



Introduzione ai metodi di simulazione Montecarlo. Esercitazioni con Matlab *

Carlo Nardone

CRS4, Centro di Ricerca, Sviluppo e Studi Superiori in Sardegna

Via Nazario Sauro, 10 - 09123 Cagliari

E-mail: cmn@crs4.it

Web URL: <http://www.crs4.it/~cmn>

Agosto 1997

*Questi appunti sono stati preparati, in forma lievemente diversa, per il "Corso Operational di Logistica Postale Integrata" svoltosi a Roma nel Febbraio 1997 presso la Divisione Formazione delle Poste Italiane.

Indice

1	Generatori	1
1.1	Generatori pseudocasuali	1
1.2	Test sui generatori pseudocasuali	2
1.3	L'ago di Buffon	3
1.4	Generazione di distribuzioni non uniformi	3
1.4.1	Metodo del test sull'intervallo per sequenze discrete	3
1.4.2	Metodo d'inversione	4
1.4.3	Metodo di reiezione	5
2	Integrali	9
2.1	Calcolo di integrali	9
2.1.1	Montecarlo <i>hit-or-miss</i>	9
2.1.2	Montecarlo <i>naive</i>	9
2.1.3	Montecarlo con campionamento d'importanza	10
3	Code	13
3.1	Simulazione di una coda M/G/1	13
4	Ottimizzazione	16
4.1	Ottimizzazione combinatoria: il problema TSP ("commesso viaggiatore")	16
4.2	Ottimizzazione stocastica: il metodo SA ("raffreddamento simulato")	16
4.2.1	Random search	17
4.2.2	Visite di Markov	17
4.2.3	Algoritmo di Metropolis	17
4.2.4	Raffreddamento simulato	18
	Bibliografia	20
A	Listati degli script Matlab	21
A.1	Capitolo 1	21
A.1.1	ranmult.m	21
A.1.2	autocorr.m	21
A.1.3	buffon.m	21
A.1.4	expdist.m	22
A.1.5	poisson.m	22

A.1.6	norm1.m	22
A.1.7	norm2.m	22
A.1.8	norm3.m	23
A.2	Capitolo 2	23
A.2.1	cerchio.m	23
A.2.2	hitmiss.m	23
A.2.3	mcnaive.m	24
A.2.4	funz.m	24
A.2.5	campiona.m	24
A.2.6	mcimport.m	24
A.3	Capitolo 3	24
A.3.1	funz.m	24
A.3.2	lognorm.m	25
A.3.3	theta.m	25
A.3.4	runcoda.m	25
A.3.5	coda.m	27
A.4	Capitolo 4	28
A.4.1	points.m	28
A.4.2	distance.m	28
A.4.3	permuta.m	28
A.4.4	move.m	28
A.4.5	move2.m	29
A.4.6	metrop.m	29
A.4.7	runtsp1.m	30
A.4.8	runtsp2.m	30
A.4.9	runtsp3.m	31
A.4.10	runtsp4.m	32
A.4.11	runtsp5.m	33

Capitolo 1

Generatori

1.1 Generatori pseudocasuali

Lo strumento basilare per condurre simulazioni di carattere stocastico è costituito dalla generazione al calcolatore di numeri “casuali”. L'apparente paradosso ¹ di considerare come casuali dei numeri generati da algoritmi, cioè da prescrizioni deterministiche, è risolto se si considera l'uso che si fa di tali sequenze di numeri. Basta infatti che le sequenze ottenute da uno specifico generatore siano “correttamente” distribuite, nel senso che le sequenze devono passare una batteria più o meno estesa di test statistici.

In pratica, un generatore di numeri *pseudocasuali* (così chiamati per distinguerli dai numeri casuali “ideali”) consiste di particolari formule ricorsive. Il grande vantaggio di questo approccio è che una specifica sequenza può essere riprodotta esattamente a partire da un numero di partenza x_0 detto *seme*.

I generatori più semplici e diffusi sulla maggior parte dei calcolatori e dei pacchetti software sono del tipo

$$x_i \equiv ax_{i-1} + c \pmod{m} \quad (1.1)$$

(dove \pmod{m} indica il resto della divisione per m) detti *generatori lineari congruenti*. Nel caso in cui $c = 0$ il generatore è detto *moltiplicativo*.

Normalmente x_i viene diviso per m in modo da ottenere un numero reale $\xi_i = x_i/m$ distribuito uniformemente nell'intervallo $[0, 1)$. Se occorre un numero ψ compreso in $[a, b)$ basterà porre

$$\psi = a + (b - a) \xi . \quad (1.2)$$

Si noti che una sequenza che soddisfa la (1.1) si ripeterà al più dopo m passi, quindi è necessario scegliere m opportunamente grande in relazione all'uso che si vuole fare della sequenza pseudocasuale. Il periodo è in generale una frazione di m , il massimo è raggiunto se c e m sono primi fra loro (per $c \neq 0$), oppure se a e m sono primi fra loro (per $c = 0$). Poiché l'aritmetica dei calcolatori è quasi sempre in base binaria, spesso si sceglie $m = 2^\alpha$ per velocizzare i calcoli ($\alpha \approx 30 \div 40$). Si noti che la scelta degli opportuni a e c è critica, nel caso $c = 0$ lo è molto di più.

¹ “Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.” (John Von Neumann, 1951)

Esempi di generatori moltiplicativi congruenti (Tabella 1):

modulo m	fattore a	periodo	commenti
2^{31}	$2^{16} + 3 = 65\,539$	2^{29}	Park & Miller "minimal standard"
2^{35}	$5^{13} = 1\,220\,703\,125$	2^{33}	
$2^{31} - 1$	$7^5 = 16\,807$	$2^{31} - 2$	

Esempi di generatori lineari congruenti (Tabella 2):

modulo m	fattore a	costante c	commenti
2^{32}	1812433253	dispari	buono
2^{35}	$2^{18} + 1$	1	scadente

Generatori più sofisticati possono essere costruiti introducendo un ulteriore livello di (pseudo-)casualità nell'output x_i (aspettando in media q passi) oppure combinando due sequenze indipendenti con differenti periodi (L'Ecuyer), ottenendo una sequenza con periodo che è il minimo comune multiplo dei due periodi originari.

1.2 Test sui generatori pseudocasuali

Alcune proprietà che devono essere necessariamente soddisfatte da sequenze pseudocasuali (riferendosi ai numeri ξ_i distribuiti uniformemente in $[0, 1)$):

$$\bar{\xi} \equiv \frac{1}{N} \sum_{i=1}^N \xi_i = \frac{1}{2} \quad (1.3)$$

$$\sigma_{\xi}^2 \equiv \frac{1}{N} \sum_{i=1}^N (\xi_i - 1/2)^2 = \frac{1}{12} \quad (1.4)$$

(statistica univariata della distribuzione uniforme su $[0, 1)$);

$$r_{\xi}(k) \equiv \frac{1}{N} \sum_{i=1}^N (\xi_i - 1/2)(\xi_{i+k} - 1/2) = 0 \quad (k \geq 1) \quad (1.5)$$

$$r_{\xi}^{(2)}(k, l) \equiv \frac{1}{N} \sum_{i=1}^N (\xi_i - 1/2)(\xi_{i+k} - 1/2)(\xi_{i+l} - 1/2) = 0 \quad (k, l \geq 1) \quad (1.6)$$

$$r_{\xi}^{(3)}(k, l, m) \equiv \dots$$

(statistiche multivariate per stabilire l'indipendenza degli ξ_i).

Esercizio 1.1: *Provare un generatore moltiplicativo a scelta (ad es. $X = \text{ranmult}(n)$), e confrontarlo con la funzione intrinseca $\text{rand}()$. Generare un vettore $X(n, 1)$, graficarne l'istogramma con b intervalli ($\text{hist}(X, b)$), calcolarne media ($\text{mean}(X)$), deviazione standard ($\text{std}(X)$), e le correlazioni $r(k)$ (1.5) (usare $\text{autocorr}(X)$).*

Si può confrontare direttamente la distribuzione sperimentale delle sequenze pseudocasuali con la distribuzione uniforme mediante il test del χ^2 . In questo caso si costruisce la quantità

$$\chi^2 = \frac{k}{N} \sum_{i=1}^k (N_i - N/k)^2 \quad (1.7)$$

dove N è il numero di campioni, k il numero di partizioni dell'istogramma ed N_i il numero osservato di campioni per ogni partizione i . Se la distribuzione sperimentale è davvero uniforme, χ^2 deve a sua volta avere una distribuzione conosciuta ($Q(\chi^2, k)$) e tabulata (vedi Fig.1.1). In pratica, valori di χ^2 molto maggiori di k indicano che la sequenza pseudocasuale è scadente. Si può applicare lo stesso test a coppie, triplette, ecc. di numeri della sequenza in esame.

1.3 L'ago di Buffon

Un esempio storico celebre dell'uso di metodi stocastici per il calcolo di grandezze matematiche è costituito dal calcolo di π mediante il lancio del cosiddetto *ago di Buffon*. Su un piano diviso da linee parallele distanziate di una lunghezza d si lancia un ago lungo $l < d$ (vedi Fig.1.2.a). Si dimostra che la probabilità che l'ago incroci una delle linee è $p = 2l/\pi d$.

Infatti la posizione dell'ago è determinata univocamente dalle quantità $x \in [0, d/2)$ (la distanza del centro dell'ago dalla più vicina parallela) e $\phi \in [0, \pi)$ (l'angolo formato dall'ago con questa parallela), per cui la condizione d'intersezione si può scrivere $x \leq l/2 \sin \phi$. La probabilità richiesta sarà data dall'area "favorevole" sottesa da $\sin \phi$ divisa per l'area della totalità degli eventi, cioè l'intero rettangolo di area $\pi d/2$ (vedi Fig.1.2.b). L'area favorevole è l'integrale $\int_0^\pi l/2 \sin \phi = l$.

Quindi è possibile stimare π dalla frequenza m di intersezioni ottenute su n lanci con $\pi \approx 2ln/dm$.

Esercizio 1.2: stimare π usando la formula precedente e "lanciando" n coppie pseudocasuali (x, ϕ) (`X = buffon(n,d,l)`).

1.4 Generazione di distribuzioni non uniformi

Una sequenza di numeri pseudocasuali distribuiti uniformemente $\xi \in [0, 1)$ si può considerare come la base per la costruzione di sequenze distribuite in vario modo. Ci sono metodi generali per ottenere distribuzioni volute, come pure metodi *ad hoc* in casi particolari.

1.4.1 Metodo del test sull'intervallo per sequenze discrete

Supponiamo di voler generare una sequenza a distribuzione *discreta* cioè i cui valori Y appartengono all'insieme $\{y_i | i = 1..n\}$ con n finito, le cui probabilità sono rispettivamente $\{p_i | i = 1..n\}$. Allora basterà utilizzare la sequenza di numeri uniformi ξ scegliendo $Y = y_i$ se $\sum_{j=1}^{i-1} p_j < \xi \leq \sum_{j=1}^i p_j$. In pratica, questo metodo è conveniente se n non è troppo grande.

1.4.2 Metodo d'inversione

Supponiamo che la variabile casuale Y sia distribuita secondo la funzione $p(y)$ che ha per integrale (funzione cumulativa) una funzione $F(y) = \int_{-\infty}^y p(y') dy'$ invertibile. Si dimostra facilmente che la variabile casuale $F^{-1}(\xi)$ è distribuita come Y (vedi Fig.1.3.a). Il metodo è conveniente se la F^{-1} si può ricavare esplicitamente, altrimenti occorre applicare metodi numerici per invertire la $F(y)$.

Distribuzione esponenziale

Un esempio immediato del metodo è costituito dalla distribuzione *esponenziale* per la quale $p(y) = 0$ per $y < 0$ e $p(y) = \lambda e^{-\lambda y}$ per $y \geq 0$. Questa distribuzione modella i tempi di attesa tra eventi poissoniani indipendenti, quindi è importante nella simulazione di code. In altre parole, se ad uno sportello si presentano in media λ nuovi utenti al minuto, gli intervalli di tempo fra nuovi arrivi sono distribuiti esponenzialmente.

In questo caso la funzione cumulativa è $F(y) = 0$ per $y < 0$ e $F(y) = 1 - e^{-\lambda y}$ per $y \geq 0$. Quindi $y \equiv F^{-1}(\xi) = -(1/\lambda) \ln(1 - \xi)$ oppure direttamente $y = -(1/\lambda) \ln(\xi)$.

Distribuzione poissoniana

Si dimostra che la distribuzione di Poisson $p_i = (\lambda^i / i!) e^{-\lambda}$ può essere ottenuta contando quanti eventi distribuiti esponenzialmente (con parametro λ) sono accaduti nell'intervallo $(0, 1]$. In altre parole si può generare una sequenza y_j distribuita esponenzialmente con il metodo precedente e fermarsi al più piccolo numero i tale che $\sum_{j=1}^{i+1} y_j > 1$. In pratica questo metodo è conveniente per λ (che equivale alla media degli i) non troppo grande.

Tornando all'esempio dello sportello con una media di λ nuovi utenti al minuto, il numero di arrivi al minuto segue la distribuzione di Poisson.

Esercizio 1.3: Generare sequenze esponenziali e poissoniane con i metodi descritti ($X = \text{expdist}(n, \lambda)$ e $X = \text{poisson}(n, \lambda)$), notando medie e deviazioni standard.

Distribuzione normale (gaussiana)

La distribuzione gaussiana $p(y) = 1/(\sqrt{2\pi}\sigma) \exp[-(y - \mu)^2 / \sigma^2]$ (per brevità indicata con $\mathcal{N}(\mu, \sigma)$) è solitamente ottenuta sfruttando il teorema del limite centrale, secondo il quale la distribuzione della variabile $Y = (\sum_{i=1}^n \xi_i - n/2) / \sqrt{n/12}$ tende appunto alla distribuzione normale $\mathcal{N}(0, 1)$ per grandi n . Per ottenere un numero z distribuito come $\mathcal{N}(\mu, \sigma)$ da un numero y distribuito come $\mathcal{N}(0, 1)$ basta porre $z = \mu + \sigma y$. Per semplicità si sceglie $n = 12$ per cui $Y = \sum_{i=1}^{12} \xi_i - 6$. Si noti che con questo metodo non si potranno mai ottenere valori fuori dall'intervallo $[-6, 6]$, il che può essere un grave difetto in molte applicazioni.

Il metodo di Box-Muller si basa invece sull'inversione della distribuzione normale in due dimensioni $p(y_1, y_2) = 1/(2\pi) \exp[-(y_1^2 + y_2^2)]$ (y_1 e y_2 sono indipendenti e distribuiti con $\mathcal{N}(0, 1)$). Si ottiene:

$$\begin{aligned}
 y_1 &= \sqrt{-2 \ln(\xi_1)} \cos(2\pi \xi_2) \\
 y_2 &= \sqrt{-2 \ln(\xi_1)} \sin(2\pi \xi_2) .
 \end{aligned}
 \tag{1.8}$$

1.4.3 Metodo di reiezione

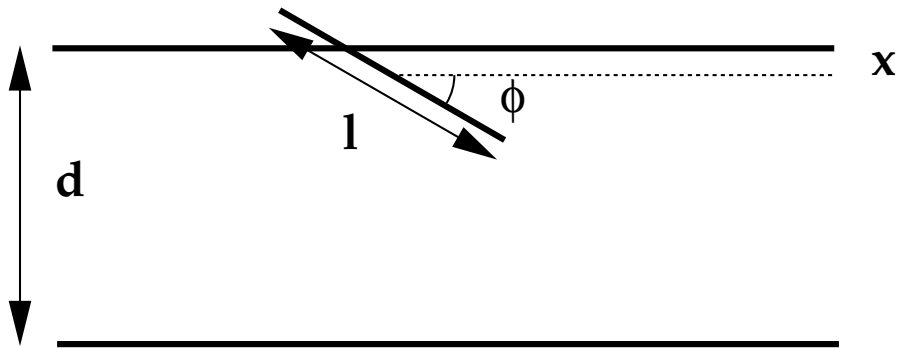
Il metodo di *reiezione* permette di ricavare distribuzioni $p(x)$ per le quali il metodo d'inversione è inapplicabile, purchè sia possibile individuare una funzione $f(x)$ di confronto tale che $f(x) > p(x)$ per ogni x . Se si riesce a generare una sequenza di punti $(x, y)_i$ distribuiti uniformemente nell'area sottesa da $f(x)$, se ne otterrà una sequenza distribuita secondo $p(x)$ accettando quei x_i per cui $y_i < p(x_i)$.

Il metodo composto inversione-reiezione utilizza inoltre l'inversione dell'integrale della $f(x)$ per ottenere dapprima x_i . $y_i \in [0, x_i)$ viene poi ricavato da un altro generatore uniforme. In questo modo $(x, y)_i$ è distribuito uniformemente sotto $f(x)$ come richiesto (vedi Fig.1.3.b).

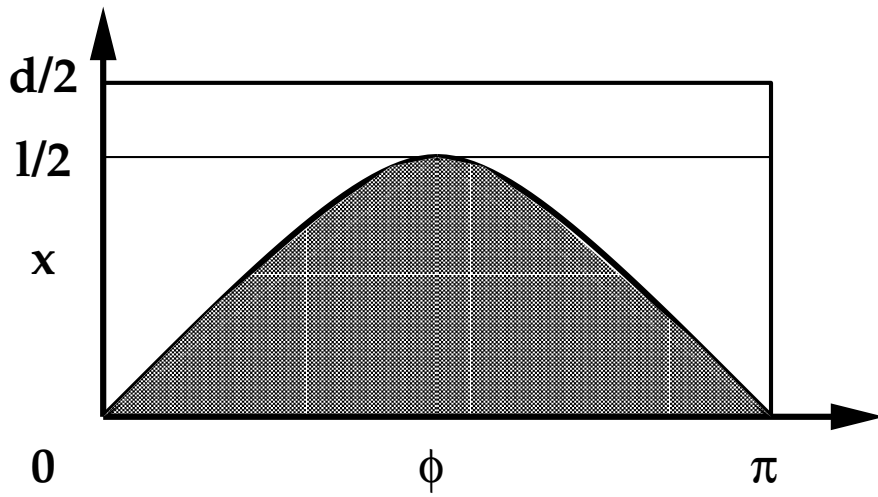
Un'applicazione del principio della reiezione alla generazione di sequenze gaussiane porta a un significativo guadagno di efficienza sul metodo del paragrafo precedente (metodo *polare*). Basta infatti generare punti $(x, y)_i$ nel quadrato $([0, 1], [0, 1])$ e accettando solo quelli entro il cerchio unitario, in modo da costruire $R^2 = x^2 + y^2$ che è a sua volta distribuito uniformemente e fa le veci di ξ_1 in (1.8). Il vantaggio sta nel fatto che \cos e \sin sono sostituiti da x/R e y/R , evitando chiamate a funzioni trigonometriche.

Esercizio 1.4: generare sequenze di n numeri normali con i tre metodi precedenti (rispettivamente $X = \text{norm1}(n)$, $[X1, X2] = \text{norm2}(n)$, $[X1, X2] = \text{norm3}(n)$) e confrontarle con la funzione intrinseca $X = \text{randn}()$.

Figura 1.1: Test del χ^2 su sequenze pseudocasuali; frequenza dei valori di χ^2 osservati in 1000 campioni distribuiti su intervalli equiprobabili (da Kalos e Whitlock). Risultati per i generatori lineari congruenti della Tabella 2: a) prima riga (buono) b) seconda riga (scadente).



a)



b)

Figura 1.2: a) Geometria per l'ago di Buffon. b) Formalizzazione della probabilità come rapporto di aree.

Figura 1.3: a) Metodo d'inversione per generare numeri pseudocasuali secondo la distribuzione $p(y)$ a partire da una sequenza uniforme. b) Metodo d'inversione-reiezione (da Press *et al*).

Capitolo 2

Integrali

2.1 Calcolo di integrali

In questa sezione si introdurranno metodi Montecarlo per il calcolo di integrali. L'idea è quella di usare la casualità per stimare grandezze deterministiche. Per semplicità daremo esempi per funzioni monodimensionali $f(x)$, ma in realtà i metodi Montecarlo sono usati per integrali multidimensionali con dimensione elevate. Infatti in tal caso i metodi numerici "deterministici", cioè gli schemi di approssimazione su griglie regolari, convergono più lentamente.

2.1.1 Montecarlo hit-or-miss

Il metodo Montecarlo *hit-or-miss* per il calcolo di integrali è molto simile al metodo di reiezione del par.1.4.3.

Consideriamo come esempio l'integrale

$$I = \int_0^1 \sqrt{1-x^2} dx = \frac{\pi}{4} \quad (2.1)$$

che corrisponde all'area di un quarto di cerchio unitario. Possiamo pensare di stimare I generando una sequenza di n punti distribuiti uniformemente nel quadrato $([0, 1], [0, 1])$ contando gli m punti che cadono nel quarto di cerchio (vedi Fig.2.1.a). Allora la stima *hit-or-miss* sarà $I^* = m/n$. In pratica basta prendere $2n$ valori ξ_i e considerare i punti (ξ_i, ξ_{i+1}) .

Poiché m è una variabile bernoulliana con probabilità $p = I$, la sua deviazione standard è $\sigma = \sqrt{np(1-p)}$ cioè l'errore per I^* sarà $\sqrt{I(1-I)/n} \approx \sqrt{I^*(1-I^*)/n}$.

2.1.2 Montecarlo naïve

Il metodo Montecarlo vero e proprio consiste nello stimare l'integrale $I = \int_a^b f(x)dx$ con la media aritmetica

$$I^* = (b-a) \bar{f} = (b-a) \frac{1}{n} \sum_{i=1}^n f(\xi_i) \quad (2.2)$$

dove come al solito gli ξ_i sono distribuiti uniformemente in $[a, b)$ (vedi Fig.2.1.b per l'esempio del quarto di cerchio unitario).

Si dimostra che l'accuratezza della stima è data da

$$\Delta \tilde{I} \simeq \sqrt{\frac{\overline{f^2} - \bar{f}^2}{n}} \quad (2.3)$$

che è superiore a quella del metodo *hit-or-miss* a parità di campioni generati.

Esercizio 2.1: Calcolare l'integrale (2.1) con il metodo *hit-or-miss* ([integrale, errore] = hitmiss(n)) e con *Montecarlo naive* ([integrale, errore] = mcnaive(n)), confrontandone l'accuratezza a parità di numero di campioni uniformi utilizzati e verificando la (2.3).

2.1.3 Montecarlo con campionamento d'importanza

L'idea è quella di campionare l'intervallo d'interesse con una distribuzione $p(x)$ diversa da quella uniforme e tale da pesare maggiormente i valori per cui $f(x)$ è più importante (cioè dove $|f(x)|$ è più grande) (vedi Fig.2.2). Si dimostra infatti che

$$I \equiv \int_a^b f(x)dx = \int_a^b [f(x)/p(x)]p(x)dx = \int_a^b [f(x)/p(x)]dP(x) \quad (2.4)$$

purché $p(x)$ sia normalizzata.

Quindi la stima (2.2) di I diviene

$$I^* = (b - a) \overline{(f/p)p} = (b - a) \overline{(f/p)}_p \equiv (b - a) \frac{1}{n} \sum_{i=1}^n f(x_i)/p(x_i) \quad (2.5)$$

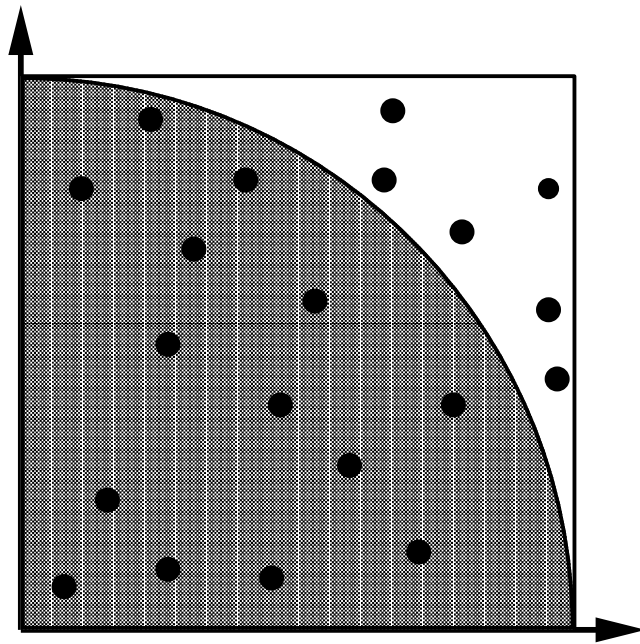
dove gli x_i campionano una funzione modificata (f/p) seguendo appunto la distribuzione $p(x)$. Si dimostra che questo espediente diminuisce enormemente la varianza della stima (2.3), cioè aumenta l'accuratezza a parità di campioni n .

Nel caso dell'integrale (2.1), ad esempio, si può scegliere

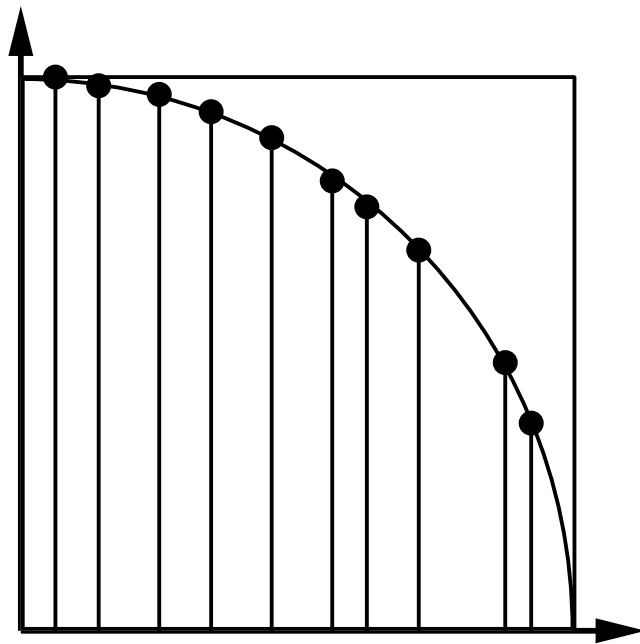
$$p(x) = (1 - \beta x^2)/(1 - \beta/3) \quad (2.6)$$

(si verifichi che $p(x) > 0$ ed è correttamente normalizzata in $[0, 1]$). La sequenza di x_i si ottiene con il metodo di reiezione, facendo attenzione al valore massimo di $p(x)$ che è $p(0) = 1/(1 - \beta/3)$.

Esercizio 2.2: Calcolare l'integrale dell'Eserc.2.1 con il campionamento d'importanza ([integrale, errore] = mcimport(n,beta)), utilizzando la distribuzione (2.6) con diversi valori di β , graficandone l'andamento ($y = funz(x, beta)$). Confrontarne l'accuratezza con i risultati precedenti.



a)



b)

Figura 2.1: a) Montecarlo *hit-or-miss*, b) Montecarlo *naive*.

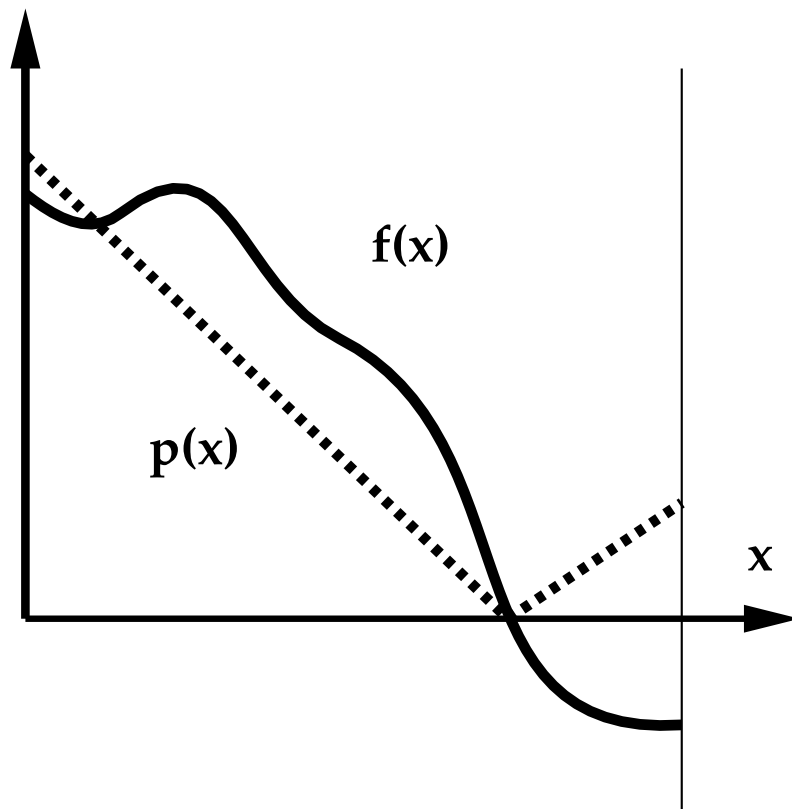


Figura 2.2: Campionamento d'importanza per la stima dell'integrale di $f(x)$ utilizzando come $p(x)$ una funzione semplice che approssima $|f(x)|$.

Capitolo 3

Code

3.1 Simulazione di una coda M/G/1

In questa sezione vedremo come la generazione di numeri pseudocasuali distribuiti in modo appropriato permetta di simulare al computer un sistema stocastico di grande interesse pratico (in contrasto con la sezione 2 qui usiamo tecniche stocastiche per un problema stocastico). Ci proponiamo di simulare una semplice coda, costituita da un solo ingresso e da un solo server (vedi Fig.3.1).

Si suppone che il processo stocastico che descrive i tempi di arrivo sia di tipo poissoniano, cioè la probabilità di arrivo in un intervallo dt è λdt ; in altre parole gli intertempi di arrivo $\{t^{(in)}\}$ sono distribuiti esponenzialmente secondo la $P(t^{(in)}) = \lambda \exp(-\lambda t^{(in)})$. Conseguentemente la media degli $\{t^{(in)}\}$ è $\langle t^{(in)} \rangle \equiv \tau_{in} = \lambda^{-1}$ mentre la deviazione standard è $\sigma(t^{(in)}) \equiv \langle t^{(in)2} \rangle - \langle t^{(in)} \rangle^2 = \tau_{in} = \lambda^{-1}$. Il processo stocastico che descrive i tempi di servizio può essere “generico”. La nomenclatura comunemente accettata parla in questo caso di sistema M/G/1.

Per specializzare M/G/1 in modo realistico supponiamo che i tempi di servizio $\{t_i^{(out)}\}$ siano distribuiti secondo una “lognormale”. Questo vuol dire che $\{\ln(t_i^{(out)})\}$ sono distribuiti come $\mathcal{N}(\mu, \sigma)$. Per generare una sequenza lognormale basterà quindi generare degli y_i normali e poi prendere $x_i = e^{y_i}$. In pratica, la lognormale assicura che i tempi $\{t_i^{(out)}\}$ siano sempre positivi (contrariamente alla normale), che la probabilità di $t_i^{(out)}$ piccoli sia bassa, che ci sia un picco vicino (ma minore) alla media $\langle t_i^{(out)} \rangle$ e che infine vi sia una “coda” di eventi rari ma con $t_i^{(out)}$ molto grande. Si noti che $\langle t_i^{(out)} \rangle$ non è affatto uguale a e^μ , ma a $\exp(\mu + \sigma^2/2)$; analogamente, per $\sigma(t_i^{(out)})$ vale una formula appena più complicata.

Esercizio 3.1: Studiare la funzione lognormale ($y = \text{funz}(x, \mu, \sigma)$) e confrontarla con l'istogramma sperimentale di sequenze lognormali ($Y = \text{lognorm}(n, \mu, \sigma)$).

La simulazione della coda vera e propria prevede la generazione di un set di tempi d'ingresso $\{t^{(in)}\}$ e d'uscita $\{t_i^{(out)}\}$ che modificano lo stato del sistema, descritto dalla coppia (T_i, n_i) (rispettivamente i tempi in cui si verifica un evento ingresso/uscita e il numero di utenti in coda). La dinamica del

sistema è schematizzabile nel modo seguente:

$$\text{se } t_i^{(in)} < t_i^{(out)} \begin{cases} T_{i+1} = T_i + t_i^{(in)} \\ n_{i+1} = n_i + 1 \\ t_{i+1}^{(in)} = 0 \\ t_{i+1}^{(out)} = t_i^{(out)} - t_i^{(in)} \end{cases} \quad (3.1)$$

$$\text{se } t_i^{(out)} < t_i^{(in)} \begin{cases} T_{i+1} = T_i + t_i^{(out)} \\ n_{i+1} = n_i - 1 \\ t_{i+1}^{(in)} = t_i^{(in)} - t_i^{(out)} \\ t_{i+1}^{(out)} = 0 \end{cases} \quad (3.2)$$

Si noti che la gestione di $t_i^{(in)}$ e $t_i^{(out)}$ secondo le Eq.3.1 e 3.2 (in aggiunta a controlli affinché $n_i \geq 0$) è necessaria perché gli ingressi e le uscite siano effettivamente dei processi indipendenti. Inoltre è possibile tenere traccia dei tempi di attesa per utente se si aggiungono apposite variabili $n_i^{(in)}$ (numero totale di arrivi al tempo T_i) e tempo totale di attesa $T_i^{(w)}$ (che è dato dall'integrale $\int n(t)dt$).

Esercizio 3.2: Eseguendo coda (script che lancia `runcoda (m,lambda,mu,sigma,n0)`), studiare il comportamento di $n(t)$, $\langle n \rangle(t)$ e $T^{(w)}(t)$ per diversi valori dei parametri λ , $\langle t_i^{(out)} \rangle$ (μ) e $\sigma(t_i^{(out)})$ (σ). Verificare la distribuzione dei tempi di ingresso e di uscita (contenuti rispettivamente nei vettori `Texp` e `Tlog`). Considerare i casi $\lambda \langle t_i^{(out)} \rangle < 1$ e $\lambda \langle t_i^{(out)} \rangle > 1$. Considerare il caso limite $\sigma(t_i^{(out)}) = 0$.

Un risultato fondamentale della teoria delle code (formula di Little) è che

$$\lambda \langle T_w \rangle = \langle n_w \rangle \quad (3.3)$$

cioè il numero medio di utenti in attesa (coda più servizio) è uguale al tempo di attesa per utente per la velocità di ingresso di nuovi utenti. Ciò vale pure per le quantità relative alla coda vera e propria (escludendo quindi i tempi di servizio e gli utenti serviti), cioè $\lambda \langle T_q \rangle = \langle n_q \rangle$. Poiché $\langle T_w \rangle = \langle T_q \rangle + \langle t_i^{(out)} \rangle$ si ottiene che $\langle n_w \rangle = \langle n_q \rangle + \lambda \langle t_i^{(out)} \rangle$. La quantità $u = \lambda \langle t_i^{(out)} \rangle$ può essere considerata il "carico" sul sistema. Infine, per una coda M/G/1 con $u < 1$ vale la formula di Pollackzek-Kinchine per $\langle n_q \rangle$:

$$\langle n_q \rangle = \frac{\lambda^2 \sigma(t_i^{(out)})^2 + u^2}{2(1 - u)} \quad (3.4)$$

Esercizio 3.3: Utilizzando ancora lo script coda, verificare le formule 3.3 e 3.4 per $u < 1$. Osservare in particolare i grafici di $\langle n \rangle(t)$ e $T^{(w)}(t)$. Osservare cosa avviene per $u \geq 1$.

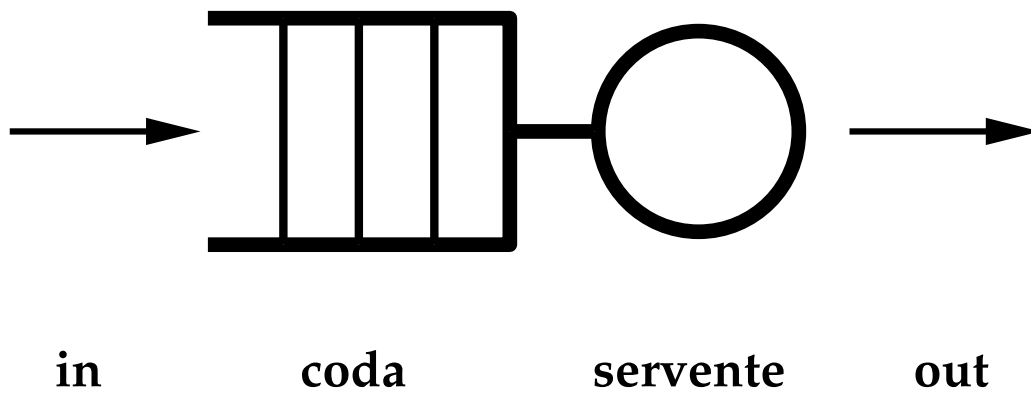


Figura 3.1: Schematizzazione di una coda a un ingresso e un servente.

Capitolo 4

Ottimizzazione

4.1 Ottimizzazione combinatoria: il problema TSP (“commesso viaggiatore”)

Il celebrato problema del “commesso viaggiatore” (TSP – *Traveling Salesman Problem*) consiste nel determinare il cammino minimo che congiunga in un percorso chiuso n punti dati. Si è dimostrato che TSP appartiene alla classe dei “problemi NP” (non polinomiali), cioè in pratica il tempo di soluzione cresce più velocemente di un polinomio all’aumentare della grandezza del problema (per TSP la grandezza del problema è determinata dal numero di punti da collegare).

In generale, un problema di ottimizzazione combinatoria consiste nel determinare la configurazione $\{\pi_i\}$ per cui una funzione obiettivo $\Phi(\{\pi_i\})$, a valori reali, raggiunge il valore ottimale (massimo o minimo a seconda dei casi). La configurazione $\{\pi_i\}$ è descritta da un set di k parametri (con k che può essere molto grande in problemi realistici) che assumono valori discreti scelti su un insieme finito. Ad esempio nel caso TSP la configurazione è descritta da una permutazione di n oggetti e la Φ è data dalla somma delle distanze euclidee tra punti successivi secondo la permutazione stessa. Il metodo di ricerca della configurazione ottima basato sulla forza bruta (metodo *esaustivo*), cioè calcolando Φ su tutte le permutazioni possibili, è impronibile in pratica per n di qualche decina. Infatti le permutazioni di n oggetti sono $n! \approx \sqrt{2\pi n} n^n e^{-n}$. Occorre dunque far ricorso a metodi *euristici* che riducano fortemente la complessità combinatoria del problema. Di tali metodi ne esistono sia deterministici che stocastici.

4.2 Ottimizzazione stocastica: il metodo SA (“raffreddamento simulato”)

I metodi di ottimizzazione stocastica si possono considerare come dei particolari metodi Montecarlo. L’idea è quella di utilizzare elementi di casualità per esplorare “creativamente” lo spazio delle configurazioni $\{\pi_i\}$ secondo una opportuna euristica.

4.2.1 Random search

L'idea più semplice è quella di esplorare lo spazio delle configurazioni in modo casuale e uniforme, tenendo nota dell'ottimo via via raggiunto.

Esercizio 4.1: Su un insieme di n punti casuali in 2D compresi in $[0, 1]$, scelti con $XY = \text{points}(n)$, generare cammini con $\text{Ind} = \text{permuta}(n)$ e calcolarne la lunghezza con $d = \text{distance}(XY)$. Utilizzare $[\text{Ind}, d] = \text{runTSP1}(XY, m)$ per cercare il minimo cammino dopo m tentativi.

4.2.2 Visite di Markov

Un miglioramento della strategia di ricerca completamente casuale prevede che le configurazioni vengano visitate in modo appropriato. In altre parole, due configurazioni successive non sono indipendenti tra loro, ma legate da opportune probabilità di transizione di Markov.

Nel caso TSP, si può pensare di modificare una configurazione mediante una "mossa locale": un nuovo cammino è generato dal precedente invertendo l'ordine di visita di due punti successivi, ed è accettato se la Φ diminuisce. Si possono progettare mosse più globali per generare nuovi cammini con maggior "fantasia", cioè esplorando lo spazio delle configurazioni in modo più efficiente. Ad esempio, si può considerare un sottocammino e invertirlo.

Esercizio 4.2: Sugli stessi punti dell'Eserc.4.1, generare nuovi cammini con move al posto di permuta in $\text{runTSP1}(XY, m)$, accettando la nuova configurazione solo se diminuisce il percorso totale. Provare con move2 .

4.2.3 Algoritmo di Metropolis

Gli algoritmi precedenti accettano una nuova configurazione se e solo se la Φ è migliore. La discesa verso il cammino minimo potrebbe però condurre verso una zona dello spazio delle configurazioni con un minimo locale, maggiore del minimo globale che è quello che invece interessa. L'idea dell'algoritmo di Metropolis, che è nato per descrivere sistemi della meccanica statistica, è quella di adottare la seguente prescrizione:

$$\Phi^{(old)} \rightarrow \Phi^{(new)} \text{ con probabilità } \begin{cases} p = 1 & \text{se } \Phi^{(new)} < \Phi^{(old)} ; \\ p = \exp[-(\Phi^{(new)} - \Phi^{(old)})/T] & \text{se } \Phi^{(new)} > \Phi^{(old)} . \end{cases} \quad (4.1)$$

dove T è interpretabile come una "temperatura". Se T è grande rispetto ai $\Delta\Phi$ tipici delle mosse adottate, saranno visitate frequentemente configurazioni con Φ relativamente grande (il sistema si

“agita”). Se viceversa T è piccolo, i cammini saranno disturbati dal rumore termico solo raramente (al limite si ricade nell’algoritmo della sezione precedente) (vedi Fig.4.1).

Esercizio 4.3: Ancora sugli stessi punti dell’Eserc.4.1, generare nuovi cammini con `move2` e accettarli secondo l’algoritmo di Metropolis (`flag = metrop(E1,E2,T)`) a temperatura costante. Utilizzare `runtsp2(XY,m,T)` per cercare il minimo cammino dopo m tentativi. Cosa avviene per $T = 0$?

4.2.4 Raffreddamento simulato

L’idea del “raffreddamento simulato” (SA – *Simulated Annealing*) è quella di sfruttare l’algoritmo di Metropolis, ma variando la temperatura in modo opportuno nel corso della simulazione. Si può pensare di partire con una temperatura elevata, per esplorare efficientemente lo spazio delle configurazioni “in grande”, e poi ridurla lentamente per lasciar “riposare” il sistema in un bacino di attrazione alla ricerca del minimo di Φ .

Esercizio 4.4: Come per l’Eserc.4.3, ma con temperatura variabile. Utilizzare `runtsp3(XY,m,T0)` (raffreddamento lineare) e `runtsp4(XY,m,T0)` (raffreddamento esponenziale) per cercare il minimo cammino dopo m tentativi.

Esistono delle euristiche generali per progettare l’andamento di $T(t)$ nel modo più efficiente possibile, in ogni caso tale scelta è critica per il successo del metodo.

Esercizio 4.5: Analogamente all’Eserc.4.4, provare a raffreddare il sistema con una funzione esponenziale a gradino (`runtsp5(XY,m,k,T0)`). Confrontare infine tutti i metodi sullo stesso set di punti e con lo stesso numero di tentativi.

Figura 4.1: Cammini per il problema del commesso viaggiatore con 400 città ottenuti con l'algoritmo di Metropolis (da Bonomi e Lutton). Temperatura decrescente da a) a d).

Bibliografia

- [1] E. Bonomi, J.-L. Lutton, "The N-City Travelling Salesman Problem: Statistical Mechanics and the Metropolis Algorithm", *SIAM Review*, **26**, n.4 (Oct 1984), pp.551-568.
- [2] J. M. Hammersley, D. C. Handscomb, *Monte Carlo Methods*, Methuen & Co., 1964.
- [3] M. H. Kalos, P. A. Whitlock, *Monte Carlo Methods. Vol I: Basics*, Wiley, 1986.
- [4] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi, "Optimization by simulated annealing", Res.Rep. RC 9335, IBM Thomas J.Watson Center, Yorktown Heights, NY, 1982.
- [5] J. Kohlas, *Stochastic methods of operations research*, Cambridge Univ. Pr., 1982.
- [6] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, E. Teller, "Equation of state calculations by fast computing machines", *J. Chem. Phys.*, **21** (1953), pp.1087-1092.
- [7] A. Papoulis, *Probability, Random variables, and Stochastic Processes*, McGraw-Hill, 2nd ed., 1994.
- [8] W. H. Press, B. P. Flannery, S. A. Teukolski, W. T. Vetterling, *Numerical Recipes*, Cambridge Univ. Pr., 2nd ed., 1992.
- [9] E. Wentzel, L. Ovcharov, *Applied Problems in Probability Theory*, Mir Publishers, 1986.

Appendice A

Listati degli script Matlab

I file sorgente sono disponibili nella directory `/u/cmn/pub/MC/matlab`.

A.1 Capitolo 1

A.1.1 `ranmult.m`

```
function X = ranmult(n)
m=2^31;
a=2^16 + 3;
Y(1,1) = 372683;
for i=2:n;
    Y(i,1) = rem(a*Y(i-1,1),m);
end;
X(:,1) = Y(:,1) / m;
return;
```

A.1.2 `autocorr.m`

```
function Y = autocorr(X)
n = size(X);
m = mean(X);
v = sum((X-m).^2);
temp = conv((X-m),(flipud(X)-m)) / v;
n1 = size(temp);
Y = temp(n:n1);
return;
```

A.1.3 `buffon.m`

```
function f = buffon(n,d,1)
X = 0.5*d*rand(n,1);
phi = pi*rand(n,1);
for i=1:n
```

```

    if X(i,1) <= 0.5*1*sin(phi(i,1))
        Y(i,1) = 1;
    else
        Y(i,1) = 0;
    end
end
end
f = sum(Y);
return;

```

A.1.4 expdist.m

```

function X = expdist(n,lambda)
X = rand(n,1);
X = -(1.0/lambda)*log(X);
return;

```

A.1.5 poisson.m

```

function X = poisson(n,lambda)
for i=1:n
    m = 0.0;
    count = 0;
    while m < 1.0,
        m = m + expdist(1,lambda);
        count = count+1;
    end;
    X(i,1) = count-1;
end
return;

```

A.1.6 norm1.m

```

function Y = norm1(n)
Y(1:n,1) = zeros(n,1);
for i=1:12;
    Y = Y + rand(n,1);
end;
Y = Y - 6.0*ones(n,1);
return;

```

A.1.7 norm2.m

```

function [Y1,Y2] = norm2(n)
X1 = rand(n,1);
X2 = rand(n,1);
R = sqrt(-2*log(X1));

```

```

Y1 = R .* cos(2*pi*X2);
Y2 = R .* sin(2*pi*X2);
return;

```

A.1.8 norm3.m

```

function [Y1,Y2] = norm3(n)
for i=1:n
    r2 = 999.0;
    while (r2 >= 1.0 | r2 == 0.0),
        x1 = 2.0*rand-1.0;
        x2 = 2.0*rand-1.0;
        r2 = x1^2 + x2^2;
    end;
    fac = sqrt(-2.0*log(r2)/r2);
    Y1(i,1) = fac*x1;
    Y2(i,1) = fac*x2;
end;
return;

```

A.2 Capitolo 2

A.2.1 cerchio.m

```

function y = cerchio(x)
y = sqrt(1.0 - x.^2);
return;

```

A.2.2 hitmiss.m

```

function [f, df] = hitmiss(n)
X = rand(n,1);
Y = rand(n,1);
Q = cerchio(X);
for i=1:n
    if Y(i,1) <= Q(i,1);
        Z(i,1) = 1;
    else
        Z(i,1) = 0;
    end
end
f = mean(Z);
df = sqrt( f*(1.0-f) / n );
return;

```

A.2.3 mcnaive.m

```
function [f, df] = mcnaive(n)
X = rand(n,1);
Y = cerchio(X);
f = mean(Y);
df = sqrt((mean(Y.^2) - f^2)/n );
return;
```

A.2.4 funz.m

```
function y = funz(x,beta)
y = (1.0 - beta*x.^2)/(1.0 - beta/3.0);
return;
```

A.2.5 campiona.m

```
function Z = campiona(n,beta)
fact = 1.0 / (1.0 - beta/3.0);
for i=1:n
    x = 999.0;
    y = 999.0;
    while y >= funz(x,beta),
        x = rand;
        y = fact*rand;
    end;
    Z(i,1) = x;
end;
return;
```

A.2.6 mcimport.m

```
function [f, df] = mcimport(n,beta)
X = campiona(n,beta);
Y = cerchio(X);
Z = funz(X,beta);
Q = Y./Z;
f = mean(Q);
df = sqrt((mean(Q.^2) - f^2)/n );
return;
```

A.3 Capitolo 3

A.3.1 funz.m

```
function y = funz(x,mu,sigma)
```

```

mul = 2*log(mu) - 0.5*log(mu^2+sigma^2);
signal = sqrt(log(mu^2+sigma^2) - 2*log(mu));
y = (1.0/sqrt(2*pi))*(1.0/signal)*exp(-(log(x)-mul).^2/(2.0*signal^2));
return;

```

A.3.2 lognorm.m

```

function Y = lognorm(n,mu,sigma)
mul = 2*log(mu) - 0.5*log(mu^2+sigma^2);
signal = sqrt(log(mu^2+sigma^2) - 2*log(mu));
X = mul + signal*randn(n,1);
Y = exp(X);
return;

```

A.3.3 theta.m

```

function h = theta(x)
if x > 0
    h = x;
else
    h = 0;
end;
return;

```

A.3.4 runcoda.m

```

function [T, n, nm, Tw, Texp, Tlog] = runcoda(m,lambda,mu,sigma,n0)
% inizializzazioni
T0 = 0.0;
T(1,1) = T0;
n(1,1) = n0;
ninw(1,1) = theta(n0-1);
nin = 0;
nout = 0;
Tinold = 0.0;
Toutold = 0.0;
% main loop
for i=2:m
    ninw(i,1) = ninw(i-1,1);
% caso degenero: coda vuota
    if n(i-1,1) <= 0
        Tin = expdist(1,lambda);
        nin = nin + 1;
        Texp(nin,1) = Tin;
        T(i,1) = T(i-1,1) + Tin;
        n(i,1) = 1;
    end
end

```

```

    Tinold = 0.0;
    Toutold = 0.0;
% coda non vuota
    else
% check generazione tempi casuali
    if Tinold == 0.0
        Tin = expdist(1,lambda);
        nin = nin + 1;
        Texp(nin,1) = Tin;
% nuovo arrivo in coda
        ninw(i,1) = ninw(i-1,1) + 1;
    else
        Tin = Tinold;
    end;
    if Toutold == 0.0
        Tout = lognorm(1,mu,sigma);
        nout = nout + 1;
        Tlog(nout,1) = Tout;
    else
        Tout = Toutold;
    end;
% check tempi
    if Tin == Tout
        T(i,1) = T(i-1,1) + Tin;
        n(i,1) = n(i-1,1);
        Tinold = 0.0;
        Toutold = 0.0;
    elseif Tin < Tout
        T(i,1) = T(i-1,1) + Tin;
        n(i,1) = n(i-1,1) + 1;
        Tinold = 0.0;
        Toutold = Tout - Tin;
    else
        T(i,1) = T(i-1,1) + Tout;
        n(i,1) = n(i-1,1) - 1;
        Tinold = Tin - Tout;
        Toutold = 0.0;
    end;
end;
end;
% generazione n medio
nm(1,1) = n(1,1);
for i=2:m
    nm(i,1) = ((T(i-1,1)-T(1,1))*nm(i-1,1) + n(i-1,1)*(T(i,1)-T(i-1,1)))
        / (T(i,1)-T(1,1));
end;

```

```

% generazione tempo di attesa medio
Twsuold = 0.0;
Tw(1,1) = 0.0;
for i=2:m
    Twsum = Twsumold + (T(i,1)-T(i-1,1))*theta(n(i-1,1));
    if ninw(i-1,1) > 0.0
        Tw(i,1) = Twsum / ninw(i,1);
    else
        Tw(i,1) = 0.0;
    end;
    Twsumold = Twsum;
end;
return;

```

A.3.5 coda.m

```

m = input('numero eventi ? ');
lambda = input('frequenza ingressi poissoniani ? ');
mu = input('tempi di servizio lognormali, media ? ');
sigma = input('tempi di servizio lognormali, sqm ? ');
n0 = input('num. utenti in coda alla partenza ? ');
tic;
[T, n, nm, Tw, Texp, Tlog] = runcoda(m,lambda,mu,sigma,n0);
toc;
% output tempi
disp('media, sqm tempi ingresso');
disp(mean(Texp));
disp(std(Texp));
hist(Texp);
pause;
disp('media, sqm tempi uscita');
disp(mean(Tlog));
disp(std(Tlog));
hist(Tlog);
pause;
disp('tempo finale');
disp(dis(T(m)));
% output n(t), n medio
disp('numero max utenti in coda');
disp(max(n));
disp('media finale utenti in coda');
disp(nm(m));
plot(T,n,'y',T,nm,'w');
pause;
% output tempi attesa
disp('tempo di attesa medio in coda');

```

```

disp(Tw(m));
plot(T,Tw);
pause;
% carico medio
u = lambda*mu;
% formula di Pollackzek-Kinchine
pk = u + 0.5*(lambda^2*sigma^2 + u^2)/(1.0-u);
% formula di Pollackzek-Kinchine e Little
pk1 = pk/lambda;
disp('carico medio');
disp(u);
disp('numero medio in coda - formula di Pollackzek-Kinchine');
disp(pk);
disp('tempo di attesa medio in coda - formula di P-K e Little');
disp(pk1);

```

A.4 Capitolo 4

A.4.1 points.m

```

function X = points(n)
X = rand(n,2);
return;

```

A.4.2 distance.m

```

function d = distance(X)
d = 0.0;
n = size(X,1);
for i=2:n
    d = d + (X(i,1)-X(i-1,1))^2 + (X(i,2)-X(i-1,2))^2;
end;
d = d + (X(1,1)-X(n,1))^2 + (X(1,2)-X(n,2))^2;
d = sqrt(d);
return;

```

A.4.3 permuta.m

```

function Ind = permuta(n)
X = rand(n,1);
[Y Ind] = sort(X);
return;

```

A.4.4 move.m

```

function Ind = move(Ind0)

```

```

n = size(Ind0,1);
pivot1 = fix(n*rand);
pivot2 = rem(pivot1+1,n);
pivot1 = pivot1 + 1;
pivot2 = pivot2 + 1;
i1 = Ind0(pivot1,1);
i2 = Ind0(pivot2,1);
Ind = Ind0;
Ind(pivot1,1) = i2;
Ind(pivot2,1) = i1;
return;

```

A.4.5 move2.m

```

function Ind = move2(Ind0)
n = size(Ind0,1);
pivot1 = fix(n*rand) + 1;
pivot2 = fix(n*rand) + 1;
while pivot2 == pivot1
    pivot2 = fix(n*rand) + 1;
end;
if pivot1 < pivot2
    np = pivot2-pivot1+1;
    tmp(1:np,1) = Ind0(pivot1:pivot2,1);
else
    np1 = n-pivot1+1;
    np2 = pivot2;
    np = np1+np2;
    tmp(1:np1,1) = Ind0(pivot1:n,1);
    tmp(np1+1:np,1) = Ind0(1:pivot2,1);
end;
tmp = flipud(tmp);
Ind = Ind0;
if pivot1 < pivot2
    Ind(pivot1:pivot2,1) = tmp(1:np,1);
else
    Ind(pivot1:n,1) = tmp(1:np1,1);
    Ind(1:pivot2,1) = tmp(np1+1:np,1);
end;
return;

```

A.4.6 metrop.m

```

function flag = metrop(E1,E2,T)
if E2 < E1
    flag = 1;

```

```

else
    p = exp(-(E2-E1)/T);
    x = rand;
    if x < p
        flag = 1;
    else
        flag = 0;
    end;
end;
return;

```

A.4.7 runtsp1.m

```

function [Indmin, d] = runtsp1(XY,m)
% random search
d0 = distance(XY);
d(1,1) = d0;
dmin = d0;
n = size(XY,1);
tmp = ones(n,1);
Indmin = conv(tmp,tmp);
Indmin = Indmin(1:n,1);
for i=2:m
%
% select shuffling
    Ind = permuta(n);
% Ind = move(Indmin);
% Ind = move2(Indmin);
%
    PXY = XY(Ind,:);
    d(i,1) = distance(PXY);
    if d(i,1) < dmin
        dmin = d(i,1);
        Indmin = Ind;
    end;
end;
return;

```

A.4.8 runtsp2.m

```

function [Indmin, d] = runtsp2(XY,m,T)
% Metropolis algorithm with constant T
d0 = distance(XY);
d(1,1) = d0;
dmin = d0;
n = size(XY,1);

```

```

tmp = ones(n,1);
Ind = conv(tmp,tmp);
Ind = Ind(1:n,1);
Indmin = Ind;
for i=2:m
    Indnew = move2(Ind);
    PXY = XY(Indnew,:);
    dnew = distance(PXY);
    flag = metrop(d(i-1,1),dnew,T);
    if flag == 1
        d(i,1) = dnew;
        Ind = Indnew;
    else
        d(i,1) = d(i-1,1);
    end;
    if d(i,1) < dmin
        dmin = d(i,1);
        Indmin = Ind;
    end;
end;
return;

```

A.4.9 runtsp3.m

```

function [Indmin, d] = runtsp3(XY,m,T0)
% Metropolis algorithm with linearly decreasing T
d0 = distance(XY);
d(1,1) = d0;
dmin = d0;
T = T0;
dT = T0 / (m-1);
n = size(XY,1);
tmp = ones(n,1);
Ind = conv(tmp,tmp);
Ind = Ind(1:n,1);
Indmin = Ind;
for i=2:m
    T = T - dT;
    Indnew = move2(Ind);
    PXY = XY(Indnew,:);
    dnew = distance(PXY);
    flag = metrop(d(i-1,1),dnew,T);
    if flag == 1
        d(i,1) = dnew;
        Ind = Indnew;
    else

```

```

    d(i,1) = d(i-1,1);
end;
if d(i,1) < dmin
    dmin = d(i,1);
    Indmin = Ind;
end;
end;
return;

```

A.4.10 runtsp4.m

```

function [Indmin, d] = runtsp4(XY,m,T0)
% Metropolis algorithm with exponentially decreasing T
d0 = distance(XY);
d(1,1) = d0;
dmin = d0;
T = T0;
Tend = 1.0e-3 * d0;
alfa = log(Tend/T0)/(m-1);
alfa = exp(alfa);
n = size(XY,1);
tmp = ones(n,1);
Ind = conv(tmp,tmp);
Ind = Ind(1:n,1);
Indmin = Ind;
for i=2:m
    T = T * alfa;
    Indnew = move2(Ind);
    PXY = XY(Indnew,:);
    dnew = distance(PXY);
    flag = metrop(d(i-1,1),dnew,T);
    if flag == 1
        d(i,1) = dnew;
        Ind = Indnew;
    else
        d(i,1) = d(i-1,1);
    end;
    if d(i,1) < dmin
        dmin = d(i,1);
        Indmin = Ind;
    end;
end;
return;

```

A.4.11 runtsp5.m

```
function [Indmin, d] = runtsp5(XY,m,k,T0)
% Metropolis algorithm with stepwise exponentially decreasing T
q = m/k;
d0 = distance(XY);
d(1,1) = d0;
dmin = d0;
T = T0;
Tend = 1.0e-3 * d0;
alfa = log(Tend/T0)/(q-1);
alfa = exp(alfa);
n = size(XY,1);
tmp = ones(n,1);
Ind = conv(tmp,tmp);
Ind = Ind(1:n,1);
Indmin = Ind;
for i=2:m
    if rem(i,k) == 0
        T = T * alfa;
    end;
    Indnew = move2(Ind);
    PXY = XY(Indnew,:);
    dnew = distance(PXY);
    flag = metrop(d(i-1,1),dnew,T);
    if flag == 1
        d(i,1) = dnew;
        Ind = Indnew;
    else
        d(i,1) = d(i-1,1);
    end;
    if d(i,1) < dmin
        dmin = d(i,1);
        Indmin = Ind;
    end;
end;
return;
```