colpiscano sponde in nessun lancio), possiamo osservare delle gaussiane sia negli angoli che nelle posizioni finali, e particolarmente interessante è quella dell'angolo, dove la media è coerente con l'angolo di lancio così come la deviazione standard (fig.2).

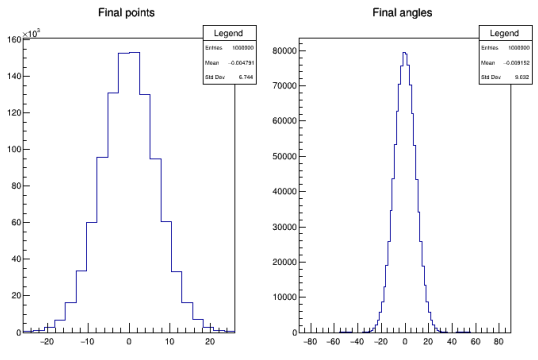

Figure 2: Risultati ottenuti con $y_0 = \theta_0 = 0, r_1 = r_2 = 31, l = 40, \sigma_{y_0} = 2, \sigma_{\theta_0} = 9$

4.2 Sponde leggermente convergenti con lunghezza variabile

Un caso più complesso è quello di sponde leggermente convergenti associate ad un lancio inclinato. Vediamo, nel grafico in fig.3, come già a lunghezze abbastanza ridotte si inizi ad osservare un certo tipo di figura: due gruppi di picchi centrati sull'inclinazione finale prevista e la sua opposta. Ciò è facilmente spiegabile: quello che stiamo vedendo non sono altro che la probabilità che la pallina faccia un numero specifico di urti, ossia ad ogni picco corrisponde una quantità specifica di urti per lancio. Quello che ci aspettiamo aumentando la lunghezza è che questi picchi diventino man mano più densi, che è esattamente quello che poi osserviamo in fig.5.

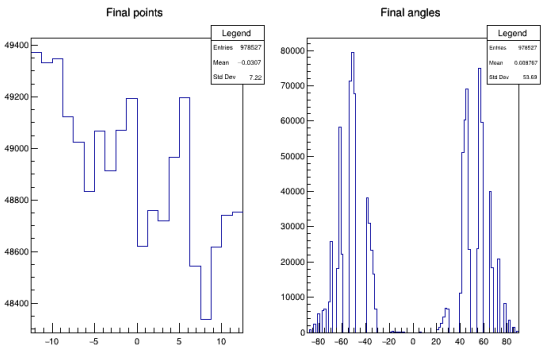

Figure 3: Risultati ottenuti con $y_0 = 0, \theta_0 = 45, r_1 = 14, r_2 = 12.5, l = 120, \sigma_{y_0} = 0, \sigma_{\theta_0} = 9$

4.3 Sponde parallele, lancio a 80° con alta σ_{θ_0}

Siano invece le sponde parallele, un lancio particolarmente inclinato può generare effetti interessanti sugli angoli finali. Otteniamo infatti, come si vede in fig.4, una metà di gaussiana nella

parte positiva degli angoli ed una nella metà negativa, il che è un comportamento prevedibile:

d'altronde le sponde parallele non fanno altro che cambiare il segno dell'angolo iniziale, e la distribuzione con cui abbiamo lanciato le palline è coerente con una distribuzione negli esiti di tipo gaussiano, seppur atipica in questo caso particolare.

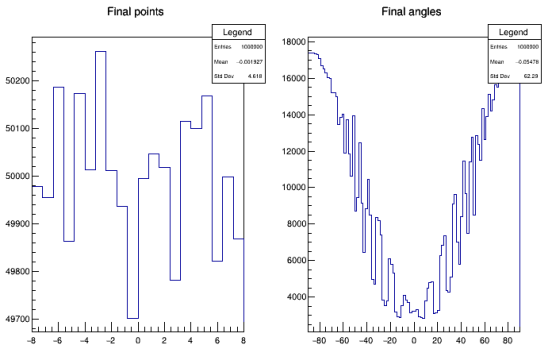

Figure 4: Risultati ottenuti con $y_0 = 0, \theta_0 = 80, r_1 = r_2 = 8, l = 140, \sigma_{y_0} = 0, \sigma_{\theta_0} = 40$

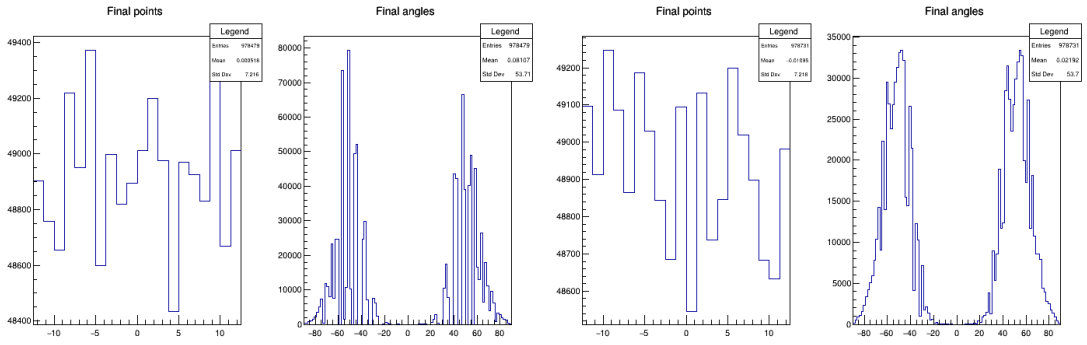

Figure 5: Risultati ottenuti con gli stessi parametri di fig.3 salvo la lunghezza, che è rispettivamente 265 e 840.

Appendice

Dimostrazione della formula 1.1

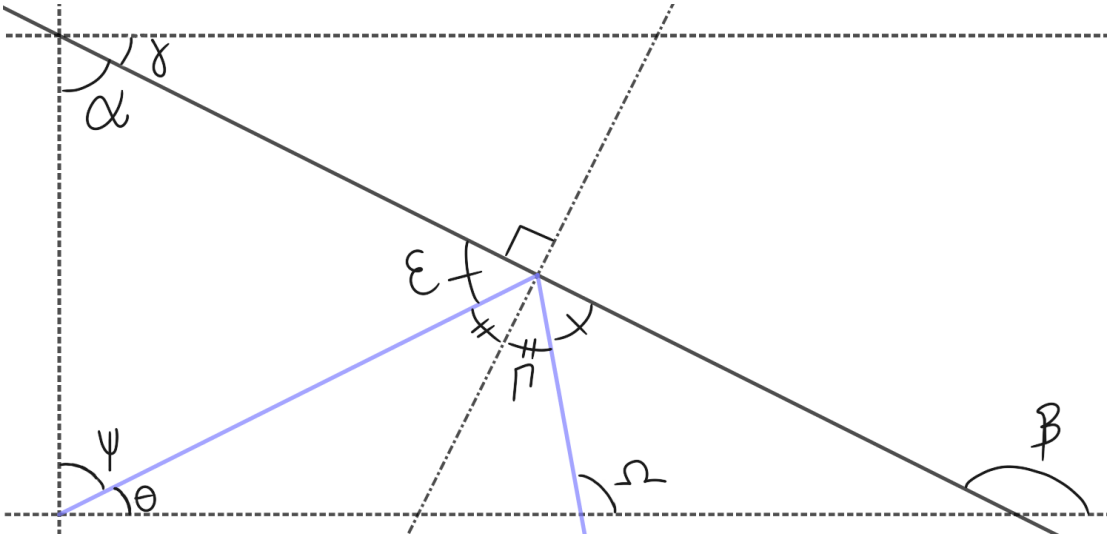

Figure 6: Rappresentazione sistema geometrico.

Sia la linea superiore la retta $y = m_t x + q$.$\Rightarrow \alpha = 90 - arctan|m_T|$.
Dalla fig.6, è ovvio che $\varepsilon = 180 - \alpha - \psi$; tenendo conto inoltre che $\psi = 90 - \theta \Rightarrow \varepsilon = 180 - 90 + arctan|m_T| - 90 + \theta = arctan|m_T| + \theta$.
La dimostrazione è conclusa se troviamo il coefficiente angolare m della retta che rappresenta la traiettoria dopo il rimbalzo, per fare ciò ci avvaliamo delle formule goniometriche relative alla tangente della differenza di angoli a $tan(\Omega)$.
Dobbiamo prima trovare una formula "amichevole" per Ω. Questo è supplementare all'angolo a sua volta supplementare alla somma di θ e 2Γ, quindi $\Omega = 180 - (180 - \theta - 2\Gamma) = \theta + 2\Gamma$.
Dal momento che $2\Gamma = 180 - 2\varepsilon$ allora $\Omega = 180 - 2\varepsilon + \theta = 180 + \theta - 2arctan|m_T| - 2\theta = 180 - \theta - 2arctan|m_T|$ dove abbiamo utilizzato i risultati precedenti di ε.
Applicando la formula della tangente di una differenza di angoli e della tangente di angoli supplementari otteniamo dopo qualche passaggio algebrico la formula 1.1. Il lettore potrebbe notare che assumiamo le sponde convergenti; ciò è corretto, lasciamo al lettore la verifica che la formula risultante per sponde divergenti sia completamente identica.