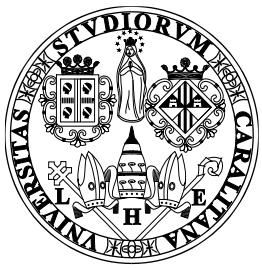

Università degli Studi di Cagliari

Facoltà di Scienze Economiche, Giuridiche e Politiche

Corso di Laurea in Economia e Gestione Aziendale

Simulazione di variabili casuali e metodo Monte Carlo: aspetti teorici e applicazioni

Relatore:

Prof. Massimo Cannas

Tesi di Laurea di:

Davide Christian Mancosu Bustos

Anno Accademico 2021/2022

INDICE

INTRODUZIONE

Gli algoritmi diventano sempre più strumenti di supporto necessari per lo svolgimento e la presa delle decisioni per le attività di marketing e, generalmente, d'impresa. Un algoritmo è quello Monte Carlo che, pur di non recente nascita, negli ultimi anni ha visto un aumento nel suo utilizzo dato sia per la sua efficienza che per la sua semplicità, infatti, qualsiasi fenomeno rappresentabile tramite un'equazione può essere sottoposto al metodo. Le sue peculiarità fanno sì che anche i settori di applicazioni sia molteplici.

L'elaborato, dunque, è volto ad analizzare l'utilizzo del metodo Monte Carlo nelle decisioni aziendali.

L'analisi e lo sviluppo di tale argomento nasce da un forte interesse personale di voler applicare congiuntamente quanto imparato nel corso della triennale e quanto più ormai viene richiesto nel mondo, ossia le conoscenze informatiche e statistiche. Anche l'esperienza fatta durante il tirocinio ha sicuramente accresciuto l'interesse ad approfondire l'argomento e a vedere le sue relazioni con quanto studiato finora. L'obiettivo del presente elaborato è verificare l'effettiva utilità di tale metodo nel settore finanziario, in particolare quello del mercato azionario.

L'obiettivo del presente elaborato è verificare l'effettiva utilità di tale metodo nel settore finanziario, in particolare quello del mercato azionario.

Per la realizzazione di questo lavoro si è fatto riferimento a più testi, indicati nella bibliografia, al quale si rimanda il lettore per ogni ulteriore approfondimento.

La tesi è organizzata in tre capitoli composti da nozioni teoriche affiancate da blocchi di codice in R personalmente implementati volti a rafforzare la comprensione dei concetti esposti. Vi sono inoltre presenti grafici e tabelle riassuntivi dei passaggi più importanti.

Nel primo capitolo viene analizzata la generazione di numeri casuali, partendo dalla differenza tra True e Pseudo Random Number Generator, differenza interessante non solo sotto un punto di vista teorico ma anche sotto quello applicativo. Si fa inoltre un cenno alla generazione di numeri casuali tramite l'algoritmo Von Neumann che, per quanto possa sembrare surreale, ci darà la possibilità di simulare il lancio di una moneta bilanciata da una moneta non bilanciata. Considereremo il più classico generatore di numeri casuali, ovvero il Linear Congruential Generator, e vari metodi di generazione partendo da una variabile casuale qualsiasi. A conclusione del capitolo vi è un approfondimento della generazione di variabili casuali così da completare il quadro sopra il linguaggio di programmazione scelto.

Nel secondo capitolo, invece, si passerà all'analisi del Metodo Monte Carlo presentandone brevemente cenni storici e applicazioni d'uso, primo tra tutti il calcolo di un integrale. Successivamente all'introduzione vengono presentate le proprietà degli stimatori e della varianza degli stimatori di Monte Carlo, dove è stato doveroso introdurre la legge che sta alla base del metodo Monte Carlo, ossia la Legge dei Grandi Numeri. Per concludere il capitolo viene presentato un esempio applicativo che riassume quanto esposto nel corso del capitolo, ossia l'approssimazione della distribuzione normale con il metodo Monte Carlo.

L'ultimo capitolo è dedicato all'applicazione economica del metodo, calcolando i rendimenti futuri di un titolo azionario con la simulazione Monte Carlo. Il titolo scelto è lo SPY, ossia uno

dei fondi più popolari tuttora esistenti. I calcoli si basano sui dati storici del titolo, i quali sono stati presi da YAHOO FINANCE. Al blocco di codice vengono affiancati dei commenti così da rendere la lettura più fluida. Oltre alla applicazione pratica vengono infine illustrati i risultati ottenuti dall'implementazione.

CAPITOLO 1 - GENERAZIONE DI VARIABILI CASUALI

1.1. GENERAZIONE DI VARIABILI CASUALI U(0,1)

Alla base dei metodi statistici delle simulazioni del comportamento di sistemi regolati da variabili aleatorie la generazione di numeri casuali è un argomento di grande rilievo. Ottenere una sequenza di *numeri casuali* è di cruciale importanza affinché la simulazione risulti il più attendibile possibile. Il concetto che sta alla base dei numeri casuali non è relativo a singoli numeri, ma è invece collegato alle proprietà delle successioni numeriche. A livello teorico i numeri che compongono la sequenza non dovrebbero presentare nessuna correlazione ma, come vedremo nel corso del paragrafo, a livello pratico si riscontrano problemi nel raggiungimento di numeri puramente casuali. Di fatto, si tratta di sequenze generate da algoritmi deterministici che sono casuali solo in apparenza.

Definizione 1.1. Un algoritmo è detto deterministico se per ogni istruzione esiste, a parità di dati d'ingresso, un solo passo successivo. Esiste uno e un solo possibile percorso dell'algoritmo e quindi a fronte degli stessi dati di partenza produrrà gli stessi risultati.

Per produrre tali sequenze ci si serve di un generatore di numeri casuali, ovvero un *RNG (random number generator)* [1]. Un RNG ha lo scopo di generare un valore casuale all'interno di un set predefinito di possibilità e può essere un dispositivo fisico o meccanico, oppure un algoritmo matematico.

Esistono due categorie principali di generatori di numeri casuali: *Pseudo Random Number Generators (PRNG)* e True Random *Number Generators (TRNG)*. Prima di trattare le differenze tra i due tipi di generatori e per poter capire al meglio cosa si intende per generatore di numeri casuali, bisogna ora specificare cosa si intende per RNG.

Definizione 1.2. Un RNG è un algoritmo per la generazione di una sequenza di variabili casuali U_1, U_2, …, tali che:

a) ogni U_i è uniformemente distribuita tra 0 e 1;
b) gli U_i sono indipendenti tra loro.

L'*uniformità* e l'*indipendenza* risultano essere le condizioni ''base'' affinché si possa parlare di un generatore di numeri casuali. Nello specifico l'indipendenza implica che tutte le coppie di valori devono essere non correlate fra di loro. Si tratta di un concetto molto importante che verrà illustrato nel corso dei successivi paragrafi.

1.2. TRUE AND PSEUDO RANDOM NUMBER GENERATORS (TRNG E PRNG)

1.2.1. TRNG

I generatori TRNG producono numeri casuali da un processo fisico. Tali dispositivi si basano su fenomeni microscopici come i fenomeni quantistici o il rumore termico. Il generatore più comunemente usato si basa su una sostanza sottoposta a decadimento atomico: le particelle subatomiche che compongono la sostanza in decomposizione si trasmutano in altre particelle in punti casuali nel tempo. Si tratta di *processi stazionari*, ovvero le loro caratteristiche non cambiano nel corso del tempo. Se è possibile misurare i tempi tra gli

eventi, allora possiamo formare una variabile casuale con una distribuzione nota. Più precisamente definiamo la *variabile 1* se il tempo di attesa del prossimo evento è maggiore di quello precedente e 0 altrimenti. Ovvero, per intervalli di tempo tra gli eventi osservati, s_1, s_2, …, definiamo:

$$X = \begin{cases} 1, & s_{2i-1} < s_{2i} \\ 0, & altrimenti. \end{cases} \quad (1.1)$$

X è una variabile casuale che segue la distribuzione di Bernoulli con parametro $p = 0{,}5$ per ogni $i = 1,2, \ldots$. Questo genere di variabile casuale è facilmente trasformabile in altre variabili casuali, per esempio in una distribuzione uniforme nell'intervallo (0,1).
Si tratta di generatori che sono, in teoria, completamente imprevedibili. I TRNG sono caratterizzati dalla lentezza per la produzione di numeri casuali e per questo vengono spesso utilizzati per generare il primo elemento di una sequenza, ovvero *il seme* dei generatori di numeri pseudo casuali, che darà luogo a sequenze di numeri pseudo casuali (si rimarca volutamente questo concetto di pseudo casualità perché ci ritornerà utile nella distinzione tra i due generatori) ad una velocità maggiore. La principale applicazione dei generatori TRNG è nella crittografia, dove vengono utilizzati per generare chiavi crittografiche necessarie per aumentare la sicurezza durante la trasmissione dei dati.
Nella pratica, i TRNG e PRNG differiscono per il livello di prevedibilità che esibiscono: i primi generano numeri non prevedibili mentre i secondi generano numeri che sono prevedibili se il seme può essere determinato o indovinato; tuttavia, data l'efficienza in termini di costo e tempo i PRNG sono preferiti ai TRNG. La generazione di numeri veramente casuali è complessa e i PRNG producono risultati soddisfacenti se affiancati da un TRNG per la generazione del seme (se si prendono in considerazione i casinò online, molti di essi si sono accontentati di un PRNG come sistema utilizzato nelle slot machine).
La difficoltà nell'utilizzo dei generatori TRNG è proprio la misurazione dei fenomeni microscopici che ne stanno alla base. Se prendiamo ad esempio un generatore basato sul decadimento atomico, ovviamente la misurazione del tempo tra gli intervalli e l'inserimento di queste misurazioni in un computer comportano non poche difficoltà. Pur non trattando il tema segnaliamo che le fasi principali di un generatore di questo tipo sono le seguenti: registrazione del fenomeno fisico e conversione in segnale elettrico; campionamento e conversione del segnale da analogico a digitale; adattamento del segnale alla distribuzione uniforme mediante test statistici.

Di seguito riportiamo i collegamenti a due risorse internet con le quali è possibile ottenere dei numeri veramente casuali. La prima, ossia HotBits[1], genera numeri casuali dalla temporizzazione di coppie successive di decadimenti radioattivi rilevati da un tubo Geiger-Müller interfacciato a un computer (ogni campione viene generato in risposta alla richiesta dell'utente, quindi i campioni sono unici); la seconda, ossia RANDOM.ORG, si basa sul rumore atmosferico:

- https://www.fourmilab.ch/hotbits/
- RANDOM.ORG - True Random Number Service

1.2.2 TRNG TRAMITE L'ALGORITMO DI VON NEUMANN

[1] Per maggiori approfondimenti sul funzionamento di HotBits si veda: https://www.fourmilab.ch/hotbits/how3.html

Abbiamo appena visto la difficoltà di generare un TRNG Bernoulliano attraverso l'osservazione di un procedimento fisico come il decadimento radioattivo di una sostanza. Si potrebbe pensare di utilizzare un procedimento fisico più semplice, come ad esempio il lancio di una moneta bilanciata. Tuttavia, è molto difficile stabilire se una moneta è bilanciata. Esiste però un semplice stratagemma, ideato da Von Neumann [2], che permette di utilizzare una sorgente bernoulliana non bilanciata per generare una variabile casuale bernoulliana perfettamente bilanciata (ovvero con probabilità di testa pari a 0,5).

Problema 1 Data una moneta sbilanciata, dove la probabilità che esca testa è maggiore di 0,5 e minore di 1, è possibile fare la simulazione del lancio di una moneta bilanciata?

Lanciamo ora la moneta per due volte: l'evento "esce prima testa e poi croce" lo chiamiamo 0, invece l'evento "esce prima croce e poi testa" 1. Nel caso esca la stessa faccia in entrambi i lanci ripetiamo il procedimento fino a ricadere in uno dei due casi precedenti. Definito un "*round*" come una coppia di lanci, la probabilità di generare uno 0 o un 1 è la stessa per ogni round; quindi, abbiamo simulato correttamente una moneta bilanciata (o un lancio imparziale).
Da ora chiameremo "*bit"* lo 0 o 1 generato da questo procedimento.

Esempio 1.1. Si vuole mostrare un'implementazione del procedimento sopra esposto in R. La probabilità p che esca testa e la probabilità q che esca croce sono rispettivamente:

$$p = 0{,}6 \qquad q = 1 - p = 0{,}4 \tag{1.2}$$

```
> S_VN <- function(p) {
+
+   q = 1-p
+   prob <- c(p, q)

#lanciamo la moneta per 2 volte: con la funzione sample()
estraiamo casualmente due elementi (0 è testa e 1 è croce)
date le probabilità p e q (1.2). L'esito è contenuto in "e"

+   e <- sample(c(0,1), size=2, replace=TRUE, prob=prob)

#se esce prima testa e poi croce, chiamiamo l'esito "0"

+   if (e[1] == 0 & e[2] == 1) {
+     print("0")
+   }

#se esce prima croce e poi testa, chiamiamo l'esito "1"

+   else if (e[1] == 1 & e[2] == 0) {
+     print("1")
+   }

#se esce la stessa faccia dopo i due lanci, ripetiamo il
procedimento da capo finché non ricadiamo in uno dei due casi
precedenti
```

```
+   else {
+     S_VN(p)
+   }
+ }
>
> S_VN(0.6)
[1] "1"
```

Come detto in precedenza siamo riusciti a simulare il lancio di una moneta bilanciata, da una moneta non bilanciata.

Problema 2 Sia *p* la probabilità che esca testa e $q = 1 - p$ la probabilità che esca croce. In media, quanti lanci ci vogliono per generare un *bit* usando il metodo di Von Neumann?

Se ogni round è composto da "*f*" lanci, e la probabilità di generare un *bit* in ogni round è "e" *(*$1 - e$ è la probabilità di non generare un *bit*), allora il numero atteso totale di lanci *"t"* può essere calcolato tramite l'equazione (1.3).
Se dal primo round generiamo un *bit*, usiamo esattamente *f* lanci e t è uguale a ef. Nel caso opposto, il numero atteso di lanci restanti è ancora *t* (dato che sarebbe come dover riiniziare da capo) ma dobbiamo anche tenere conto dei due lanci effettuati ($f + t$). Quindi, *t* è composto dalla somma dall'esito positivo ef e quello negativo $(1 - e)(f + t)$:

$$t = ef + (1 - e)(f + t) = \frac{f}{e} \qquad (1.3)$$

Usare la strategia Von Neumann richiede due lanci per round. Sia uno 0 che un 1 sono generati con probabilità pq, quindi la probabilità che in un round si generi con successo un *bit* è $2pq$. Possiamo ora riscrivere la formula (1.3) come:

$$t = \frac{f}{e} = \frac{2}{2pq} = \frac{1}{pq} \qquad (1.4)$$

Esempio 1.2. Si vuole conoscere quanti lanci ci vogliono per generare un bit usando il metodo Von Neumann, sapendo che $p = \frac{2}{3}$.
Risolviamo l'esercizio creando una funzione in R che segua il metodo sopra descritto:

```
> S_VN_2 <- function(p) {
+   q = 1-p
+   t = 1/(p*q)          #usiamo la formula (1.4)
+   t
+ }
> S_VN_2 (2/3)
[1] 4.5
```

Secondo il metodo Von Neumann, ci vogliono in media 4,5 lanci per poter generare un bit.

1.2.3. PRNG

Per PRNG si intendono quei generatori costituiti da un algoritmo deterministico che produce una sequenza di numeri. Per algoritmo deterministico intendiamo un algoritmo che è puramente determinato dai suoi input, in cui nessuna casualità è coinvolta nel modello;

pertanto, gli algoritmi deterministici produrranno sempre lo stesso risultato dati gli stessi input. I generatori PRNG producono una sequenza di numeri in maniera che siano all'*apparenza* casuali. Il generatore di numeri pseudocasuali produce valori che, apparentemente, soddisfano le condizioni a) e b) in tema RNG (vedi sopra).
Prima caratteristica di questo processo è la *ricorsività* dei numeri in una data sequenza. Sono infatti i precedenti *k* numeri (spesso anche il singolo precedente numero) a determinare il successivo:

$$x_i = f(x_{i-1}, \dots, x_{i-k}) \qquad (1.5)$$

il valore iniziale x_0 prende il nome di *seme*.

Dato che l'insieme dei numeri direttamente rappresentabili dal computer è finito ad un certo punto si ripeterà un valore x_i e i valori ad esso successivi. La lunghezza della sequenza prima che inizi a ripetersi è chiamata *periodo* e in generale si preferiscono generatori con periodi lunghi. Bisogna ora soffermarsi su altre due caratteristiche: la *replicabilità* e la *casualità*.

Per replicabilità si intende che la generazione di una sequenza di numeri deve essere replicabile, dati gli stessi input, deve preservare le stesse condizioni nelle applicazioni successive. Questo aspetto è molto importante dato che permette di effettuare l'attività di testing. La casualità, invece, è la condizione più importante e più complessa da definire. I generatori devono sottostare a varie proprietà teoriche per garantire la apparente casualità dei valori generati; i generatori con buone proprietà teoriche sono successivamente sottoposte a test statistici per verificare l'effettiva casualità del risultato (in particolare si può usare un *test chi quadrato di adattamento*).

La domanda che sta alla base alla generazione di numeri casuali è: quando un processo è realmente ''*casuale*''? Sotto una veste teorica un processo è definibile ''casuale'' se la probabilità condizionata nota dell'evento successivo, date le informazioni storiche, non è differente dalla probabilità incondizionata conosciuta. I PRNG sono algoritmi che, data la loro natura deterministica, generano serie di numeri che non soddisfano completamente la proprietà di casualità; infatti, le loro proprietà si avvicinano solo a quelle ideali delle sequenze casuali perfette.
È possibile riassumere quanto detto citando una celebre frase del matematico John von Neumann:

''Chiunque consideri metodi aritmetici per produrre numeri casuali sta, ovviamente, commettendo un peccato''

Per quanto riguarda i TRNG, si può affermare che essi generano veri numeri casuali dato che, per loro natura, sfruttano determinate proprietà fisiche per la generazione di tali numeri (come il decadimento atomico).

1.3. LINEAR CONGRUENTIAL GENERATORS (LCG)

Un algoritmo per la generazione di numeri pseudocasuali è il *Linear Congruential Generator* (LCG) proposto per la prima volta dal matematico D. H. Lehmer nel 1948 [3]. Si tratta di un generatore dove ogni singolo numero determina il suo successore mediante una semplice funzione lineare seguita da una riduzione modulare.

È necessario ora definire due concetti, ovvero quello di *equivalenza in modulo m* di due numeri *a* e *b* e quello di *radice primitiva*.

Definizione 1.3. Dato un numero intero e positivo *n*, si dice che due numeri *a* e *b* sono congrui tra loro in modulo *n* se, divisi per *n*, danno lo stesso resto. In equazione abbiamo:

$$a \equiv b \, mod \, n \tag{1.6}$$

Definizione 1.4. Si chiama "*radice primitiva*" di un intero *n* ogni numero *r* primo rispetto a *n*, tale che $r^k \, mod \, n$ assume ϕ (n) valori distinti se *k* varia da 1 a n – 1; se *n* è primo, $r^k \, mod \, n$ assume tutti i valori da 1 a n – 1.

La Tabella Tabella 1.1riporta i numeri naturali inferiori a 50 per i quali *n* è radice primitiva, per *n* da 1 a 15. Come vedremo nell'**esempio 1.3.** $n = 7$ non è radice primitiva di 31 a differenza di $n = 3$.

n	Numeri naturali per cui *n* è radice primitiva
1	2
2	3, 5, 9, 11, 13, 19, 25, 27, 29, 37
3	2, 4, 5, 7, 10, 14, 17, 19, 25, 29, 31, 34, 38, 43, 49, 50
5	3, 6, 7, 9, 14, 17, 18, 23, 27, 34, 37, 43, 46, 47, 49
6	11, 13, 17, 41
7	2, 4, 5, 10, 11, 13, 17, 22, 23, 26, 34, 41, 46
8	3, 5, 11, 25, 29
10	7, 17, 19, 23, 29, 47, 49
11	2, 3, 4, 6, 9, 13, 17, 18, 23, 26, 27, 29, 31, 34, 41, 46, 47
12	5, 7, 17, 25, 31, 41, 43, 49
13	2, 5, 10, 11, 19, 22, 25, 31, 37, 38, 41, 47, 50
14	3, 9, 17, 19, 23, 27, 29
15	2, 4, 13, 19, 23, 26, 29, 37, 38, 41, 46, 47

Tabella 1.1.: numeri inferiori a 50 per i quali n è radice primitiva, da 1 a 15.
Fonte: Vincenzo Librandi e T.D. Noe, The Online Encyclopedia of Integer Sequences, http://oeis.org

Il generatore è dato dalla formula ricorsiva:

$$x_i \equiv (ax_{i-1} + c) mod \, m, \qquad \text{con} \qquad 0 \leq x_i < m \tag{1.7}$$

dove *a* è chiamato il *"moltiplicatore"*, *c* l'*"incremento"* e *m* il *"modulo"* del generatore. Essendo x_i determinato dal suo precedente x_{i-1} e poiché ci sono solo *m* possibili valori diversi delle x_s, il periodo massimo del generatore LGC è *m*. Se vengono generati tutti i numeri tra 0 e $m - 1$ si ha il cosiddetto *periodo completo* (o *pieno*). Le condizioni affinché un periodo sia pieno sono collegate alla scelta dei parametri *a*, *c* ed *m*. Dato un *m* primo (per numero primo intendiamo un numero naturale maggiore di 1 e divisibile solo per 1 o sé stesso), si ha un periodo completo per ogni $x_0 \neq 0$ se:

- $a^{m-1} - 1$ è un multiplo di *m*;
- $a^j - 1$ non è un multiplo di *m* per $j = 1, \ldots, m - 2$.

Un numero *a* che soddisfa tali proprietà è detto una *radice primitiva* (vedi **definizione 1.4.**) *di m*. La sequenza di numeri generata è:

$$x_o, ax_0, a^2x_0, a^3x_0, \dots \bmod m \qquad (1.8)$$

Pertanto, la sequenza di numeri ritorna al valore iniziale x_0 non appena si arriva al più piccolo *k* per cui $a^k x_0 \; mod \; m \; = \; x_0$, cioè al più piccolo *k* per cui $a^k \; mod \; m = 1$, cioè il più piccolo *k* per cui $a^k - 1$ è un multiplo di *m*, e quindi si ha un periodo completo esattamente quando *a* è una radice primitiva di m.

Per quanto riguarda l'incremento *c* il caso originariamente preso in considerazione da Lehmer è quello con $c = 0$. In questo caso, il generatore prende il nome di ''*generatore Park-Miller*'' o ''*moltiplicativo*'':

$$x_i \equiv ax_{i-1} mod \; m, \qquad \text{con} \qquad 0 < x_i < m \qquad (1.9)$$

Con $c = 0$ escludiamo la possibilità che nella sequenza compaia il numero zero, che farebbe azzerare tutti i valori successivi. Il periodo non potrà quindi essere esattamente uguale a *m*; infatti, nel generatore moltiplicativo il periodo massimo è $m - 1$. Il periodo di un generatore Park-Miller con parametri *a* ed *m* dipende dal più piccolo valore positivo di *k* per il quale è valido:

$$a^k \equiv 1 \; mod \; m \qquad (1.10)$$

Questo perché quando il rapporto è soddisfatto, la sequenza si ripete.
Si ha che $a, m \in \mathrm{N}$ e il valore iniziale x_0 è un intero compreso tra 1 e $m - 1$ fornito dall'utente. Una sequenza risultante dalla formula ricorsiva è chiamata sequenza di Lehmer. Ogni x_i generato, è convertito nell'intervallo unitario (0,1) mediante la divisione con *m*:

$$u_i \equiv \frac{x_i}{m} \qquad (1.11)$$

Se *a* e *m* sono opportunamente scelti, allora i u_{is} sembreranno effettivamente casualmente e uniformemente distribuiti in un intervallo tra 0 e 1.

Di seguito si mostra un esempio applicativo della funzione ricorsiva del generatore LGC:

Esempio 1.3. sostituiamo i parametri m=31 e a=7 nella equazione (1.9), ottenendo un generatore moltiplicativo di equazione:

$$x_i \equiv 7x_{i-1} mod \; 31$$

Dato il seme x_0=19, la sequenza generata sarà la seguente

9, 1, 7, 18, 2, 14, 5, 4, 28, 10, 8, 25, 20, 16, 19, 9, 1, 7, 18, 2, (…)

La sequenza ha un periodo $k = 15$ e successivamente inizierà a ripetersi (19 è l'ultimo numero prima che la sequenza inizi a ripersi); prendendo ora l'equazione (1.10), e sostituendo i valori fino ad ora ottenuti abbiamo che:

$$7^{15} \equiv 1 \; mod \; 31$$

Essendo questa relazione soddisfatta, abbiamo la conferma che dopo 15 numeri la sequenza si ripeterà. Dato che $a = 7$ non soddisfa le due condizioni affinché si possa raggiungere il periodo pieno ($a^{m-1} - 1$ è un multiplo di *m*; $a^j - 1$ non è un multiplo di *m* per $j = 1, \ldots, m - 2$.), questo non viene raggiunto, *a* non è radice primitiva di 31(vedi tabella Tabella 1.1).
Ora consideriamo gli stessi valori di *m* e x_0 ma con un moltiplicatore $a = 3$

$$x_i \equiv 3 * 19 \; mod \; 31$$

9, 1, 7, 18, 2, 14, 5, 4, 28, 10, 8, 25, 20, 16, 19, 9, 1, 7, 18, 2, 14, 5, 4, 28, 10, 8, 25, 20, 16, 19, 9, 1, 7, 18, 2, 14, 5, (…)

In questo caso il periodo è di 30 numeri prima che si ripeta la sequenza. Abbiamo raggiunto il periodo massimo $m - 1 = 30$ perché 3 soddisfa le condizioni affinché si possa raggiungere il periodo completo; pertanto, 3 è radice primitiva di 31 (vedi tabella Tabella 1.1).

Un modulo *m* comunemente usato è il Mersenne prime, dove $m = 2^{31} - 1$ e per questo modulo un multiplo comune è 7^5 (16807), solitamente il modulo non supera questo valore. Nella tabella Tabella 1.2 vengono riportati delle combinazioni dei parametri *m*, *a* e *c* per gli LCG:

Modulo m	Moltiplicatore a	Incremento c
2^{32}	69069	0 o 1
2^{32}	1664525	0
$2^{31} - 1$	16807	0
$2^{31} - 1$	39373	0
$2^{31} - 1$	742938285	0
$2^{31} - 1$	950706376	0
$2^{31} - 1$	1226874159	0

Tabella 1.2.: parametri per un LCG.

Un modo per poter allungare il periodo è quello di mescolare l'output di un generatore LCG usando un altro generatore. Esamineremo questo metodo nel paragrafo successivo.

Il Gentle osserva che la semplicità e l'efficienza del LCG lo rendono uno dei generatori di numeri pseudocasuali più usati. Ma, questa semplicità del generatore porta avanti vari problemi di applicabilità. Gli LCG non dovrebbero essere utilizzati in applicazioni dove viene richiesta una alta casualità, ad esempio non sono adatti per *simulazioni Monte Carlo* poiché i numeri generati sono correlati.

1.3.1. COMBINAZIONE DI GENERATORI

Nel corso del tempo sono state proposte varie tecniche per poter aumentare l'efficienza dei generatori, un modo per poter fare ciò è quello che si basa sulla combinazione di più generatori, così da poterne creare uno che possa essere migliore di quelli che lo compongono (presi singolarmente).

La scelta di combinare uno o più generatori ha dei vantaggi, inffatti può migliorare sia il periodo che l'apparente casualità del generatore di numeri casuali. Esistono vari metodi di combinazione, nello specifico ne prenderemo in considerazione due: quello proposto da Wichmann e Hill, e quello proposto da Ecuyer.

Un modo elementare per combinare più generatori è attraverso l'addizione. Wichmann e Hill descrivono un generatore a tre sorgenti, il quale è una combinazione di LCG. Secondo il Gentle [1], si tratta di un metodo di facile programmazione e che gode di buone proprietà di casualità. Il generatore combinato consiste in tre generatori LCG con diversi moduli primi, ciascuno dei quali viene utilizzato per produrre un numero uniformemente distribuito compreso tra 0 e 1, questi vengono sommati, modulo 1, per produrre il risultato. Le tre equazioni congruenziali lineari dell'algoritmo di Wichmann-Hill (WH) sono le seguenti:

$$\begin{aligned} x_i &\equiv 171 x_{i-1} mod\ 30269 \\ y_i &\equiv 172 y_{i-1} mod\ 30307 \\ z_i &\equiv 170 z_{i-1} mod\ 30323 \end{aligned} \tag{1.12}$$

Da cui

$$u_i = \left(\frac{x_i}{30269} + \frac{y_i}{30307} + \frac{z_i}{30323}\right) \bmod 1 \tag{1.13}$$

Il seme di questo generatore è il vettore (x_0, y_0, z_0). La somma dei tre generatori produce una sequenza pseudocasuale di ordine 10^{12}.

Un altro metodo di combinazione di generatori LCG è quello proposto da Ecuyer. L'Ecuyer propone di combinare k generatori congruenziali moltiplicativi che hanno modulo primo m_j, tale che $\frac{m_j-1}{2}$ siano relativamente primi, e con moltiplicatori che producono periodi completi. Assumendo che il primo generatore sia relativamente buono e che m_1 sia abbastanza grande (si assume che sia il più grande tra gli m_j), formiamo la sequenza come

$$\begin{aligned} x_i &\equiv \sum_{j=1}^{k} (-1)^{j-1}_{xj,i}\ mod\ (m_1 - 1) u_i \\ &= \begin{cases} \dfrac{x_i}{m_i}, & x_i > 0 \\ \dfrac{(m_1 - 1)}{m_1}, & x_i = 0 \end{cases} \end{aligned} \tag{1.14}$$

Uno specifico generatore suggerito da Ecuyer è il seguente:

$$\begin{aligned} x_i &\equiv 40014 x_{i-1} mod\ 2147483563 \\ y_i &\equiv 40692 y_{i-1} mod\ 2147483399 \\ z_i &\equiv (x_i - y_i)\ mod\ 2147483563 \end{aligned} \tag{1.15}$$

e

$$u_i = 4656613 z_i * 10^{-10} \tag{1.16}$$

In questo caso il periodo è di ordine 10^{18}.

Una buona combinazione tra più generatori migliora l'apparente casualità della sequenza e aumentarne il periodo. Sarebbe possibile ottenere un periodo molto lungo con un normale generatore LCG utilizzando un modulo molto grande, ma questo potrebbe portare a problemi di overflow nel calcolatore (si tratta della situazione che si verifica quando viene superata la capacità massima del registro aritmetico di un calcolatore, cioè quando il risultato dell'operazione impostata è un numero con tante cifre da eccedere il massimo numero rappresentabile). La combinazione si usa per rendere il processo più difficile da prevedere. Deng e George (1990) e Deng et al. (1997) forniscono argomenti a favore dell'utilizzo di generatori combinati basati su densità di probabilità pressoché costanti.

Come scritto a inizio paragrafo, il generatore LCG è quello tuttora più utilizzato, ma è bene dire che ne esistono altri di uso comune i quali, per completezza, si citano di seguito:

- Generatori di Fibonacci ritardati (o lagged Fibonacci);
- Registri di traslazione a retroazione lineare;
- Algoritmo Fortuna;
- Algoritmo Blum Blum Shub.

L'algoritmo Fortuna e l'algoritmo Blum Blum Shub sono due algoritmi che prendono il nome di *generatori crittograficamente sicuri (CRNG)*, ovvero generatori con proprietà tali da essere adatti all'uso in crittografia.

1.4. GENERAZIONE DI VARIABILI CASUALI QUALSIASI

In questo paragrafo prenderemo in considerazione i metodi per la generazione di v.c. qualsiasi, metodi che hanno come obiettivo quello di convertire una sequenza di numeri casuali uniformemente distribuiti nell'intervallo (0,1) in una sequenza $\{x_i\}$ di numeri casuali distribuiti secondo una densità di probabilità *f(x).* Come vedremo, il generatore uniforme è fondamentale sia per il *metodo accept reject* sia per il *metodo dell'inversione* che presenteremo in questo paragrafo. Prima di procedere con l'esposizione, è bene fare delle considerazioni preliminari. In questi metodi svolgono un importante ruolo la funzione di densità e quella di ripartizione. Infatti, l'input del metodo del rigetto è proprio la funzione di densità *f(x)* e nel metodo dell'inversione è la funzione di ripartizione *F(x)*. Sfrutteremo inoltre una proprietà di trasformazione, in breve, ci dice che una v.a. *Y* ottenuta con una trasformazione lineare di una v.a. *X* con distribuzione uniforme in [0,1]:

$$Y = aX + b \qquad (1.17)$$

è ancora una uniforme su un altro intervallo, ovvero $Y \sim U(b\,, a+b)$.
Bisogna infine precisare quando una funzione è *invertibile* (le definizioni seguenti ci serviranno, nello specifico, per il metodo dell'inversione).

Definizione 1.5. Una funzione $f\colon X \to Y$ si dice invertibile se e solo se esiste una funzione che associa a ciascun elemento dell'immagine di *f* la sua controimmagine. Tale funzione prende il nome di *funzione inversa* di *f* e viene solitamente indicata con la notazione f^{-1}.

Se una funzione è invertibile, allora è per definizione *biiettiva*, ovvero è sia *suriettiva* che *iniettiva.*

Definizione 1.6. Una funzione $f: X \rightarrow Y$ si dice *iniettiva* se ogni coppia di elementi distinti x_1 e x_2 del dominio hanno immagini distinte $f(x_1)$ e $f(x_2)$ nel codominio, ossia:

$$se\ x_1 \neq x_2 \Rightarrow f(x_1) \neq f(x_2) \tag{1.18}$$

Definizione 1.7. Una funzione $f: X \rightarrow Y$ è detta *suriettiva* se ogni elemento del codominio è immagine di almeno un elemento del dominio, ovvero se:

$$\forall\, y \in Y, \exists\, x \in X \mid f(x) = y \tag{1.19}$$

In generale una funzione iniettiva $f: X \rightarrow Y$ non è invertibile, dato che dovrebbe essere anche suriettiva ma, effettuando una restrizione del codominio all'immagine si ottiene che la funzione f e la sua inversa f^{-1} operino tra gli stessi insiemi. Così facendo, se una funzione invertibile f è iniettiva, anche la sua inversa f^{-1} lo è.

Teorema 1.1. Sia $f: X \rightarrow Y$ una funzione iniettiva, allora risulta definita una funzione $h: f(X) \rightarrow X, y \mapsto x$ dove $\forall\, y \in f(x), \exists!\ x \mid h(y) = x$ oppure tale che $f(x) = y$. La funzione h è proprio la funzione inversa di f, $h := f^{-1}$.

1.4.1. METODO DELL'INVERSIONE

Vogliamo generare una variabile aleatoria X con funzione di ripartizione

$$F_X(x) = Prob\{X \leq x\} \tag{1.20}$$

Se tale funzione è continua, monotona e strettamente crescente nell'intervallo [0,1] allora si tratta di una funzione invertibile, pertanto è possibile applicare il metodo dell'inversione seguendo due step:

1. Si effettua l'estrazione di un numero casuale u_i da $U \sim Unif[0,1]$;
2. Ricaviamo x_i da u_i tramite l'equazione $x_i = F_X^{-1}(u_i)$.

Graficamente:

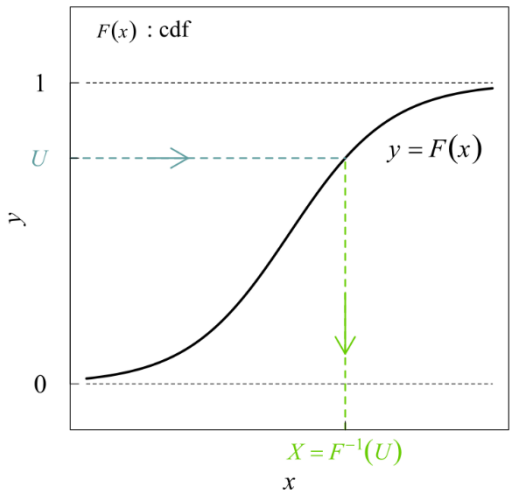

Figura 1.1.: rappresentazione grafica metodo dell'inversione.
fonte: https://it.wikipedia.org/wiki/Metodo_dell%27inversione

Alla base del metodo dell'inversione vi è il *teorema della trasformazione integrale di probabilità* [4]:

Teorema 1.2. Se *X* è una variabile casuale con funzione di ripartizione continua $F_X(X)$, allora $U = F_X(X)$ è distribuita uniformemente nell'intervallo (0,1). Viceversa, se *U* è distribuito uniformemente nell'intervallo (0,1), allora $X = F_X^{-1}(U)$ ha funzione di ripartizione $F_X(\cdot)$.

Dimostrazione:

$$P[U \leq u] = P[F_X(X) \leq u] = P[X \leq F_X^{-1}(u)] = F_X\left(F_X^{-1}(u)\right) = u \quad per 0 < u < 1.$$
$$Viceversa, \quad P[X \leq x] = P[F_X^{-1}(U) \leq x] = P[U \leq F_X(X)] = F_X(X).$$

È solo in virtù del teorema che il metodo dell'inversione è sensato. Infatti, il metodo sfrutta la seconda parte del teorema (si veda la figura Figura 1.1): se *U* è una v. c. con distribuzione uniforme nell'intervallo (0,1), allora $X = F_X^{-1}(U)$ è una v. c. avente funzione di ripartizione $F_X(\cdot)$. Riassumendo quanto detto, per ottenere un valore *x* di una v. c. *X*, si genera un valore *u* di una v. c. *U*, si calcola $F_X^{-1}(u)$, e lo si pone uguale a *x*.

Nel caso di una v.c. discreta non abbiamo una funzione di ripartizione invertibile. Quando ci troviamo a lavorare con v.c. discrete bisogna precisare che non è detto che ad una data *p* corrisponda un solo valore x tale che $F(x) = p$ (vedi sopra quando detto per le condizioni affinché una funzione sia invertibile). Tuttavia, il metodo dell'inversione può essere esteso anche al caso delle variabili discrete. Consideriamo ora in particolare una variabile aleatoria *X* finita, ovvero che può assumere i valori x_i, con $i = 1, \dots, N$. Supponiamo ora di riportare in successione su un asse, a partire dall'origine, N segmenti di lunghezza pari alle probabilità p_i. Se ora estraiamo un numero casuale con distribuzione uniforme in (0,1), esso cadrà nel segmento i-simo con probabilità pari alla sua lunghezza, ossia p_i. La funzione di ripartizione cumulata della variabile discreta è la seguente:

$$P_0 = 0, \qquad P_i = \sum_{k=1}^{i} p_k \ \ con\ i = 1, \dots, N. \tag{1.21}$$

Ora generiamo un numero casuale U_i dalla distribuzione uniforme in [0,1] e si determina l'intero *i* tale che:

$$P_{i-1} \leq U_i < P_i \tag{1.22}$$

Graficamente (vedi figura Figura 1.2), riportando le probabilità cumulative in funzione del rispettivo indice *i*, essa assume una forma a gradini (o scaletta). Estratto U_i, si traccia sul grafico la retta $y = U_i$ (parallela alle ascisse) e al medesimo ci sarà la corresponsione di un valore x_i nelle ascisse (si tratta del valore campionario per la variabile aleatoria *X*).

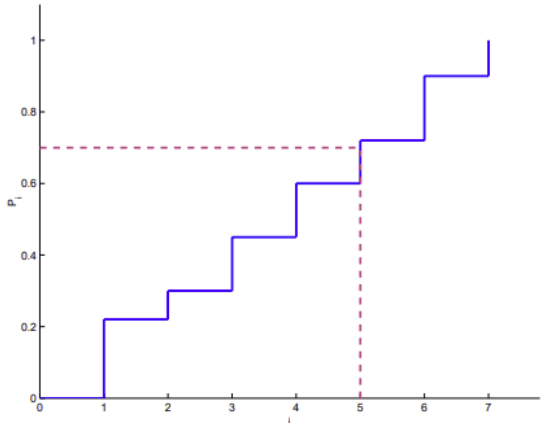

Figura 1.2.: rappresentazione grafica metodo dell'inversione con v.a. discrete.
fonte: Leonardo Angelini, Tecniche Monte Carlo, Università degli Studi di Bari.

Esempio 1.4. Si vuole mostrare un'implementazione del metodo dell'inversione caso v.c. discreta in R. Per questo esempio è stata utilizzata una v.c. discreta $X = 0{,}1{,}2$ con

$$f(0) = 0{,}25, f(1) = 0{,}25, f(2) = 0{,}5$$

```
#salviamo questi valori in R con i comandi

> X = c(0,1,2)
> p = c(0.25,0.25,0.50)

#si creano poi dei vettori vuoti con la funzione c(), che
successivamente useremo per inserire i valori campionati

> x_1=c()
> x_2=c()
> x_3=c()

#grazie alla funzione cumsum() creiamo un vettore i cui
elementi sono le somme cumulative di p

> cs=cumsum(p)

#con la funzione runif() generiamo le sequenze di numeri
distribuiti uniformemente nell'intervallo [0,1], nel nostro
esempio si è scelto di generare 25 numeri e con l'ausilio di
sort() li disponiamo in ordine crescente

> u=sort(runif(25,0,1))
>
> count=1

#è ora che l'algoritmo dell'inversione inizia a lavorare. Con
while()apriamo il ciclo che lavorerà in loop finché il
campione che stiamo analizzando risulta essere minore o uguale
di 25. Grazie alla struttura di controllo if siamo in grado di
impostare le diverse condizioni

> while (count<=25) {
```

```
+   test_u = u[count]

#se il valore generato è minore della probabilità cumulata in
0 (ossia u < 0.25), allora inseriamo nel vettore x_1 il valore u

+   if (test_u < cs[1]) {
+     x_1=rbind(x_1, u[count]);
+     count = count + 1;
+   }

#se il valore generato è compreso tra la probabilità cumulata
in 0 e 1 (ossia 0.25 < u ≤ 0.50), allora inseriamo nel vettore x_2
il valore u

+   if (test_u >= cs[1] & test_u < cs[2]) {
+       x_2=rbind(x_2, u[count]);
+       count = count + 1;
+   }

#in tutti gli altri casi (ossia 0.50 < u), allora inseriamo nel
vettore x_3 il valore u

+   if (test_u >= cs[2]) {
+     x_3=rbind(x_3, u[count]);
+     count = count + 1;
+   }
+ }
```

Una volta conclusa la fase di analisi dei dati, abbiamo ottenuto i tre vettori x_1, x_2 e x_3 contenenti i valori rispettanti la condizione (1.22) (vedi tabella Tabella 1.3). Si è scelto di separare i valori in tre diversi vettori per semplificare i calcoli e l'elaborazione finale.

	X_1	X_2	X_3
1	0.07897091	0.3005726	0.5128243
2	0.12643401	0.3195855	0.5579551
3	0.13292511	0.3683340	0.5859620
4	0.18193406	0.3811250	0.6300647
5	0.18930645	0.3961297	0.6653596
6		0.4044993	0.6678732
7		0.4517005	0.6770537
8		0.4535886	0.7134051
9			0.7828385
10			0.8885890
11			0.8917718
12			0.9423602

Tabella 1.3.: valori di x_1, x_2, x_3.

1.4.2. METODO DEL RIGETTO (ACCEPT-REJECT ALGORITHM)

L'accept-reject algorithm è un algoritmo per generare campioni casuali da una distribuzione di probabilità arbitraria usando come ingredienti campioni casuali da una distribuzione correlata e la distribuzione uniforme. Elemento centrale del metodo è la densità desiderata, chiamata anche densità target o obiettivo o desiderata. Il concetto alla base di questo algoritmo è molto semplice: dato un insieme di numeri casuali, ne vogliamo eliminare una parte così che i rimanenti siano distribuiti secondo la densità target. Più precisamente, data una variabile con funzione di densità *f(x)* (ossia la distribuzione desiderata) definita sull'intervallo [a,b] definiamo il massimo di $f(x)$ in [a,b]:

$$c = \max[f(x)] \geq f(x) \forall x \in [a, b] \tag{1.23}$$

L'idea del metodo è accettare i numeri casuali generati uniformemente nel rettangolo con lati *c* e *(b-a)* e che allo stesso tempo siano sottostanti alla curva f(x). Generiamo quindi due sequenze casuali uniformi distribuiti in [0,1], che chiameremo u_1 e u_2. Da questo momento siamo in grado di generare altre sequenze *x* e *y* da u_i e $\bar{u}_i$ sulla base delle seguenti formule:

1. Dagli u_i generiamo una sequenza x_i di numeri distribuiti uniformemente nell'intervallo [a,b]:

$$x_i = (b - a)u_i + a \tag{1.24}$$

2. Dagli $\bar{u}_i$ generiamo una sequenza y_i di numeri distribuiti uniformemente nell'intervallo [0,c]:

$$y_i = c\,\bar{u}_i \tag{1.25}$$

3. Accettiamo gli x_i solo se:

$$y_i < c \tag{1.26}$$

Ad ogni coppia di valori (u_i, $\bar{u}_i$) corrisponderà una coppia (x_i, y_i) appartenente al rettangolo $[a, b]x[0, c]$. Se la coppia (x_i, y_i) non è concorde con quanto detto in precedenza allora la procedura viene ripetuta finché non viene trovata una nuova coppia che rispetti quanto richiesto. La figura Figura 1.3 rappresenta graficamente il procedimento del metodo del rigetto: i punti rossi rappresentano quelli scartati e quelli blu quelli accettati (si trovano sotto la curva).

La sequenza di valori ottenuta x'_i risulta distribuita secondo la legge di distribuzione di *f(x)* differendo da essa soltanto per una costante di normalizzazione. Questo metodo permette di sorvolare eventuali problemi analitici, ma presenta diversi svantaggi sotto l'aspetto del tempo di computazione. In generale l'efficienza del metodo è legata alla quantità di rigetti; è intuibile che l'efficienza del metodo viene aumentata quanto più l'area di *f(x)* copre quasi tutto il rettangolo $[a, b]x[o, c]$ cosicché il numero di rigetti diminuisca.

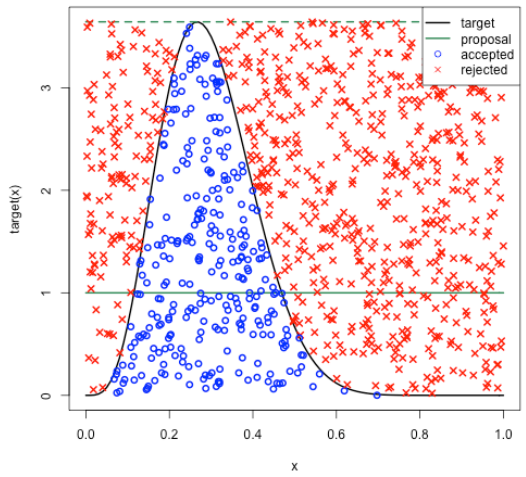

Figura 1.3.: rappresentazione grafica del metodo del rigetto.
Fonte: https://www.jarad.me/teaching/2013/10/03/rejection-sampling

Esempio 1.5. Si vuole mostrare un'implementazione del metodo del rigetto in R. Per questo esempio è stata utilizzata come densità target la funzione $f(x) = 2x$ su [0,1] e come distribuzione nota quella uniforme nell'intervallo [0,1].

```
#come prima cosa, con le funzioni curve() e abline() creiamo
il grafico (vedi figura Figura 1.4) della funzione f(x) e di c =
max[f(x)] (che nel nostro esempio è uguale a 2 dato che
l'intervallo è [0;1]):

> curve(2*x, 0, 1)
> abline(a = 2, b = 0)
```

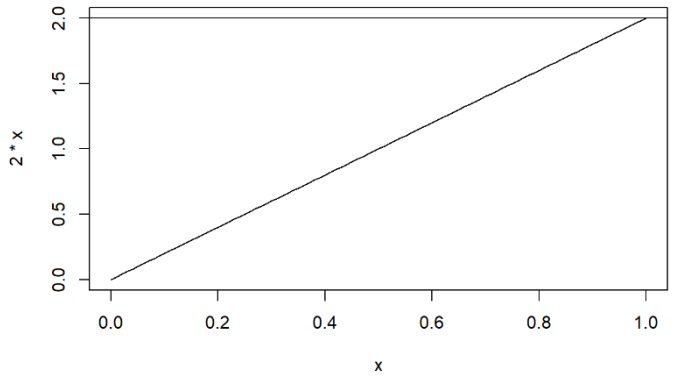

Figura 1.4.: grafico di f(x)=2x e di c=2.

```
#con la funzione runif() generiamo le sequenze di numeri
distribuiti uniformemente nell'intervallo [0,1], nel nostro
esempio si è scelto di generare 5000 numeri

> X = runif(5000, 0, 1)
> U = runif(5000, 0, 1)
>
> f_d <- function(x) {
```

```
+   new_x = 2*x
+   return(new_x)
+ }
>
> count = 1

#Creiamo accept, ovvero un vettore che conterrà i valori
accettati

> accept = c()

#è ora che l'algoritmo del rigetto inizia a lavorare. Con
while()apriamo il ciclo che lavorerà in loop finché il
campione che stiamo analizzando risulta essere minore o uguale
di 5000 e finché abbiamo meno di 1000 campioni accettati (i
valori sono stati scelti a titolo di esempio, è possibile
modificarli a propria discrezione)

> while(count <= 5000 & length(accept) < 1000){
+   test_u = U[count]
+   test_x = f_d(X[count])/(2*dunif(X[count],0,1))

#se il valore generato rientra nella distribuzione target,
allora la accettiamo e inseriamo in accept

+   if (test_u <= test_x){
+     accept = rbind(accept, X[count])
+     count = count + 1
+   }
+   count = count + 1
+ }

#una volta conclusa la fase di analisi dei dati, abbiamo che
il vettore precedentemente creatore conterrà solo i valori
accettati. Con la funzione hist() creiamo l'istogramma (vedi
figura Figura 1.5) degli accettati che andiamo a comparare con la
funzione f(x):

> hist(accept,prob=T)
> curve(2*x,add=T)
```

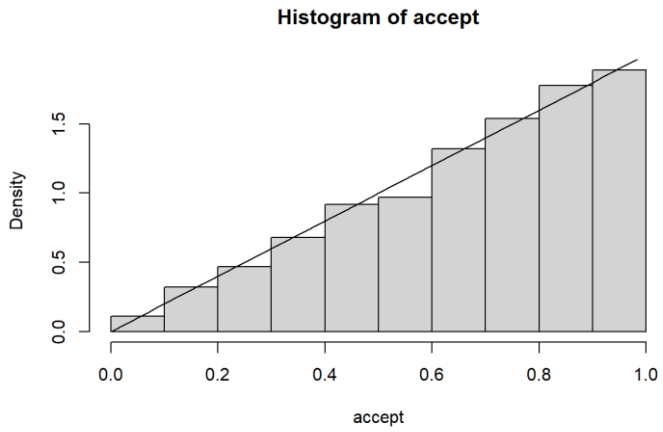

Figura 1.5.: confronto campioni accettati con la funzione target f(x)=2x.

1.4.3. METODO MISTO

Si tratta di un metodo misto, ossia che utilizza sia il metodo del rigetto che quello dell'inversione, che risulta utile quando il metodo del rigetto è inefficiente e la funzione di densità di probabilità è di difficile integrazione e inversione. Prendiamo in considerazione il caso in cui vogliamo generare una sequenza di numeri per una variabile aleatoria che segue la funzione di densità $f(x)$ con $x \in [a,b]$ per la quale si verificano le condizioni seguenti:

1. La primitiva non è nota;
2. Il metodo del rigetto è poco efficiente.

Data la funzione $f(x)$:

$$f(x) = g(x)z(x) \tag{1.27}$$

Con $z(x)$ uniformemente distribuita in [0,1] e funzione monotona di *x*, normalizzata e di cui la primitiva è nota ($Z(x)$). Dati questi presupposti possiamo ora scrivere:

$$f(x)dx = g(x)z(x)dx = g(x)dZ(x) \tag{1.28}$$

Con

$$Z(x) = \int_a^x z(t)dt \in [0,1] \tag{1.29}$$

Per risolvere il nostro problema basterà ora seguire e ripetere n volte quanto segue:

1. Estrarre un numero casuale U_i uniformemente distribuito in [0,1];
2. Trovare il valore di x_i tramite l'equazione

$$x_i = H^{-1}(U_i) \tag{1.30}$$

4. Generare un secondo numero casuale $\overline{U}_i$ uniformemente distribuito in [0,1] ed accettare x_i solo se si rispetta

$$g_{MAX}(\overline{U}_i) \leq g(x_i) \tag{1.31}$$

   dove g_{MAX} risulta essere il valore massimo di g(x) in [a,b].

La sequenza di numeri x_i viene in un primo momento generati secondo la distribuzione $z(x)$ mediante il metodo dell'inversione e successivamente filtrati tramite il metodo del rigetto con la funzione $g(x)$.

1.5. GENERAZIONE DI VARIABILI CASUALI IN R

R è un ambiente di sviluppo specifico per l'analisi statistica dei dati. In R è possibile generare numeri pseudocasuali che seguano una determinata distribuzione mediante delle funzioni apposite. Le funzioni hanno un nome intuitivo formato da

$$r + iniziale\ della\ distribuzione + ()$$

Ad esempio per generare numeri da una normale abbiamo la funzione `rnorm(n, mean, sd)` dove all'interno delle parentesi è possibile specificare la dimensione del campione *n*, la media del campione *mean* (di default `mean=0`) e la deviazione standard *sd* (di default `sd=1`). Nella tabella Tabella 1.4 vengono riportati alcuni esempi di distribuzioni note con le correlate funzioni R.

Distribuzione	Funzione R
Uniforme	`runif()`
Normale	`rnorm()`
Binomiale	`rbin()`
Geometrica	`rgeom()`
Ipergeometrica	`rhyper()`
Binomiale negativa	`rnbinom()`
Poisson	`rpois()`

Tabella 1.4.: alcune funzioni in R per la generazioni di numeri pseudocasuali.

Prendendo ora in considerazione la funzione `runif()`, esso genera numeri casuali seguendo una distribuzione Uniforme all'interno di un intervallo [a,b] (i valori di default sono `min=0` e `max=1`). Per quanto riguarda il seme di generazione, esso è impostato automaticamente dal programma, ma se si vuole inserire un seme specifico è possibile farlo tramite la funzione `set.seed()` (fare ciò consente al compilatore di garantire la replicabilità degli esperimenti).

Di seguito, a titolo di esempio, si riportano due applicazioni in R della funzione `runif()`.

Esempio 1.6.

```
#generiamo 5 numeri casuali distribuiti uniformemente
nell'intervallo [0,1]

> runif(5)
[1] 0.4032305 0.2469070 0.2020771 0.5227551 0.9912532

#impostiamo un seme a nostra scelta

> set.seed(8)
> runif(5,1,8)
[1] 4.264067 2.454763 6.597606 5.563099 3.250564

#inserendo lo stesso seme si avrà lo stesso risultato

> set.seed(8)
> runif(5,1,8)
[1] 4.264067 2.454763 6.597606 5.563099 3.250564
```

Per la generazione di numeri pseudocasuali, di default R utilizza l'algoritmo Mersenne-Twister. È possibile, tramite la funzione `RNGkind()` impostare un generatore diverso da quello di default, scegliendo uno tra gli algoritmi: Super-Duper; Wichmann-Hill; Marsaglia - Multicarry; L'Ecuyer-CMRG.

CAPITOLO 2 - IL METODO MONTE CARLO

2.1. INTRODUZIONE AL METODO: CENNI STORICI E APPLICAZIONI

Il metodo Monte Carlo [5] affonda le sue radici nella Seconda Guerra Mondiale grazie all'avvento di due importanti eventi: la creazione del primo computer elettronico (ENIAC) e la costruzione della bomba atomica. Gli ideatori sono Stanislaw Ulam, Enrico Fermi e John Von Neumann. In particolare, il matematico Stanislaw Ulam fu colui che perfezionò il metodo Monte Carlo subito dopo la fine della Seconda guerra mondiale. A dare il nome al metodo fu Nicholas Constantine Metropolis, facendo riferimento al noto casinò situato a Monte Carlo nel Principato di Monaco. Una credenza comune è che la scelta del nome sia casuale o incentrata nella sola affinità del metodo con il gioco d'azzardo, ma in realtà ha una storia interessante. La scelta è legata ad un parente stretto di Ulam, lo zio, che all'epoca era conosciuto per la sua dipendenza verso il gioco d'azzardo. Lo zio era solito prendere in prestito denaro per andare a Monte Carlo ed amava così tanto il gioco che anche dopo che il nipote venne sottoposto a un intervento preferì andare al casinò anziché visitarlo in ospedale. È proprio in nome dello zio che oggi il metodo è conosciuto come "Monte carlo".
Il metodo Monte Carlo è uno strumento di notevole importanza in moltissimi ambiti, dalla fisica (metodi quantistici), biologia (sistemi biologici), finanza (computazione volare di una impresa) ad altri settori anche non puramente scientifici quali progettazione grafica o creazione di videogiochi. Il suo sempre maggiore utilizzo è conseguente alla creazione dei primi computer. Oggigiorno, con il termine Monte Carlo, ci si riferisce ad un qualunque metodo che utilizzi sequenze di numeri casuali per condurre simulazione statistiche. Il funzionamento della simulazione Monte Carlo si basa sull'osservazione ripetuta di un fenomeno con l'obiettivo di approssimare le caratteristiche del fenomeno stesso. Gli esempi più pratici del metodo si basano sulle scienze matematiche e di seguito vedremo il suo uso nel calcolo di un integrale definito.

2.1.1. IL CALCOLO DI UN INTEGRALE

Nella sua forma più semplice, la *simulazione di Monte Carlo* serve per la approssimazione di un integrale definito su un dominio D chiuso

$$\theta = \int_D f(x)dx \qquad (2.1)$$

Questa operazione può essere svolta mediante l'uso di *metodi diretti* quali l'integrazione per parti, serie e sostituzione, ma quando ci si trova davanti a un calcolo più complesso risulta troppo laborioso utilizzare questi metodi ed è necessario ricorrere all'uso di un calcolatore e la simulazione Monte Carlo offre una semplice soluzione. Si sottolinea il fatto che a differenza dei metodi diretti che sono esatti il metodo Monte Carlo fornirà un'*approssimazione* dell'integrale e non il suo valore preciso.
Per l'esposizione di questo argomento prenderemo in considerazione una notazione semplificata: sia dato un integrale θ della funzione *f(x)* limitata sull'intervallo [a,b]:

$$\theta = \int_a^b f(x)dx \qquad (2.2)$$

Sia *M* il massimo della funzione *f(x)* sull'intervallo [a,b]:

$$M = max[f(x)] \tag{2.3}$$

Si delimita un rettangolo di base [a,b] e altezza *M* tale che l'area sottesa dal grafico *f(x)* sia contenuta nell'area del rettangolo (vedi figura Figura 2.1).

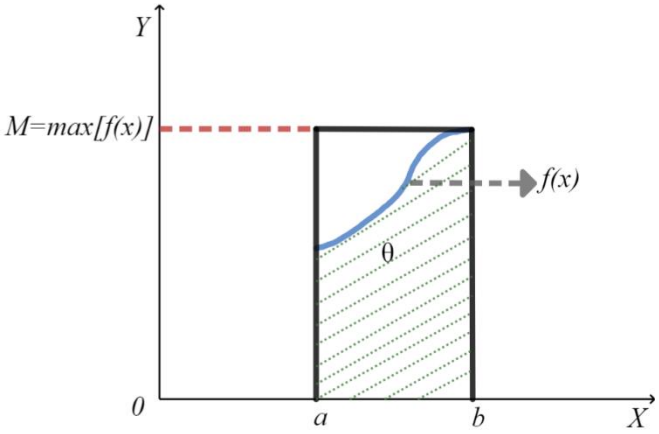

Figura 2.1.: rappresentazione grafica rettangolo [a,b]x[0,M] e f(x).

Siano *x* e *y* due numeri casuali uniformi tali che:

- $x \in [a, b]$
- $y \in [0, M]$

Consideriamo ora un punto di coordinate $(x; y)$. Ciò che ci interessa è calcolare la probabilità che tale punto si trovi sotto il grafico della funzione *f(x)* (vedi figura Figura 2.1), ossia la probabilità che $y \leq f(x)$. Tale probabilità è coincidente con:

$$p\big(y \leq f(x)\big) = \frac{\theta}{A} = \frac{\theta}{(b-a) \cdot M} \tag{2.4}$$

Ovvero con il rapporto tra l'area sottesa dalla funzione *f(x)* e l'area del rettangolo con base [a,b] e altezza *M*. Essendo il nostro obiettivo l'approssimazione di θ, l'unico termine non noto risulta essere $p(y \leq f(x))$. Il metodo Monte Carlo ci permette di stimare tale probabilità.

Si generino *n* coppie di numeri casuali $(x_i;\ y_i)$, con:

- $i = 1, \dots, N$;
- $x_i \in [a; b]$;
- $y_i \in [0; M]$;

Tra le coppie $(x_i;\ y_i)$ generate, ci sarà un certo numero di esse tali che $y_i \leq f(x_i)$ che denominiamo U_n. Possiamo dunque approssimare la quantità $p(y \leq f(x))$ come:

$$p \cong \frac{U_n}{n} \tag{2.5}$$

E ora possiamo stimare l'integrale θ come:

$$I \cong p(b-a)M \cong \frac{U_n}{n}(b-a)M \qquad (2.6)$$

I è l'approssimazione Monte carlo di θ.

Si vuole ora far notare come questa approssimazione rispecchia quanto visto nel paragrafo 1.4.2. dedicato al metodo accept-reject. Infatti, ripercorrendo brevemente i passaggi più importanti dell'approssimazione possiamo notare come M (2.3)non è altro che *c*, il massimo della funzione f(x) (1.23), la generazione delle coppie $(x_i;\ y_i)$ segue le formule (1.24) e (1.25) e che p (2.5) segue la regola (1.26). Dato questo parallelismo è possibile sviluppare l'approssimazione dell'integrale seguendo quando già mostrato nell'**esempio 1.5.**.

2.2. STIMATORI E VARIANZA DEGLI STIMATORI DI MONTE CARLO

Con il Monte Carlo, come già detto in precedenza, arriviamo ad avere una approssimazione del valore, una stima, e non il valore effettivo; pertanto, le stime risultanti dalle procedure di Monte Carlo hanno errori di campionamento associati. La variabilità dei dati casuali risulta in un errore sperimentale. Come in ogni problema di stima statistica, una stima va accompagnata da una stima della sua varianza. Un modo efficace di fare questo è dando la deviazione standard del campione. Quando vengono riportati un certo numero di risultati e le deviazioni standard variano da uno all'altro, un buon modo di presentare i risultati è scrivere la deviazione standard tra parentesi accanto al risultato stesso, per esempio:

$$3.147\ (0.0051) \qquad (2.7)$$

La varianza di uno stimatore di Monte Carlo ha importanti utilizzi nella valutazione della qualità della stima dell'integrale pertanto risulta utile calcolarne un valore approssimativo.

Definizione 2.1. Uno stimatore $\hat{\theta}(Y)$ è una qualsiasi funzione delle variabili aleatorie $Y = (Y_1, \ldots,\ Y_n)$ che "si avvicina" al vero valore di θ. Si tratta di una variabile casuale.

Definizione 2.2. Una stima $\hat{\vartheta} = \hat{\theta}(y)$ è una qualsiasi funzione dei dati $y = (y_1, \ldots,\ y_n)$ che "si avvicina" al vero valore di θ. La stima è la realizzazione di $\hat{\theta}(Y)$, perciò si tratta di un numero.

Definizione 2.3. Uno stimatore $T = t(X_1, \ldots, X_n)$ si definisce *stimatore non distorto* (o corretto) di θ se e solo se il suo valore atteso coincide con esso:

$$E_\theta[T] = E_\theta[t(\ldots\ldots, X_n)] = \theta \qquad \text{per ogni theta} \qquad (2.8)$$

In altre parole, uno stimatore è non distorto se in media i suoi valori uguagliano il parametro che stima.

Oltre a ciò, si tratta di uno stimatore *consistente*. Per poterlo definire come tale, dobbiamo ora introdurre la legge che sta alla base del metodo Monte Carlo, ossia la *Legge dei Grandi Numeri* [6]:

Teorema 2.1. Sia $(X_n)_n$ una successione di v.a. indipendenti ed aventi tutte la stessa legge. Supponiamo che esse abbiano speranza matematica μ e varianza finita σ^2. Allora, posto

$$\bar{X}_n = \frac{1}{n}(\dots + \dots + X_n), \tag{2.9}$$

si ha, per ogni $\eta > 0$,

$$\lim_{n\to\infty} P(|\bar{X}_n - \mu| \geq \eta) = 0 \tag{2.10}$$

2.2.1. PROPRIETÀ DELLO STIMATORE MONTE CARLO

Nel paragrafo 2.1.1. siamo stati in grado di approssimare l'integrale definito arrivando alla equazione (2.6). Una stima Monte Carlo, di solito, ha la forma

$$\hat{\theta} = \mathrm{c}\frac{\sum f_i}{m} \tag{2.11}$$

e la varianza ha la forma

$$V(\hat{\theta}) = c\int \left(f(x) - \int f(t)dt\right)^2 dx \tag{2.12}$$

Questo stimatore è appropriato solo se gli elementi dell'insieme di variabili casuali F_i, nelle quali abbiamo le osservazioni f_i, sono indipendenti e quindi hanno zero correlazioni. Esistono vari metodi per la stima della varianza, nel paragrafo 2.2.2. si tratterà di questo argomento.

Prima di continuare mostriamo che lo stimatore Monte Carlo:

a) è uno *stimatore non distorto*
b) ha una varianza che diminuisce al crescere del numero di punti usati.

Per il punto a) è sufficiente applicare La Legge dei Grandi Numeri che ci dice che quando *n* si avvicina all'infinito, lo stimatore Monte Carlo (2.11) converge in probabilità a θ. In altre parole, la probabilità che $\hat{\theta}$ sia diversa da θ diventa nulla quando *n* diventa molto grande. In effetti è proprio grazie alla Legge dei Grandi Numeri che garantisce:

$$P\left\{\lim_{n\to\infty} I = \theta\right\} = 1 \tag{2.13}$$

Inoltre, il punto b) ci dice che aumentando l'ampiezza del campione *n*, lo stimatore fornisce un'approssimazione sempre migliore di θ.

Bisogna ora prendere in considerazione un altro risultato importante che otteniamo dallo stimatore di Monte Carlo, ossia la varianza. Infatti, come tutte le approssimazioni, anche questo metodo comporta un errore; infatti, lo stimatore ${\sigma^2}_M$ della varianza è:

$$ {\sigma^2}_M \approx \frac{\sigma^2}{n} \tag{2.14} $$

Si dimostra la (2.14) con le proprietà della varianza, infatti [7]:

$$ Var(\bar{X}) = Var\left(\frac{1}{n}X_1 + \frac{1}{n}X_2 + \ldots + \frac{1}{n}X_n\right) = \sum_{i=1}^{n}\left(\frac{1}{n}\right)^2 Var(X_i) = \frac{n\sigma^2}{n^2} = \frac{\sigma^2}{n} \tag{2.15} $$

Possiamo ora dire che, in concordanza con il punto b), la varianza diminuisce all'aumentare dell'ampiezza *n* del campione, maggiore sarà il campione, minore sarà la dispersione.

La relativa deviazione standard σ_M è data da:

$$ \sigma_M \approx \frac{\sigma}{\sqrt{n}} \tag{2.16} $$

È evidente il ruolo chiave del numero *n* di valori utilizzati. Si noti che questa espressione implica che l'errore diminuisce con la radice del numero di prove, il che significa che abbiamo bisogno di *n* volte più campioni per ridurre l'errore dello stimatore di $\frac{n}{2}$, ossia della metà. Una maggiore ampiezza del campione aumenta la precisione ma presenta anche un costo in termini di risorse; infatti, il calcolatore impiegherà sempre più tempo all'aumentare di *n*.

2.2.2. METODO ''BATCH MEANS''

Un metodo comune per stimare la varianza in una sequenza di osservazioni non indipendenti è quello di utilizzare le medie delle sottosequenze successive abbastanza lunghe, tali che le osservazioni in una sottosequenza siano quasi indipendenti dalle osservazioni in un'altra sottosequenza. Le medie delle sottosequenze sono chiamate "Batch Means". Si tratta della tecnica dei Batch means che si basa essenzialmente sulla incorrelazione realizzazioni ottenute dal metodo Monte Carlo. Data la sequenza $F_1, \ldots, F_b, F_{b+1}, \ldots, F_{2b}, F_{2b+1}, \ldots, F_{kb}$, ossia una sequenza di variabili casuali tale che la correlazione tra F_i e F_{i+b} è approssimativamente zero, una stima della varianza della media, $\bar{F}$, delle $m = kb$ variabili casuali può essere sviluppata osservando che

$$ V(\bar{F}) = V\left(\frac{1}{m}\sum F_i\right) = V\left(\frac{1}{k}\sum_{j=1}^{k}\left(\frac{1}{b}\sum_{i=(j-1)b+1}^{jb} F_i\right)\right) $$

$$ \approx \frac{1}{k^2}\sum_{j=1}^{k} V\left(\frac{1}{b}\sum_{i=(j-1)b+1}^{jb} F_i\right) \approx \frac{1}{k}V(\bar{F}_b) $$

Dove $\bar{F}_b$ è la media di un Batch di lunghezza b. Se i lotti sono abbastanza lunghi, può essere ragionevole supporre che le medie abbiano una varianza comune. Uno stimatore della varianza di $\bar{F}_b$ è la varianza campione standard dalle k osservazioni, $\bar{f}_1, \ldots, \bar{f}_k$:

$$\frac{\sum(\bar{f}_j - \bar{f})^2}{k-1} \qquad (2.17)$$

Quindi, lo stimatore delle Batch Means della varianza di $\bar{F}$ è

$$\hat{V}(\bar{F}) = \frac{\sum(\bar{f}_j - \bar{f})^2}{k(k-1)} \qquad (2.18)$$

Questo stimatore di varianza Batch Means dovrebbe essere utilizzato se lo studio di Monte Carlo produce un flusso di osservazioni non indipendenti, come in una serie temporale o quando la simulazione utilizza una catena di Markov. La dimensione dei sottocampioni dovrebbe essere il più piccola possibile e avere ancora mezzi indipendenti.

2.3. APPROSSIMAZIONE DELLA DISTRIBUZIONE NORMALE CON IL METODO MONTE CARLO

Prima di concludere con il secondo capitolo mostriamo un esempio applicativo in R che possa riassumere il funzionamento dell'approssimazione Monte Carlo. Prenderemo in considerazione *X* una Normale Standard di cui si vuole approssimare la probabilità $P(X > a)$. Per fare ciò implementiamo una funzione in R che racchiuda i passaggi dell'approssimazione nominandola "`Metodo_MC`". L'output finale della funzione sarà una tabella che comprenderà il valore reale, la stima di essa, la varianza e l'errore standard dello stimatore (formula (2.14) e (2.16)). Lo scopo è quello di mostrare operativamente il metodo e verificare se l'approssimazione Monte Carlo migliora al crescere del numero delle repliche *n* (secondo quanto detto nel paragrafo 2.2.1.).

Esempio 2.1. Sia X una v.c. continua $X \sim N(0,1)$ di cui si vuole approssimare la probabilità

$$P(X > 1)$$

Come prima cosa, all'interno della funzione `Metodo_MC`, generiamo *n* valori con la funzione `rnorm()` e creiamo *Z* un vettore logico che verifica se $X > 1$ o meno. Prima di ottenere l'approssimazione calcoliamo il reale valore della probabilità come `1 - pnorm(1)` che useremo per il confronto finale. Ora l'approssimazione la otteniamo considerando la proporzione di successi (formula (2.5)), ossia la media di *Z* (`mean(Z)`). La varianza (formula (2.14)) è stata calcolata grazie all'ausilio della funzione `var()` al numeratore ed infine per l'errore standard (formula (2.16)) ci è bastato calcolare la radice quadrata della varianza (`sqrt(varianza)`). Alla fine della funzione `Metodo_MC` si riorganizzano tutti i valori all'interno di `approssimazione` così da poter avere i dati ordinati.

```
> Metodo_MC <- function(n){
+   X <- rnorm(n)
+   Z <- (X > 1)
+   c <- c(1 - pnorm(1))
+   stima <- mean(Z)
+   varianza <- (var(Z) / n)
```

```
+   errore_standard <- sqrt(varianza)
+   approssimazione <- c(c, stima, varianza, errore_standard)
+   names(approssimazione) <- c("Valore","Stima valore", "Var.
stim.", "Err. std. stim.")
+   approssimazione
+ }
```

Creata la funzione `Metodo_MC`, possiamo ora provare con alcuni valori diversi di n. Nel nostro esempio i valori di *n* sono stati rispettivamente 100, 5000 e 10^6. Nella tabella Tabella 2.1 sono stati riassunti i risultati dell'esempio.

```
> Metodo_MC(100)
        Valore    Stima valore      Var. stim. Err. std.
stim.
    0.158655254     0.190000000     0.001554545
0.039427724
>
> Metodo_MC(5000)
        Valore    Stima valore      Var. stim. Err. std.
stim.
   1.586553e-01    1.618000e-01    2.712958e-05    5.208606e-
03
>
> Metodo_MC(10^6)
        Valore    Stima valore      Var. stim. Err. std.
stim.
   1.586553e-01    1.584090e-01    1.333157e-07    3.651243e-
04
```

n	Valore esatto	Stima Monte Carlo	Varianza stimatore	Errore standard stimatore
100	0,158655254	0,190000000	0,001554545	0,039427724
5000	0,158655254	1,618e-01	2,712958e-05	5,208606e-03
10^6	0,158655254	1,58409e-01	1,333157e-07	3,651243e-04

Tabella 2.1.: tabella riassuntiva valori generati.

Dall'esempio 2.1. è possibile vedere come all'aumentare di *n* l'approssimazione Monte Carlo aumenti la propria precisione, avvicinandosi sempre più al valore reale. Si noti inoltre come la varianza dello stimatore diminuisca all'aumentare di *n*. Questo risultato conferma quanto detto nel paragrafo 2.2.1.: si tratta di uno stimatore non distorto (punto a)) e la varianza diminuisce al crescere dei punti usati (punto b)).

CAPITOLO 3 - APPLICAZIONE AZIENDALE

3.1. SIMULAZIONE MONTE CARLO DEI RENDIMENTI DI UN TITOLO AZIONARIO

Come visto finora, il metodo Monte Carlo, può essere applicato a qualsiasi problema che coinvolga formule o equazioni. Ad esempio, nel campo della finanza è possibile effettuare la valutazione di portafoglio, nel campo del retail è possibile fare la stima delle vendite ed infine nel marketing si possono prevedere i profitti e le vendite.
In questo terzo e ultimo capitolo si vuole presentare un caso pratico di utilizzo del metodo riguardante la valutazione di portafoglio e per fare ciò simuleremo il rendimento di un titolo azionario in un determinato periodo di tempo.
L'esempio è una rielaborazione dell'esercizio proposto da Jonathan Regenstein [8] dal quale si riprende il concetto generale apportando delle implementazioni semplificate in R elaborate personalmente.
L'obiettivo di questa applicazione è quello di poter rispondere alla domanda "quali rendimenti possiamo aspettarci di fare sul mercato azionario in *n* giorni?", ossia una delle prime considerazioni che sorgono quando ci si affaccia al mondo degli investimenti.

Come visto nel corso del capitolo precedente ogni tipo di simulazione Monte Carlo è caratterizzata dalle seguenti fasi:

- definizione di un insieme di possibili input e della loro distribuzione;
- generazione di v.c. da una distribuzione di probabilità;
- esecuzione di un calcolo deterministico;
- analisi dei risultati ottenuti.

Per questa simulazione andremo a valutare il titolo azionario SPY e prenderemo i dati storici da Yahoo Finance grazie alla funzione `getSymbols()` di R:

```
> getSymbols("SPY",src = 'yahoo', auto.assign = TRUE)
[1] "SPY"
```

Avendo importato i dati storici dei rendimenti giornalieri del titolo, siamo ora in grado di calcolare la media e la deviazione standard dei rendimenti. Questi saranno i due parametri con i quali andremo a effettuare la nostra simulazione Monte Carlo:

```
#con la funzione dailyReturn() siamo in grado di calcolare i
rendimenti giornalieri dai quali ricaviamo la media e la
deviazione standard rispettivamente con le funzioni mean() e
sd()

> media <- mean(dailyReturn(SPY))
> std_dev <- sd(dailyReturn(SPY))
> media
[1] 0.0003439322
> std_dev
[1] 0.01285889
```

Abbiamo ora ottenuto che la media è 0,0003439322 e la deviazione standard è 0,01285889.

Assumiamo ora che il titolo sia normalmente distribuito e con i valori ottenuti andiamo a generare v.c. normalmente distribuiti.

Per la nostra simulazione prendiamo in considerazione un orizzonte temporale di 120 giorni e il nostro prezzo di partenza sarà l'ultimo dei valori storici importati in precedenza, ossia $p = 395{,}57$:

```
> n_giorni <- 120
> prezzo_t0 <- last(SPY$SPY.Close)[[1]]
> prezzo_t0
[1] 395.57
```

Nel seguente blocco di codice genereremo i rendimenti giornalieri usando la distribuzione normale con i parametri calcolati precedentemente, $N(0.0003439322;\ 0.01285889)$.

```
#impostiamo il seme affinché la simulazione sia riproducibile

> set.seed(73)

#generiamo una lista contenente i rendimenti giornalieri
percentuali del titolo

> rendimenti <- 1+rnorm(n_giorni, media, std_dev)

#generiamo una lista contenente la simulazione dei prezzi a
120 giorni. Per fare ciò calcoliamo il prodotto cumulativo del
prezzo di partenza e i rendimenti precedentemente calcolati

> prezzi <- cumprod(c(prezzo_t0, rendimenti))

#creiamo il grafico della simulazione effettuata (vedi figura
Figura 3.1)

> plot(prezzi, type='l', ylab="Prezzo simulato di SPY",
xlab="Giorni")
```

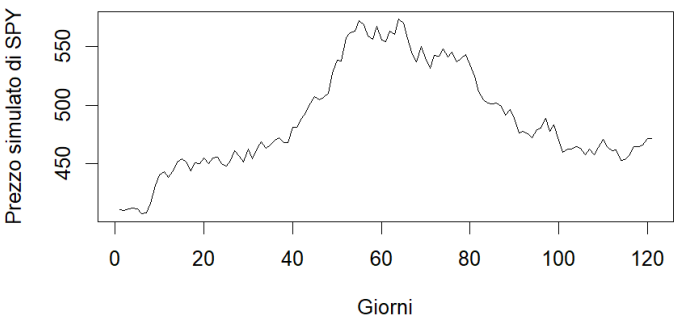

Figura 3.1: grafico della simulazione a 120 giorni del titolo SPY.

Come detto nei capitoli precedenti, una sola simulazione non risulta soddisfacente in termini di risultati ottenuti. Infatti, dalla Legge dei Grandi Numeri (vedi paragrafi 2.2. e 2.2.1.) sappiamo che

quando *n* si avvicina all'infinito, lo stimatore Monte Carlo (2.11) converge in probabilità al fenomeno che si vuole simulare; quindi, i risultati ottenuti da un gran numero di simulazioni sono più vicini al valore atteso. Più simulazioni eseguiamo, più possiamo fare affidamento sui dati.

```
#per ognuno dei 120 giorni eseguiamo 1001 simulazioni

> n_sim <- 1001

#creiamo, rispettivamente, una matrice vuota per i rendimenti
e una per i prezzi

> rendimenti_m <- matrix(0, nrow = n_sim, ncol = n_giorni)
> prezzi_m <- matrix(0, nrow = n_sim, ncol = n_giorni+1)

#come effettuato in precedenza generiamo i rendimenti da una
normale N(0.0003439322; 0.01285889) e calcoliamo i prezzi con il
prodotto cumulativo tra il prezzo di partenza e i rendimenti
calcolati. Con il ciclo for effettuiamo questo procedimento
per il numero di simulazioni desiderato, ossia 1001 volte nel
nostro esempio

> for(i in 1:n_sim) {
+   rendimenti_m[i,] <- rnorm(n_giorni, media, std_dev)
+   prezzi_m[i,] <- cumprod(c(prezzo_t0, 1+rendimenti_m[i,]))
+ }

#creiamo ora il grafico delle simulazioni effettuate
diversificandole per colore (vedi figura Figura 3.2). Per
semplicità non si prenderanno in considerazione tutte le
simulazioni ma soltanto le prime 50

> plot(prezzi_m[1,], type='l', ylab="Prezzo simulato di SPY",
xlab="Giorni", ylim=c(300, 580))
> for(i in 1:50) {
+   lines(prezzi_m[i, ], type = 'l', col=i)
+ }
```

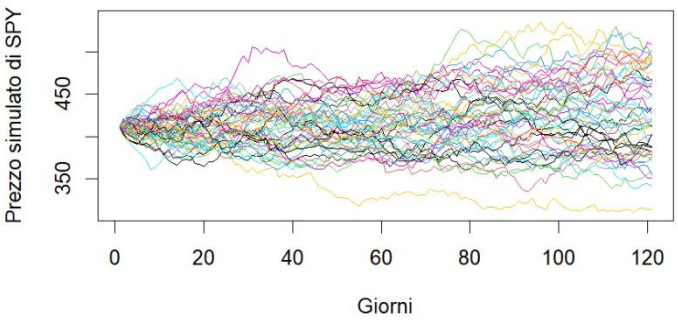

Figura 3.2: grafico delle prime 50 simulazioni effettuate.

Dal grafico possiamo vedere come (anche se abbiamo considerato solo 50 simulazioni) i risultati ottenuti varino l'uno rispetto all'altro. Ci sono simulazioni che portano sia il prezzo al di sotto del prezzo di partenza, che al di sopra. Da questa prima analisi, supponiamo che lo stesso comportamento è seguito anche dalle restanti simulazioni.

Possiamo ora calcolare i rendimenti totali:

```
#creiamo un vettore con numero di dimensione pari al numero
delle simulazioni effettuate

> rendimenti_tot <- array(NA, dim= n_sim, dimnames=NULL)

#calcoliamo i rendimenti totali in percentuale per ognuno dei
120 giorni che inseriamo nel vettore creato in precedenza

> for (i in 1:n_sim) {
+   rendimenti_tot[i] <- (prezzi_m[i, 121]-prezzi_m[i,
1])/prezzi_m[i,1]
+ }
```

Dai dati appena calcolati, possiamo ora fare alcune analisi statistiche:

> summary(rendimenti_tot)

Minimo	Primo quartile	Mediana	Media	Terzo quartile	Massimo
-0.29509	-0.05471	0.03099	0.03941	0.12770	0.54367

Tabella 3.1.: tabella rappresentante le caratteristiche di sintesi dei rendimenti.

> var(rendimenti_tot)
[1] 0.02258884
> sd(rendimenti_tot)
[1] 0.1502959

Secondo quanto appena calcolato (vedi tabella Tabella 3.1) possiamo dire che in media la simulazione offre dei rendimenti positivi e che, essendo il primo quartile quasi zero, in totale un po' più del 25% delle simulazioni effettuate ha un rendimento negativo.

Per semplificare la lettura dei dati generati, creiamo un grafico dei prezzi che contenga solo tre simulazioni, ossia il minimo, la mediana e il massimo.

```
#con la funzione which calcoliamo rispettivamente la posizione
del massimo e del minimo all'interno dei rendimenti totali.
Questa posizione la useremo in seguito per trovare i valori,
rispettivamente, del prezzo massimo e minimo all'interno della
matrice dei prezzi calcolata in precedenza

> massimo <- which.max(rendimenti_tot)
> minimo <- which.min(rendimenti_tot)
> if (n_sim %% 2 != 0) {
+   mediana <- match(median(rendimenti_tot), rendimenti_tot)
+ }

#creiamo il grafico contenente il prezzo massimo, minimo e
mediano, differenziando per colore (vedi figura Figura 3.3)

> plot(prezzi_m[minimo,], type='l', ylab=" Prezzo simulato di
SPY", xlab="Giorni", col="red", ylim=c(250, 620))
> if (n_sim %% 2 != 0) {
+   lines(prezzi_m[mediana, ], type = 'l', col='blue')
+ }
> lines(prezzi_m[massimo, ], type = 'l', col='green')
```

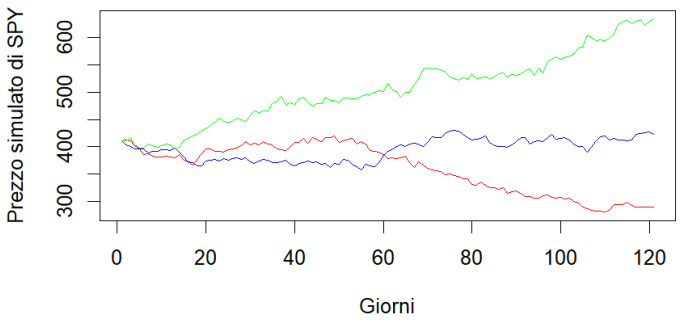

Figura 3.3.: grafico con il prezzo massimo, mediano e minimo delle simulazioni.

Ora possiamo facilmente visualizzare il trend della simulazione migliore, peggiore e mediana. A inizio capitolo ci siamo chiesti "quali rendimenti possiamo aspettarci di fare sul mercato azionario in *n* giorni?" e ora possiamo dare una risposta. Secondo i calcoli effettuati finora, possiamo aspettarci un rendimento percentuale che va dal -29,51% al 54,37%. Abbiamo preso questi risultati dalla tabella Tabella 3.1, considerando il minimo ed il massimo. È possibile visualizzare questi risultati nel graficoFigura 3.3. dove la linea rossa corrisponde al prezzo minimo e il verde a quello massimo.
Con questa stessa applicazione siamo in grado di fare ulteriori considerazioni. Possiamo calcolare la probabilità di ottenere un rendimento superiore a una determinata soglia o, viceversa, la probabilità di perdere più di una determinata soglia considerando un investimento a 120 giorni.

Per il nostro esempio imponiamo come soglia minima di rendimento 12% e come soglia massima di perdita -10%. Consideriamo ora solo i risultati che soddisfano le nostre esigenze che poi useremo per calcolare la probabilità che si verifichino.

```
#calcoliamo, rispettivamente, la probabilità che il rendimento
sia superiore al 12% e la probabilità che le perdite siano
maggiori del 10%

> p_1 <- (sum(rendimenti_tot> 0.12)/n_sim)*100
> p_1
[1] 26.47353
> p_2 <- (sum(rendimenti_tot < -0.1)/n_sim)*100
> p_2
[1] 16.68332
```

La probabilità di ottenere un rendimento maggiore del 12% è 26,47%, e la probabilità di perdere più del 20% è 16.68%.

Come visto finora siamo stati in grado di calcolare i rendimenti futuri di un dato titolo azionario conoscendone i dati storici e le probabilità di successo o di insuccesso dello stesso. I passaggi appena svolti sono applicabili anche nel caso di un portafoglio più complesso, composto da più titoli, con l'aggiunta di maggiori componenti quali: il peso dei titoli nel portafoglio, la raccolta dei dati storici di ogni titolo, aggiunta di formule pesate e la definizione di maggiori funzioni. Per quanto possa rendere in efficienza, un esempio del genere risulterebbe di più difficile lettura e implementazione, pertanto, si rimanda alla bibliografia per maggiori approfondimenti a riguardo [8].

L'applicazione presentata in questo capitolo è una rappresentazione della realtà semplificata, della complessità delle variabili e dipende inoltre della precisione dei dati inseriti. Un modo per renderla più realistica potrebbe essere quello di considerare le correlazioni tra le variabili caratterizzanti, quali ad esempio il prezzo e gli agenti esogeni: basti pensare all'impatto che il COVID-19 ha avuto sul mercato azionario.
Il metodo Monte Carlo può quindi essere un utile supporto per i nostri calcoli.

BIBLIOGRAFIA

1. Gentle, J.E. 1998. “Random Number Generator and Monte Carlo Methods”. New york, Springer.
2. Mitzenmacher, “Tossing a biased coin”. Harvard university: pp. 1-3.
3. Lehmer, D. H. 1951. “Mathematical Methods in Large Scale Computing Units”. The Annals of the Computation Laboratory of Harvard University 26: pp. 141–146.
4. Mood, A. M. 1988. “Introduzione alla statistica”. Milano, McGraw-Hill: pp210-211.
5. Kalos, M. H. e Whitlock, P. A 2008. “Monte Carlo Methods”, Weinheim, Wiley-VCH: pp. 1-5.
6. Baldi, P. 2011. “Calcolo delle probabilità”. Milano, McGraw-Hill.
7. Newbold, P. 2010. “Statistica”. Milano, Pearson: pp. 264-267.
8. Regenstein J. 2018. “Monte Carlo”. https://rviews.rstudio.com/2018/06/05/monte-carlo/.