Progetto Programmazione per la Fisica: Dinamica in un biliardo triangolare

Riccardo Grandicelli

1. Introduzione

Il programma intende simulare il movimento di un oggetto all'interno di un biliardo triangolare, così come descritto nelle istruzioni che ci sono state fornite [1]. La particella che si muove all'interno del biliardo può urtare tutte e tre le pareti (quella superiore, quella inferiore e quella di partenza), e prosegue il suo moto finché non esce dal biliardo dalla parete di destra. Gli urti con le pareti sono supposti elastici.

Il programma fornisce posizione finale e angolo di uscita dal biliardo della particella. È possibile sia determinare le coordinate finali di una singola particella, con coordinate di partenza fornite dall'utente, sia determinare le coordinate finali di un certo numero di particelle, con coordinate di partenza generate casualmente seguendo una distribuzione gaussiana.

Il programma può infine calcolare media, deviazione standard, coefficiente di simmetria (indice di simmetria di Fischer [2]) e di appiattimento (indice di curtosi di Pearson [3]) delle distribuzioni di posizione e angolo finali delle particelle generate casualmente.

In allegato ho aggiunto una macro di ROOT che permette di visualizzare graficamente la distribuzione di posizione e angolo, sia iniziali che finali.

2. Classi, metodi e funzioni

Le classi, i metodi e le funzioni implementate per la risoluzione del problema sono state divise in tre namespaces:

- Gen: contiene i function objects
- Stats: contiene le classi e i metodi per il calcolo delle statistiche delle distribuzioni finali di posizione e angolo delle particelle
- Ric: contiene tutti gli altri metodi, classi e funzioni utili alla risoluzione del problema

2.1 Namespace Ric

Qui sono implementate le classi Point, Line e Particle:

- **Point:** è una coppia di double che definiscono le coordinate x e y di un punto nello spazio bidimensionale
  - L'**operatore !=** serve per definire l'uguaglianza tra due punti. Poiché è difficile usare l'operatore == per i double, tale operatore != considera uguali due punti se la differenza tra le x e la differenza tra le y sono entrambe, in valore assoluto, minori di 0.001 (cioè praticamente uguali a 0)
- **Line:** rappresenta una retta nello spazio bidimensionale, nella forma $ax + by + c = 0$ La scelta di utilizzare i parametri a, b e c per definire una retta (anziché usare

coefficiente angolare e intercetta) è dovuto al fatto che ciò pone meno problemi nel caso in cui dovesse essere definita una retta parallela all'asse y.
    - La retta può essere inizializzata mediante tre costruttori: il primo utilizza coefficiente angolare e intercetta, il secondo due punti nello spazio, e il terzo una particella (che, come illustrerò in seguito, è definita in base a un punto e all'angolo con l'asse x, parametri sufficienti per definire una retta)
    - I metodi **m** e **q** restituiscono rispettivamente coefficiente angolare e intercetta, mentre il metodo **set_new** serve per modificare a, b e c, partendo da una particella (in pratica è un costruttore per una retta già inizializzata)
- **Particle:** definisce una particella nel biliardo, caratterizzata da una posizione e dall'angolo formato dalla sua velocità con l'asse x.
    - Il modulo della velocità non è stato inserito in quanto è irrilevante ai fini di determinare la posizione finale della particella (visto che non è presente l'attrito, e che gli urti sono elastici).
    - L'angolo può essere compreso tra -90° e 90° (in radianti). Nel costruttore non c'è nessun controllo su questa invariante di classe, ma nel codice ogni volta in cui viene inizializzata una particella si controlla che il valore dell'angolo sia nell'intervallo appena definito.
    - I metodi **set_position** e **set_angle** servono a modificare angolo e posizione della particella, mentre i metodi **rotate_forward** e **rotate_backward** ruotano la velocità della particella rispettivamente in senso antiorario o orario, di un angolo compreso tra 0° e 90°. Il funzionamento di questi metodi è spiegato in Appendice 1.

Sempre nel namespace Ric sono definite le funzioni intsec, ort e find_angle

- **Intsec:** restituisce il punto di intersezione tra due rette.
- **Ort:** data una retta e un punto, la funzione trova la perpendicolare alla retta passante per quel punto, sfruttando il fatto che il prodotto tra i coefficienti angolare di due rette perpendicolari è -1.
- **Find_angle:** trova l'angolo acuto compreso tra due rette. Il suo funzionamento è spiegato in Appendice 2.

2.2 Namespace Gen

Qui sono definite le classi (function objects) PartG e PartM:

- **PartG:** serve per generare posizioni e angoli iniziali delle particelle. Si inizializza con i parametri della distribuzione gaussiana della posizione iniziale in y (la posizione di partenza in x è 0) e i parametri della distribuzione gaussiana dell'angolo iniziale. L'**operatore ()** permette di generare particelle casualmente seguendo tali distribuzioni gaussiane. La generazione viene ripetuta fino a quando la particella generata non ha y compresa tra -r1 e r1 e angolo compreso tra -90° e 90°.
- **PartM:** serve per trovare la posizione e l'angolo finali di una particella. Si inizializza con i parametri che definiscono il biliardo (r1, r2 e l). L'**operatore ()** prende in input una particella e ne cambia posizione e angolo portandola dalla sua configurazione iniziale a quella finale. Nel paragrafo 3 è spiegato il metodo utilizzato per trovare la configurazione finale.

2.3 Namespace Stats

Qui sono definite le classi Statistics e Sample:

- **Statistics:** è un insieme di quattro double: media, deviazione standard, coefficiente di simmetria e di appiattimento
- **Sample:** definisce un campione di particelle. Una variabile Sample può essere inizializzata con un vettore di Particle.
    - Il metodo **statistics_y** permette di calcolare, e di salvare in una variabile Statistics, le statistiche relative alle posizioni in y delle particelle contenute nella variabile Sample.
    - La media è calcolata con la formula $(\sum_{i=1}^{N} y_i)/N$, dove N è il numero di elementi della variabile Sample, mentre la deviazione standard con la formula $\sqrt{(\sum_{i=1}^{N}(y_i - y_{medio})^2)/(N-1)}$. I coefficienti di simmetria e appiattimento sono calcolati come indicato rispettivamente in [2] e [3].
    - Il metodo **statistics_ang** permette di calcolare le statistiche relative agli angoli delle particelle nella variabile Sample, nello stesso modo in cui sono calcolate le statistiche per le posizioni in y.

3. Come trovare la configurazione finale delle particelle

L'operatore () del function object PartM permette di trovare la posizione e l'angolo di uscita di una particella, tramite il seguente procedimento:

1. Vengono generate le variabili Line *upborder*, *downborder*, *leftborder* e *rightborder* (i quattro bordi del biliardo), mentre la variabile Line *go* viene inizializzata a partire da posizione e angolo della particella da muovere (*go* indica la direzione del moto della particella). Vengono inoltre inizializzate le variabili Line *down_perp* e *up_perp*, rispettivamente le perpendicolari al bordo inferiore e superiore (che non dipendono dal punto di impatto della particella)
2. Inizia ora un ciclo while, nel quale a ogni iterazione *go* viene intersecata con i quattro bordi. Chiaramente solo due delle intersezioni saranno interne ai bordi del biliardo, di cui una è la posizione attualmente occupata dalla particella e l'altra il punto d'impatto della particella con il biliardo.
3. Le quattro condizioni if successive servono proprio per vedere quale dei quattro bordi del biliardo urta la particella, controllando che tale punto d'impatto sia diverso dalla posizione attualmente occupata dalla particella. A seconda del bordo colpito, la particella ha quattro possibilità:
    - Se viene urtato il bordo a sinistra, l'angolo della particella viene cambiato di segno (dato che la perpendicolare al punto d'impatto è parallela all'asse x). Il punto d'impatto diventa la nuova posizione della particella, e i membri privati di *go* vengono aggiornati ai nuovi metodi privati della particella.
    - Se vengono urtati il bordo superiore o inferiore, la funzione **find_angle** trova l'angolo tra *go* e la perpendicolare al bordo colpito. A questo punto il metodo **rotate_forward** ruota l'angolo della particella di un angolo pari a quello appena trovato, in senso antiorario. Se ora l'angolo della particella corrisponde a quello di *up_perp* o *down_perp* (ossia l'arcotangente del loro coefficiente angolare), la rotazione è

avvenuta nel senso corretto, quindi viene effettuata un'altra rotazione. Se invece i due angoli non corrispondono, viene effettuata per tre volte una rotazione in senso orario con il metodo **rotate_backward**. In entrambi i casi, l'angolo della particella dopo le rotazioni è il suo angolo finale in seguito all'urto. Il punto d'impatto diventa dunque la nuova posizione della particella e i membri privati di go vengono aggiornati ai nuovi metodi privati della particella.

- Se viene urtato il bordo a destra, la particella è uscita dal biliardo. Posizione e angolo della particella vengono aggiornati al punto d'impatto finale, e il ciclo termina.

4. L'ultima condizione if serve solo se *go* non interseca nessuno dei quattro bordi. Ciò avviene solamente nello sfortunato caso in cui la particella urta uno dei vertici del biliardo.
   - Se una particella dovesse urtare il vertice di un biliardo, verrebbe rispedita all'indietro nella stessa direzione da cui è arrivata. Ciò significa che essa ripercorrerebbe tutto il percorso fatto fino a quel momento e tornerebbe nella posizione iniziale, invertendo l'angolo di partenza e ripartendo da capo. L'ultima condizione evita tutto questo percorso, facendo ripartire direttamente dall'inizio la particella, ma cambiando il segno del suo angolo iniziale.
   - Se anche in questo caso la particella urta uno dei bordi del biliardo, il ciclo termina (significa che la particella rimarrà per sempre nel biliardo senza mai uscire). La funzione **main** gestirà poi questa eventualità, informando l'utente di quante particelle non escono dal biliardo.

4. Test implementati per verificare la correttezza del codice

Nel source file project.test.cpp sono implementati alcuni test.

- I test relativi alle funzioni e ai metodi degli oggetti geometrici (ad esempio l'intersezione tra due rette, o la costruzione di una retta perpendicolare passante per un punto) sono stati implementati svolgendo manualmente i calcoli, con semplici formule di geometria analitica. Anche per i test relativi alle statistiche i calcoli sono stati svolti con una calcolatrice.
- I test relativi alle configurazioni finali delle particelle sono stati invece implementati usando la calcolatrice grafica di GeoGebra.
  - Si disegna il biliardo, con i parametri r1, r2 e l impostati nei test, si disegna il punto di partenza e la retta passante per quel punto con angolo uguale all'angolo iniziale.
  - Si trova quindi l'intersezione tra la retta e uno dei bordi del biliardo e si trova la retta perpendicolare a tale bordo passante per il punto di intersezione appena trovato.
  - Si trova la nuova direzione della particella realizzando una simmetria assiale della retta che identifica la direzione di incidenza della particella rispetto alla retta perpendicolare al bordo. In questo modo l'angolo di incidenza è uguale all'angolo con cui la particella viene riflessa.
  - Si ripetono i due punti precedenti fino a quando si interseca il bordo di uscita dal biliardo (x=l). A questo punto si calcolano posizione in y del punto di intersezione con tale bordo, l'angolo della retta che interseca il bordo rispetto all'asse x, e si inseriscono tali valori nei test.

5. Funzione main

La funzione main del programma attende un comando dall'utente: tale comando può essere soltanto uno dei seguenti caratteri: 'm', 'a', 's' e 'q'.

- 'm' consente di trovare la configurazione finale di una singola particella: il programma chiede in input all'utente alcuni dati relativi al biliardo e allo stato iniziale della particella, poi calcola posizione e angolo finale (se la particella riesce a uscire dal biliardo, altrimenti viene visualizzato un messaggio di errore).
- 'a' seguito da un numero intero n genera n particelle in modo pseudocasuale e ne trova la configurazione finale. Tutti i dati relativi al biliardo e ai parametri della distribuzione gaussiana di posizione e angolo iniziali delle particelle vengono letti dal programma dal file "input.txt". In tale file i dati vanno inseriti nel seguente ordine:
    - r1
    - r2
    - l
    - media y iniziale
    - sigma y iniziale
    - media angolo iniziale
    - sigma angolo iniziale

  Il programma salva le configurazioni iniziali delle particelle generate nel file "outinit.txt" e le configurazioni finali nel file "outfin.txt". La configurazione finale viene salvata soltanto se la particella esce dal biliardo. Il programma informa comunque l'utente di quante particelle non escono dal biliardo.

- 's' permette il calcolo delle statistiche, sia per la distribuzione delle particelle in input che per quella delle particelle in output. Prima di utilizzare questo comando bisogna generare un certo numero di simulazioni del moto con il comando 'a'. I dati salvati in "outinit.txt" e "outfin.txt" verranno anche salvati in due vettori, rispettivamente *input* e *output*. Tali vettori saranno usati per estrarre i dati di cui calcolare le statistiche (con i metodi statistics_y e statistics_ang).
- 'q' permette di uscire dal programma.

6. Come compilare, testare ed eseguire il codice

La compilazione è effettuata con CMake, mediante in file CMakeLists.txt.

- Per attivare i test, occorre commentare nel file CMakeLists.txt (ossia aggiungere un # davanti) l'ultima riga prima dell'if (ovvero add_executable….), e togliere il commento alle righe all'interno dell'if. Per disabilitare i test e attivare la compilazione del main, occorre effettuare il processo inverso.
- La compilazione si effettua scrivendo sulla command line i comandi "cmake -S . -B build -DCMAKE_BUILD_TYPE=Debug" e in seguito "cmake --build build". L'eseguibile si trova ora all'interno della directory "build", ed è nominato "project.t" (per i test) o "project" (per il main).
- Il metodo di inserimento dei dati in input e della visualizzazione dei dati in output è già stato descritto nel paragrafo 5.

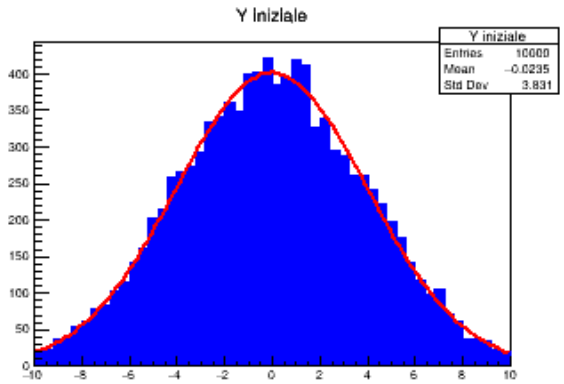
- Ogni volta che viene eseguito il programma (con “./build/project”), il contenuto di “outinit.txt” e “outfin.txt” viene cancellato
- Per visualizzare gli istogrammi relativi alle configurazioni iniziali e finali delle particelle, basta avviare ROOT e digitare il comando “.x graphs.C”. Si aprirà una finestra con quattro istogrammi, relativi rispettivamente alle posizioni in y iniziali, agli angoli iniziali, alle posizioni in y finali e agli angoli finali. Ogni istogramma è diviso in 50 bin. I dati per i primi due istogrammi sono estratti da “outinit.txt”, i dati per gli ultimi istogrammi da “outfin.txt”.
- Per comodità, si suggerisce di eseguire il programma direttamente dalla command line di ROOT.

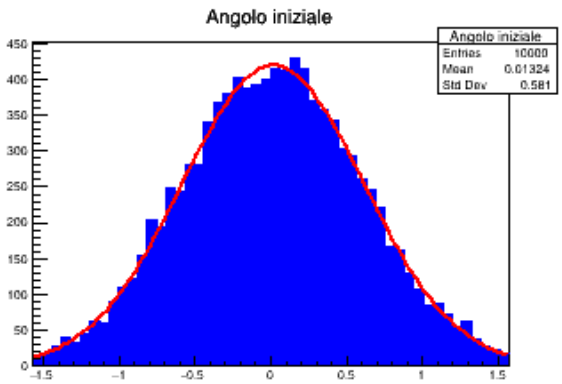
7. Alcune considerazioni sui risultati ottenuti dal programma

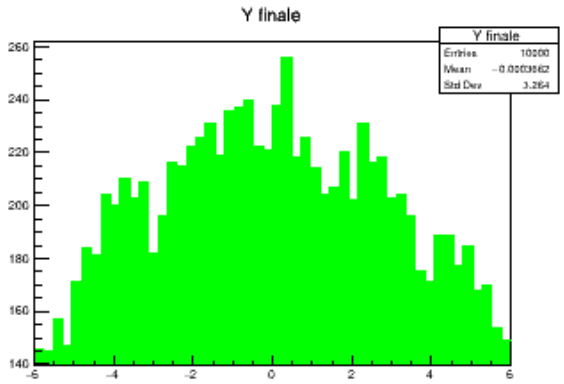
Aggiungo alcune considerazioni su come varia la distribuzione di posizione e angolo delle particelle in uscita al variare della lunghezza del biliardo (ossia al variare di l, mantenendo fissi gli altri parametri).

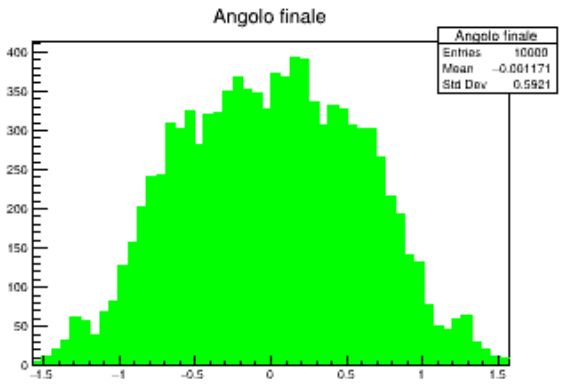
Prendiamo un biliardo con r1=10 e r2=6, e impostiamo la media di posizione e angolo iniziale a 0, la deviazione standard della posizione iniziale a 4 (in modo da generare un discreto numero di particelle con posizione iniziale vicina al bordo) e deviazione standard dell’angolo inziale a 0.6 (in modo da generare un discreto numero di particelle con angoli vicini a -90° e 90°, considerando che 90° è circa uguale a 1.6 radianti). Generiamo diecimila simulazioni del moto per ogni valore di l.

Con l=5 otteniamo i seguenti grafici:

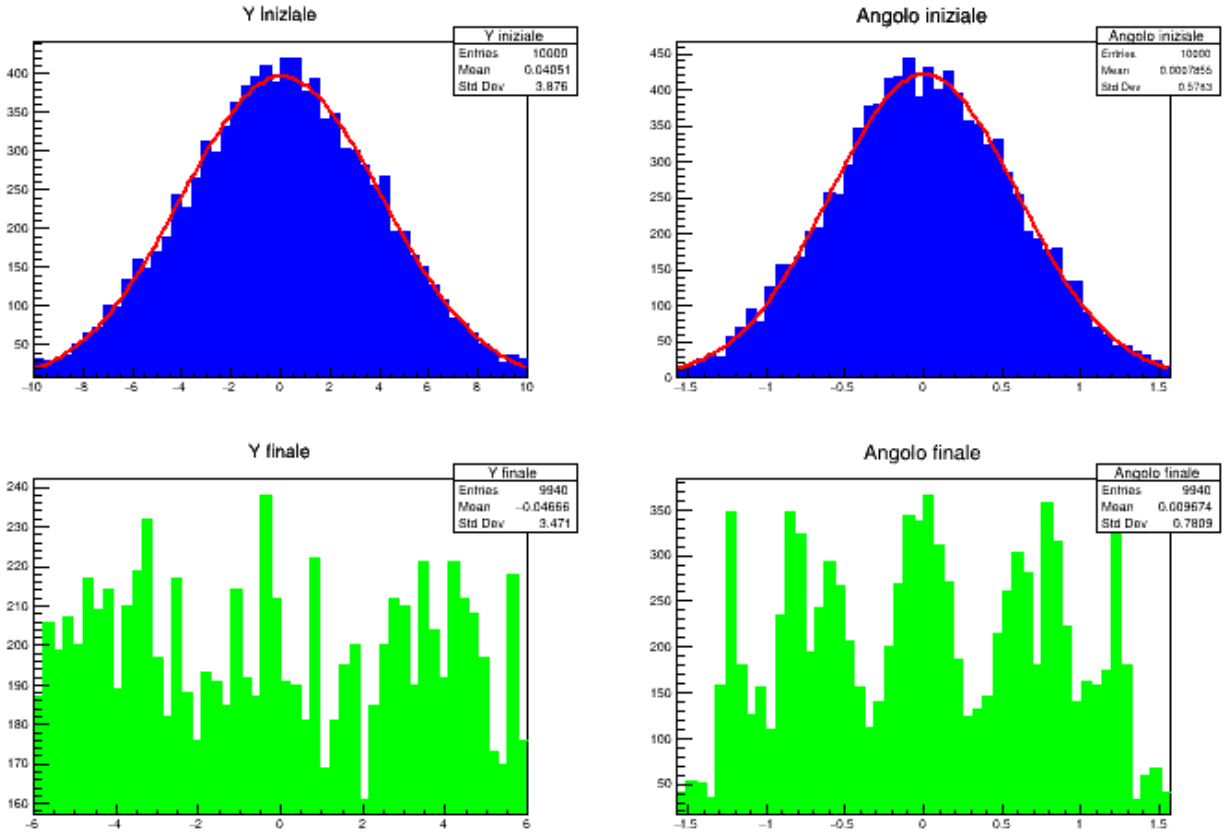








Mentre con l=30 abbiamo:



Per l piccolo, la forma delle distribuzioni di posizioni e angoli finale è vicina a una gaussiana. Più si aumenta l, più la distribuzione della posizione finale diventa all'incirca uniforme. I valori dell'angolo finale, invece, sono distribuiti in modo circa uniforme, mentre quasi nessuna particella esce con valori di angolo vicini a 90° o a -90°.

Si nota inoltre che all'aumentare di l il numero di particelle che non esce dal biliardo aumenta (0 per valori piccoli di l, mentre si arriva anche a 60 per l=30). Inoltre, se le distribuzioni di partenza di posizione e angolo sono simmetriche (ossia la media è 0), lo sono anche le distribuzioni finali. In particolare, la distribuzione degli angoli finali mantiene circa la stessa deviazione standard di quella iniziale.

È curioso anche notare come alcuni bin dell'istogramma delle posizioni finali abbiano molti meno ingressi di altri, in modo apparentemente casuale. Ciò avviene in particolare all'aumentare di l.

Cambiando i valori di r1 e r2 si ottengono all'incirca gli stessi risultati.

Bibliografia

[1] Repository Github per il corso di Programmazione per la Fisica

https://github.com/Programmazione-per-la-Fisica/progetto2022/blob/main/biliardo.md

[2] Università degli studi di Verona, Laboratorio di Probabilità e Statistica, docente Bruno Gobbi, diapositiva 4

https://www.corsi.univr.it/documenti/OccorrenzaIns/matdid/matdid905822.pdf

[3] Università degli studi di Verona, Laboratorio di Probabilità e Statistica, docente Bruno Gobbi, diapositiva 13

https://www.corsi.univr.it/documenti/OccorrenzaIns/matdid/matdid905822.pdf

Appendice A: metodi rotate_forward e rotate_backward di Particle

Il metodo **rotate_forward** della classe Particle riceve in input un angolo da 0 a 90°, e ruota la direzione del moto della particella in senso antiorario di tale angolo (ossia somma l'angolo in input all'angolo della particella). Nel caso in cui la direzione del moto della particella formi con l'asse x un angolo ottuso, l'angolo che deve avere la particella è il corrispondente angolo acuto negativo (l'angolo della particella con l'asse x va da -90° a 90°). Quindi, se a seguito della rotazione l'angolo della particella supera i 90°, rotate_forward sottrae 180° a tale angolo, in modo da ottenere un angolo acuto negativo.

Il metodo **rotate_backward** è identico, ma la rotazione avviene all'indietro (l'angolo in input viene sottratto), e se l'angolo ottenuto è inferiore a -90°, rotate_backward aggiunge 180°, in modo da avere un angolo acuto positivo.

Appendice B: funzione find_angle

La funzione **find_angle** trova l'angolo acuto positivo tra due rette (ossia due variabili Line fornite in input). L'angolo compreso tra due rette è la differenza tra gli angoli che queste rette formano con l'asse x (angolo che può andare da 0 a 180°).

L'angolo che una retta forma con l'asse x è l'arcotangente del suo coefficiente angolare. Se tale angolo è però ottuso, l'arcotangente del coefficiente angolare è il corrispondente angolo acuto negativo. Per questo, se il coefficiente angolare della retta è negativo, find_angle aggiunge 180° a tale angolo, trovando così il corrispondente angolo ottuso positivo.

Una volta trovati gli angoli che le due rette formano con l'asse, la funzione calcola il valore assoluto della loro differenza, che corrisponde all'angolo compreso tra le due rette. Tale angolo può essere acuto o ottuso, ma la funzione cerca l'angolo acuto. Per questo, se l'angolo è ottuso, la funzione restituisce l'angolo supplementare all'angolo così trovato.