Treball Final de Grau

GRAU DE MATEMÀTIQUES

Facultat de Matemàtiques i Informàtica
Universitat de Barcelona

GENERACIÓ DE NOMBRES ALEATORIS I LES SEVES APLICACIONS

Autor: Bernat Armengol Vall

Director: Dr. Àngel Jorba
Realitzat a: Departament de Matemàtiques i Informàtica

Barcelona, 27 de juny de 2018

Resum/Abstract

En aquest treball, com el seu nom indica, estudiarem la generació de nombres aleatoris, farem tests per verificar la seva validesa i veurem les seves aplicacions.
Per començar, en el primer capítol farem una breu introducció on, entre altres coses, definirem com són les successions aleatòries.

En el segon, començarem explicant els generadors de congruència lineal, congruència multiplicativa i diferents conceptes importants per després poder endinsar-nos a la generació de lleis uniformes. A continuació, estudiarem els exemples més usats: *estàndard mínim*, RANDU i el mètode de la barreja; els implementarem en llenguatge C i finalment comentarem altres mètodes importants.

En el tercer capítol, estudiarem els dos tests d'aleatorietat més coneguts que s'usen per detectar si una successió és de variables aleatòries indepedents amb llei uniforme: el test χ^2 i el test de Kolmogorov - Smirnov

Tot seguit, en el quart i el cinquè capítol, veurem diferents aplicacions de la generació de nombres aleatoris.
Llavors, estudiarem el mètode de Monte Carlo mitjançant un exemple senzill i aplicarem aquest mètode al càlcul d'integrals. A continuació, veurem dos mètodes distints per poder reduir la variància ja que és una manera de reduir l'error d'estimació d'aquest mètode.
Finalment, aplicarem la generació de nombres aleatoris en tècniques de simulació discreta per poder gestionar cues. Per poder estudiar aquestes tècniques de simulació de processos discrets, ho introduirem mitjançant un exemple senzill de la gestió d'una única cua i també definirem els conceptes d'esdeveniment i agenda. D'aquesta manera, podrem aplicar aquestes tècniques en aplicacions més complexes i amb més utilitat en la vida real.

In this project, as its name indicates, we will study the generation of random numbers by doing tests to verify its validity and by seeing its applications. Firstly, in the first chapter we will introduce the topic where, for example, we will define how the random sequences work.

In the second chapter we will start by explaining the generators of linear congruence, multiplicative congruence and different important concepts, then to study thoroughly the generation of the uniform distribution. To continue with, we will study the most used examples: *minimum standard*, RANDU and the method of shuffling; we will implement them in C language and finally we will discuss other important methods.

In the third chapter we will study the most common statistical tests used to detect whether a succession is of independent random variables with uniform distribution: the test χ^2 and the test of Kolmogorov – Smirnov.

Then, in the fourth and fifth chapter, we will see the different applications of the generation of random numbers.
After that, we will examine the Monte Carlo method through a simple example and we will apply this method to the calculation of integrals. Following on, we will see two different methods in order to reduce the variance since it is a way to reduce the estimation error of this method.
To conclude, we will apply the generation of random numbers in simulation techniques to handle queues. In order to study these simulation techniques, we will introduce it through a simple example of the management of a single queue and we will also define the concepts of event and agenda. In this way, we will be able to apply these techniques in more complex applications which are also more useful in real life.

2010 Mathematics Subject Classification. 65C05, 65C10, 68U20

Agraïments

En primer lloc, vull donar les gràcies al Dr. Àngel Jorba per la tutela i el suport constant durant la realització d'aquest treball. També, als amics i companys que m'han acompanyat en tot aquest viatge que ha sigut el grau de Matemàtiques. Finalment, no em vull oblidar de la meva família que ha fet tot això possible i els amics de sempre, que com el propi mot diu, sempre han estat allà.

Índex

Capítol 1

Introducció

Des de sempre, els éssers humans hem volgut saber que passaria en el futur i, fins i tot, mitjançant la informació que teníem a l'abast, hem intentat predir-ho. Per exemple, qui guanyaria el següent clàssic per emportar-nos la porra.

Així doncs, per predir esdeveniments futurs, utilitzem estadístiques basades en fets passats que, suposadament, en el futur també s'han de complir. Tanmateix, en molts casos, hi ha un ingredient aleatori que provoca que les variables a modelitzar siguin desconegudes per nosaltres. Per tant, per fer la simulació que estem interessats necessitem generar aquests nombres aleatoris.

Primerament, els estadístics s'encarreguen de recollir les dades, analitzar-les i intentar seleccionar les lleis aleatòries correctes. A partir d'aquí, hem de generar aquests nombres aleatoris que rigurosament compleixin la llei estadística concreta. Aquest pas de la simulació discreta és el qual tractarem durant els dos primers capítols.

Per altra banda, abans d'entrar en el contingut del nostre treball, farem una introducció explicant conceptes importants que necessitem de base per poder desenvolupar aquest treball. Aquests conceptes són contingut de dos assignatures del grau que estem finalitzant ([7] i [11]) i també són extrets de les referències bibliogràfiques [3] i [4].

1.1 Successions aleatòries

Definició 1.1.1. Tota experiència aleatòria té associat un espai de probabilitat. Un espai de probabilitat és una terna (Ω, $\mathscr{F}$, P) on Ω és l'espai mostral, és a dir, el conjunt de resultats possibles; $\mathscr{F}$ és una família de parts d'Ω amb estructura de σ-àlgebra i P és una aplicació que satisfà:
$\mathscr{F} \longrightarrow [0,1]$
$A \longmapsto P(A)$, que compleix:

- $P(\Omega) = 1$

- σ - additiva en $\{A_n, n \geq 1\} \subseteq \mathscr{F}$ disjunts 2 a 2.

Definició 1.1.2. Una variable aleatòria és una aplicació $X : \Omega \to \mathbb{R}$ on es compleix que $X^{-1}(B) = \{w \in \Omega : X(w) \in B\} \in \mathscr{F}$, $\forall B \in \mathscr{B}(\mathbb{R})$, on $\mathscr{F}$ és una família d'esdeveniments i $\mathscr{B}(\mathbb{R})$ són els borelians de $\mathbb{R}$.

Així doncs,

Definició 1.1.3. Una successió aleatòria n-dimensional és una aplicació $X : \Omega \to \mathbb{R}^n$, és a dir, $X = (X_1, X_2, \ldots, X_n)$, on cada componenent $X_i : \Omega \to \mathbb{R}$, $i \in \{1, \ldots, n\}$ és una variable aleatòria.

Definició 1.1.4. Direm que $x_1, x_2, \ldots, x_n$ variables aleatòries són independents si $\forall B_1, B_2, \ldots, B_n \in \mathscr{B}(\mathbb{R})$ es compleix

$$P(x_1 \in B_1, x_2 \in B_2, \ldots, x_n \in B_n) = \prod_{i=1}^{n} P(x_i \in B_i)$$

Una col·lecció infinita de variables aleatòries és independent si qualsevol subcol·lecció finita és independent.

A continuació, ens centrarem en la part computacional de les successions aleatòries ja que avui en dia aquestes no s'entenen sense la generació mitjançant un ordenador. Per tant, encararem el nostre treball cap aquest àmbit.

Les successions aleatòries han de complir les següentes propietats:

P1: La successió ha de seguir la llei estadística amb què treballem (en la següent secció estudiarem les lleis més importants que seran les que utilitzarem).

P2: No hi ha d'haver cap relació entre els termes de la successió, és a dir, les variables aleatòries de la successió han de ser independents entre elles. (definit anteriorment)

Així doncs, mitjançant l'ordenador és impossible crear successions aleatòries ja que no compleix la segona propietat, com a conseqüència de ser una màquina determinista que la seva funció és executar ordres preestablertes.

Llavors, com que nosaltres treballarem amb l'ordenador, volem successions que compleixin P1 i que "aparentment" compleixin P2, és a dir, que la relació entre els diferents membres de la successió sigui tant complexa que no es vegui fàcilment. Aquestes successions s'anomenen successions pseudoaleatòries.

Finalment doncs, farem un abús de llenguatge i com que sempre parlarem de successions pseudoaleatòries, les anomenarem successions aleatòries.

1.2 Lleis estadístiques

Tot seguit, estudiarem les lleis estadístiques més comunes. Tot i que no ens centrarem en la base teòrica ja que és temari de diferents assignatures del grau com ara [7] i [11], sinó que estudiarem la part computacional, és a dir, com generar aquestes lleis per poder-les implementar en diferents aplicacions que veurem en capítols més endavant.

Primer de tot, la llei més important i base de moltes de les altres lleis més utilitzades és la llei $U([0,1])$, la qual ja li dediquem el segon capítol íntegrement. A partir d'ella, estudiarem les lleis exponencials i normals. Per tant, en tota aquesta secció suposarem que la successió aleatòria $\{\alpha_n\}_n$ és obtinguda mitjançant una llei $U([0,1])$.

1.2.1 Llei exponencial

Aquesta llei és molt senzilla de generar, ja que la successió $\{e_n\}_n$ definida per

$$\{e_n\}_n = -\frac{ln\{\alpha_n\}}{\lambda}$$

és una successió aleatòria que segueix una llei exponencial de paràmetre λ, la demostració de la qual la podreu trobar en [1], [4] o [10].

Com que és així de senzilla de generar, conseqüentment la seva implementació també és trivial, com podreu veure en el capítol 5 que l'utilitzarem per la gestió de cues. L'única observació remarcable és que hem de vigilar el valor "0" ja que utilitzem la funció ln, per això, en el capítol següent explicarem una variant per evitar aquest problema.

1.2.2 Llei normal

Per poder generar una successió aleatòria que segueixi una llei normal a partir de la llei uniforme, ho podem fer mitjançant diferents mètodes. Nosaltres ho farem mitjançant un dels més coneguts, el mètode Box- Muller que s'implementa així:

Sigui $\{\alpha_n\}_n$ una successió que segueix la llei $U([0,1])$. Llavors, definim la successió $\{(x_n, y_n)\}_n$ com:

$$x_n = \cos(2\pi\alpha_{2n})\sqrt{-2ln\alpha_{2n-1}},$$
$$y_n = \sin(2\pi\alpha_{2n})\sqrt{-2ln\alpha_{2n-1}}$$

Per tant, la successió $x_1, y_1, x_2, y_2, \ldots, x_n, y_n$ segueix una llei $N([0,1])$, és a dir, mitjana 0 i variància 1. La demostració d'aquest mètode la podeu trobar en [1], [4] i [10].

A més, sabem per teoria, que a partir d'una llei $N([0,1])$, podem definir qualsevol altra llei normal. Així doncs, si $\{\beta_n\}_n$ és una successió aleatòria que segueix una llei $N([0,1])$, llavors la successió aleatòria $\{\gamma_n\}_n$ definida com $\gamma_n = \sigma\beta_n + \mu$ segueix una llei normal de mitjana μ i variància σ^2.

Finalment, si voleu més informació sobre aquestes lleis no uniformes o estudiar-ne d'altres, podeu cercar en aquestes referències [1], [4] i [10].

Capítol 2

Generació de lleis uniformes

Primerament, farem una introducció mitjançant els generadors de congruència lineal, congruència multiplicativa i diferents conceptes importants per després poder endinsar-nos a la generació de lleis $U([0,1])$. A continuació, estudiarem els exemples més usats i els implementarem en llenguatge C i finalment, comentarem altres mètodes.

Durant aquest capítol, les principals fonts bibliogràfiques utitlitzades en la realització del treball han estat [3] i [4]. També, part de la informació complementària serà extreta de [5] i en la informació més específica ja ho detallarem en cada cas.

2.1 Generació de congruència lineal

Definició 2.1.1. Definim *generador de congruència lineal* la fòrmula:

$$X_i = (aX_{i-1} + c) \mod m, \tag{2.1.1}$$

on m, a i c són enters positius tals que $m < \max\{a, c\}$.

Per tant, aquest mètode funciona d'aquesta manera: Primer de tot, triem un X_0 enter tal que $0 \leq X_0 < m$. Aquest valor X_0 l'anomenarem llavor ja que és el valor inicial d'una successió. A partir d'ell i aplicant la fòrmula obtenim un valor X_1. Fem el mateix amb el valor X_1 i obtenim X_2, i així successivament.

Exemple 2.1.2.

$$X_i = (5X_{i-1} + 2) \mod 8, \tag{2.1.2}$$

on la llavor és $X_0 = 5$. Llavors apliquem el mètode i obtenim: $X_1 = 3$, $X_2 = 1$, $X_3 = 7$, $X_4 = 5 \ldots$

Ara mitjançant l'exemple, veurem si els generadors de congruència lineal semblen bons generadors de nombres enters aleatoris. Hem de veure si la successió generada

a partir de l'exemple, és una successió aleatòria d'enters del 0 al 7. Així doncs, desenvolupem la successió:

$$5, 3, 1, 7, 5, 3, 1, 7, 5, 3, \ldots$$

Primerament, veiem que és una successió periòdica, és a dir, hi ha un període que la successió repeteix infinitament. Per tant, ja no compleix la P2. Després, veiem que tampoc genera tots els enters del 0 al 7, doncs tampoc compleix la P1. En definitiva, aquest exemple no és un bon generador de nombres enters aleatoris.

En general, tots els generadors de congruència lineal són periòdics i, com que la generació es basa en la fòrmula lineal, cada nombre està determinat per l'anterior. Per tant, mai compliran P2. Llavors, el que ens interessa és que compleixin P1 i per això, busquem generadors amb m gran i període màxim.

Definició 2.1.3. Una *ratxa* és la repetició d'un valor en una successió aleatòria.

En les successions generades per congruència lineal, o no tenim cap ratxa o quan apareix és infinita. Clarament, aquesta propietat és un altre problema que tenen els generadors de congruència lineal ja que si la ratxa és infinita, és trivial que la successió no és aleatòria.
Per altra banda, si no apareix cap ratxa tampoc es compleix l'aleatorietat ja que si ens fixem en l'exemple anterior, que es produeixi una ratxa d'un valor en algun terme de la successió té una probabilitat de $\frac{1}{64}$, és a dir, hi ha $\frac{1}{8}$ de probabilitat que surti un valor i un $\frac{1}{8}$ de que es repeteixi. Per tant, en algun pas de la successió infinita s'hauria de produir i com que no és el cas, et demostra que la successió en la qual no es produiex cap ratxa, no es comporta com una successió aleatòria.

Quan $c = 0$, tenim un cas particular dels generadors de congruència lineal que definim:

2.1.1 Generació de congruència multiplicativa

Definició 2.1.4. Anomenem *generador de congruència multiplicativa* la fòrmula:

$$X_i = (aX_{i-1}) \mod m \tag{2.1.3}$$

Aquests nous generadors tenen dos principals diferències amb els anteriors. La primera és que els de congruència multiplicativa tenen un greu problema amb el valor 0, el qual els de congruència lineal no tenien. Aquest inconvenient és que si apareix el 0, la successió ja s'estaciona en aquest número infinitament (ho veiem trivialment). L'altra és que els generadors de congruència multiplicativa s'executen una mica més ràpid que els de congruència lineal ja que ens estalviem una suma.

A continuació, deixarem els nombres enters i entrarem als nombres reals amb la llei estadística $U([0, 1])$ que, com hem comentat anteriorment, és la llei en que ens basarem principalment durant tot el treball.

2.2 Condició d'aleatorietat

Sigui $\{x_n\}_n = \{x_1, x_2, ...\}$ una successió generada a partir d'una llei $U([0,1])$, per tal que compleixi la propietat P1, ha de verificar:

A1: Els números $x_1, x_2, x_3, ...$ estan uniformement distribuïts a l'interval $[0,1]$.

A2: Les parelles (x_1, x_2),(x_2, x_3),$(x_3, x_4), \cdots$ estan uniformement distribuïdes al quadrat $[0,1]^2$.

$$\vdots$$

AN: Les tuples $(x_1, x_2, ..., x_n), (x_2, x_3, ..., x_{n+1}), ...$ estan uniformement distribuïdes a $[0,1]^n$.

$$\vdots$$

Comentem una mica aquestes condicions. La condició 1 significa que els termes de la successió "recobreixen" uniformement tot l'interval $[0,1]$, mentres que la condició 2, les parelles $(x_1, x_2), (x_2, x_3), (x_3, x_4)$, etc. "recobreixen" uniformement tot el quadrat unitat $[0,1]^2$. Ara veurem mitjançant un exemple que si es compleix A1 no implica que es compleixi A2:

Exemple 2.2.1. Siguin $\{a_n\}_n$ i $\{b_n\}_n$ successions en $U([0,1/2])$ i $U([1/2,1])$. Definim una nova successió $\{c_n\}_n$:

$$c_n = \begin{cases} a_{\frac{n}{2}}, \text{ si } n \text{ és parell} \\ b_{\frac{n+1}{2}}, \text{ si } n \text{ és senar} \end{cases}$$

És a dir, $c_1 = b_1$, $c_2 = a_1$, $c_3 = b_2$, $c_4 = a_2$,...
Es veu trivialment que aquesta successió compleix A1. Anem a veure si compleix A2: agafem les parelles $(c_1, c_2), (c_2, c_3), (c_3, c_4)$,etc. que són $(b_1, a_1), (a_1, b_2), (b_2, a_2)$,etc. Per tant, aquesta successió només cobreix el conjunt $\left[0, \frac{1}{2}\right] \times \left[\frac{1}{2}, 1\right] \cup \left[\frac{1}{2}, 1\right] \times \left[0, \frac{1}{2}\right]$ i no compleix A2. A més, és evident la correlació de nombres ja que després d'un més gran que $\frac{1}{2}$ sempre el segueix un nombre més petit que $\frac{1}{2}$.

Doncs, ja hem vist via l'exemple que una successió compleixi A1, A2,..., AN no implica que verifiqui la condició N+1.

2.3 Generació de lleis $U([0,1])$

Suposem que tenim una successió X_n de nombres enters aleatoris compresos entre 0 i $m-1$ amb distribució uniforme. Llavors, la successió $\{y_n\}_n$ definida com:

$$y_n = \frac{X_n}{m}, n = 0, 1, 2, ...,$$

és una successió dins l'interval $[0, 1]$ però veiem que només tenen valors dins del conjunt

$$Y_m = \{0, \frac{1}{m}, \frac{2}{m}, ..., \frac{m-1}{m}\}.$$

Així doncs, teòricament, no podem dir que la successió y_n segueixi una llei $[0, 1]$. No obstant, com que treballem amb un ordenador i estem treballant amb variables reals float que tenen 7 decimals, si m és 10^7 o més gran llavors en el nostre ordenador Y_m és el mateix que el conjunt $[0, 1]$.
Tot i que el valor "1" no és mai generat en aquesta successió, si realment volem que aparegui aquest valor, es pot corregir fàcilment definint

$$y_n = \frac{X_n + 1}{m}.$$

Ara, en aquesta successió el valor que no es genera mai és "0" i això ens pot interessar quan volem fer el logaritme d'una $U([0, 1])$, com podria ser el cas quan implementarem la llei exponencial en el cinquè capítol.

En resum, triarem un valor m prou gran per poder treballar amb variables 7 decimals. A més, si escollim els valors m i a de forma adequada, podem aconseguir generadors de congruència multiplicativa de la mateixa qualitat que de congruència lineal i, com que els de congruència multiplicativa són més ràpids ja que ens estalviem una suma, com ja hem dit anteriorment, ens centrarem exclusivament en aquests.

2.4 Exemples concrets

En aquest apartat parlarem de diferents exemples concrets de generadors que s'han usat durant anys i la seva implementació en C. Més específicament, ens centrarem en tres mètodes: l'*estàndard mínim*, el RANDU i el mètode de la barreja.

2.4.1 Estàndard mínim

Començarem per l'exemple clàssic anomenat *estàndard mínim*:

Exemple 2.4.1.

$$X_i = (16807X_{i-1}) \mod 2147483647 \tag{2.4.1}$$

on, com veieu, $a = 7^5 = 16807$, $m = 2^{31} - 1 = 2147483647$ i té període màxim ja que m és un nombre primer. Aquest generador es va proposar per primera vegada a [6] i s'estudia moltes altres vegades (consulteu [8]).

Com que el període és m, podem prendre valors de 0 a $m-1$, i amb nombres grans propers a $m-1$ si fem el producte $16807X_{i-1}$ poden ser més grans que $2^{31}-1$. Fent memòria, recordem que en C les variables enteres més grans són les **long int** i sabem que emmagatzemen nombres enters entre -2^{31} i $2^{31}-1$. Per tant, tenim un problema i per resoldre'l utilitzarem la proposició següent:

Proposició 2.4.2. *Sigui*

$$q = \left[\frac{m}{a}\right], r = m \ mod \ a,$$

és a dir, $m = aq + r$. *Sigui* $x \in \{1, 2, ..., m-1\}$. *Aleshores, si* $r < q$,

1. *Els valors* $a\,(x \ mod \ q)$ *i* $r\left[\frac{x}{q}\right]$ *estan compresos entre* 0 *i* $m-1$ *(els dos inclosos).*
2. *Definim* $u = a\,(x \ mod \ q) - r\left[\frac{x}{q}\right]$.

Llavors,

$$ax \ mod \ m = \begin{cases} u, & si \ u \geq 0, \\ u + m, & si \ u < 0. \end{cases}$$

Demostració. Per demostrar-ho, haurem de demostrar els dos punts.

1.

$$a\,(x \bmod q) \leq a\,(q-1) = aq - a = m - r - a < m$$

$$r\left[\frac{x}{q}\right] \leq r\left[\frac{m-1}{q}\right] = ra < qa = m - r < m$$

Per tant, hem vist que els dos valors estan compresos entre 0 i $m-1$.

2. Com que per una banda tenim

$$a\,(x - (x \bmod q)) = a\,(x - s) = abq = aq\left[\frac{x}{q}\right]$$

I per l'altra

$$\left[\frac{x}{q}\right](m - (m \bmod q)) = b\,(m - r) = abq = aq\left[\frac{x}{q}\right]$$

Veiem que aleshores

$$a\,(x - (x \bmod q)) = \left[\frac{x}{q}\right](m - (m \bmod q))$$

on $m = aq + r$ i $x = bq + s$. Llavors,

$$ax - a\,(x \bmod q) = \left[\frac{x}{q}\right] m - \left[\frac{x}{q}\right] (m \bmod q)$$

on, sabem per hipòtesi que $r = m \bmod q$ i canviant termes de banda de la igualtat, obtenim

$$a\,(x \bmod q) - r\left[\frac{x}{q}\right] = ax - m\left[\frac{x}{q}\right]$$

Per tant,

$$u = a\,(x \bmod q) - r\left[\frac{x}{q}\right] = ax - bm$$

D'aquesta manera, sabem que ax menys un múltiple de m és el mateix que $ax \bmod m$. Gràcies al primer apartat, veiem que el valor de $u \in (-(m-1), m-1)$. A més, si u és negatiu li sumem m i, òbviament, seguirà sent $ax \bmod m$. Així doncs, ja hem demostrar la proposició, ja que

$$ax \bmod m = \begin{cases} u \text{ si } u \geq 0 \\ u + m \text{ si } u < 0 \end{cases}$$

□

Per tant, ja no tenim cap problema de desbordament de la capacitat dels **long int** ja que, com hem dit a la demostració, el valor u està comprès entre $-(m-1)$ i $m-1$, on $m = 2^{31} - 1$ i les variables **long int** poden emmagatzemar nombres enters entre -2^{31} i $2^{31} - 1$.

Implementació *estàndard mínim*
Aquesta proposició l'utilitzem en l'implementació en C de l'*estàndard mínim*, agafant la congruència multiplicativa de l'exemple (2.4.1).

```
1  /* ESTANDARD MINIM */
2
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  float estminim (int *seed){
7
8          if(*seed<=0){
9                  printf("seed <= 0\n");
10                 exit(1);
11         }
12
13         /* X_(i) = [a*X(i-1)] mod m */
```

```
14
15          int a = 16807; /* 7^5 */
16          int m = 2147483647; /* 2^31-1 */
17          int q,r;
18
19      /* apliquem la prop. anterior */
20
21      q = m/a;
22      r = m%a;
23      *seed = a*(*seed%q)-r*(*seed/q);
24
25          if(*seed < 0 ){
26      *seed = *seed + m;
27          }
28
29          return (*seed/((float)m));
30
31  }
```

2.4.2 RANDU

Ara, parlarem del mètode RANDU.

$$X_i = (65539X_{i-1}) \mod 2^{31}.$$

Aquest generador va aparèixer per primer cop a [13], i des de llavors, s'ha usat en diferents paquets *software* i llibres de text.
És un dels molts exemples dels generadors dolents que podem trobar. Aquesta gran quantitat de generadors dolents pensem que és a causa que creien que era fàcil generar nombres aleatoris i no comprovaven les condicions d'aleatorietat.
Els principals defectes de RANDU són: no té període màxim i que no compleix la condició d'aleatorietat A3.
El paquet comercial SAS en la seva cinquena edició incorpora una variant d'aquest generador (vegeu [12]) , que té els mateixos defectes que RANDU, definida per:

$$X_i = (16807X_{i-1}) \mod 2^{31},$$

en el qual usa una barreja addicional per millorar-ho, anomenat mètode de la barreja (vegeu la següent secció).

Per altra banda, pel lector interessat en conèixer més exemples dolents de generadors recomanem la lectura de [8], on podrà observar una llista de generadors amb els seus respectius problemes i també el lloc on s'han usat.

Implementació RANDU

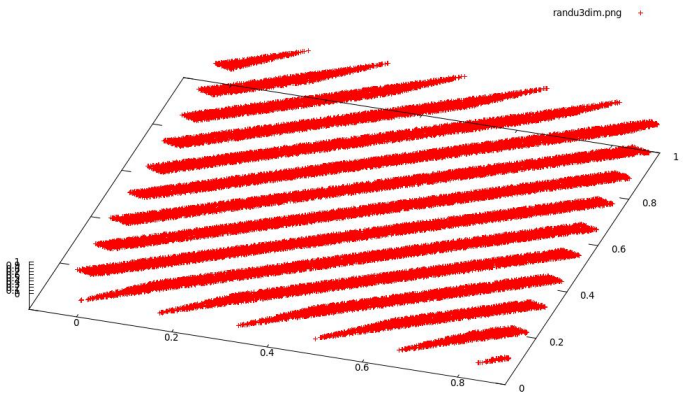
A continuació, implementem el mètode RANDU en llenguatge C:

```c
/* RANDU */

#include <stdio.h>
#include <stdlib.h>

float randu (int *seed){

     if(*seed<=0){
                printf("seed <= 0\n");
                exit(1);
        }

        /* X_(i) = [a*X(i-1)] mod m */

        int a = 65539;
        unsigned int m = 2147483648U; /* 2^31 */
        int q,r;

        /* apliquem la mateixa prop. que en la implementacio de l'est.
   minim */

        q = m/a;
        r = m%a;
        *seed = a*(*seed%q)-r*(*seed/q);

        if(*seed < 0 ){
          *seed = *seed + m;
        }
        return (*seed/((float)m));
}
```

Finalment, estudiarem exhaustivament un dels grans defectes de RANDU, que com ja hem comentat, no compleix la condició d'aleatorietat A3. Per fer-ho, hem implementat un programa en C en el qual generem 50.000 punts de 3 dimensions mitjançant RANDU i el resultat l'hem graficat amb el gnuplot:

```c
#include <stdio.h>
#include <stdio.h>

float randu (int *seed);

int main (void){

  int i,j, seed=1312, n=50000;
  float aux;
  FILE *fitxer;

  fitxer = fopen("randu3dim.txt", "w");

```

```
14    for(i=0;i<n;i++){
15      for(j=0;j<3;j++){
16        aux = randu(&seed);
17        fprintf(fitxer,"%f\t", aux);
18      }
19      fprintf(fitxer,"\n");
20    }
21
22    fclose(fitxer);
23    return 0;
24  }
```



Com veiem en el gràfic tots els punts estan en 15 plans, és a dir, existeix una correlació entre ells. Això és conseqüència de que si analitzem la congruència multiplicativa generadora, veiem que:

$x_{i+2} = (2^{16}+3)x_{i+1} \to x_{i+2} = (2^{16}+3)^2 x_i \to x_{i+2} = (2^{32}+9+6\cdot 2^{16})x_i \to$
$x_{i+2} = [6(2^{16}+3)-9]x_i \to x_{i+2} = 6x_{i+1} - 9x_i$

això succeix agafant cada terme mod 2^{31} i, conseqüentment, $2^{32} \bmod 2^{31} = 0$.

Per tant, hem comprovat que hi ha una correlació de RANDU en $\mathbb{R}^3$ i en conseqüència no compleix la propietat A3.

2.4.3 Mètode de la barreja

Primer de tot, explicarem la seva principal funció i com s'implementa aquest mètode teòricament; després, el programarem en C i discutirem la seva utilitat.

La principal funció d'aquest mètode és barrejar la successió aleatòria, és a dir, consisteix en reduir la correlació existent entre els diferents termes de la successió. Per

tant, qualsevol generador si incorpora aquest mètode compleix molt millor les condicions $A1$, $A2$, etc.

Implementació del mètode de la barreja

X_n és la successió d'enters que volem barrejar. Com que aquests nombres s'obtenen mitjançant una congruència lineal o multiplicativa amb mòdul m, $X_n \in (0, m-1)$. Definim T com un vector amb k components, és a dir, $T := (T_0, T_1, \ldots, T_{k-1})$ i P com variable entera tal que $T_0, T_1, \ldots, T_{k-1}$ i P són inicialitzats com $X_0, X_1, \ldots, X_{k-1}$ i X_k respectivament.
Llavors, l'algoritme és:

1. Sigui $j = \left[k\left(\frac{P}{m}\right)\right]$.
2. Sigui $P = T_j$.
3. Omplim T_j amb el següent terme de la successió X_n.
4. El resultat és P.

Comentem una mica aquest algorisme. En el pas 1 seleccionem el valor j mitjançant els primers dígits del valor P. En el pas 2 actualitzem el valor de P amb T_j, aquest valor serà el resultat de sortida al pas 4. Finalment, en el pas 3, actualitzem T_j amb el següent terme de la successió X_n.
Ara, veurem una modificació de la funció RANDU que incorpora el mètode de la barreja. Aquesta nova funció, l'hem anomenat barreja i és la següent:

```
1
2  /* METODE DE LA BARREJA */
3
4  #include <stdio.h>
5  #include <stdlib.h>
6
7  float barreja (int *seed){
8
9          int a = 65539;
10         unsigned int m = 2147483648U; /* 2^31 */
11         int q,r,i,j;
12         int k=32;
13         static int t[32];
14         static int p, inici=0;
15
16         if(inici==0){
17                 if(*seed>=0){
18                         printf("seed >= 0\n");
19                         exit(1);
20                 }
21             inici=1;
```

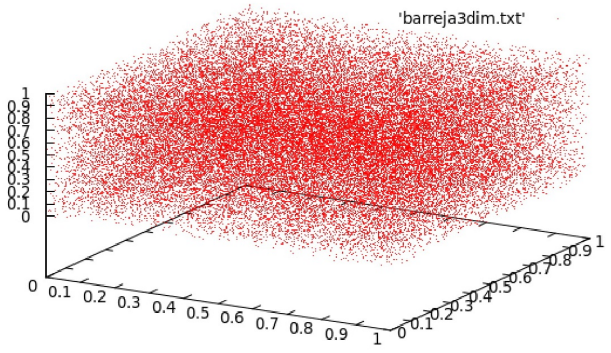
```
22          }
23
24      /* Inicialitzo q i r aplicant Randu */
25      q = m/a;
26          r = m%a;
27
28          if(*seed<0){
29                  *seed = -(*seed);
30                  for(i=0;i<k;i++){
31                          *seed = a*(*seed%q)-r*(*seed/q);
32                          if(*seed<0){
33                                  *seed = *seed + m;
34                          }
35                          t[i]=*seed;
36                  }
37                  /* Ara p = t[k] */
38                  *seed = a*(*seed%q)-r*(*seed/q);
39                  if(*seed<0){
40                          *seed = *seed + m;
41                  }
42                  p=*seed;
43          }
44
45      /* pas 1 i 2 */
46          j = (k * (p/((float)m)));
47          p = t[j];
48      *seed = a*(*seed%q)-r*(*seed/q);
49
50          if(*seed<0){
51                  *seed = *seed + m;
52          }
53
54          /* pas 3 */
55          t[j] = *seed;
56          return (p/((float)m));
57  }
```

Com hem vist, el mètode de la barreja serveix per barrejar la successió aleatòria tot i que el generador amb aquest mètode és lleugerament més lent i utilitza una mica més de memòria. Tenint-ho tot en compte, la meva conclusió és que és recomanable utilitzar-lo ja que els inconvenients són ínfims comparats amb la utilitat que aporta.

Per tant, durant la resta del treball aplicarem aquest mètode al generador RANDU per intentar arreglar el problema de correlació que hem vist anteriorment. Tot i que només ho aplicarem a RANDU, és obvi que el mètode de la barreja millora qualsevol generador i també ho podríem aplicar a *estàndard mínim*.

Finalment, aplicarem aquest mètode a la generació mitjançant RANDU en $\mathbb{R}^3$ per veure si en aquest cas compleix la condició d'aleatorietat A3. Per fer-ho, hem implementat un programa en C en el qual generem 50.000 punts de 3 dimensions mitjançant barreja i el resultat l'hem graficat amb el gnuplot.

Com que la implementació en C és gairebé idèntica que RANDU en $\mathbb{R}^3$, fiquem el programa en l'annex [1] i en el treball grafiquem el resultat:



Així doncs, acabem de veure que el mètode de la barreja implementat a RANDU corregeix el problema que teníem amb la condició d'aleatorietat A3 ja que els punts formen un núvol de punts i no "cauen" tots en els mateixos plans.

2.4.4 Crida dels generadors

En aquest apartat, farem un programa en C on cridarem els mètodes de generació de nombres aleatoris que acabem d'estudiar.

Primer de tot, li donarem un valor qualsevol a la llavor, nosaltres utilitzarem $X_0 = 1312$ en l'*estàndard mínim* i RANDU ja que necessiten un valor inicial positiu i, en canvi, en el mètode de la barreja utilitzarem $X_0 = -1312$ ja que necessitem una llavor negativa inicialment.

A continuació, cridarem les funcions programades en les seccions anteriors n vegades per poder crear els n nombres de les successions i, en cada pas, passarem la llavor actualitzada per adreça i la funció ens retorna el nombre aleatori.

És a dir, nosaltres només ens hem de preocupar en escollir una llavor inicial qualsevol (totes les llavors ens produiran uns resultats similars) i, llavors, el propi programa ja va actualitzant la llavor en cada pas.

```c

#include <stdio.h>
#include <stdlib.h>

float estminim (int *seed);
float randu (int *seed);
float barreja (int *seed);

int main (void){

int i, n=1000000, seed=1312;
float aux1;

  for(i=0;i<n;i++){
    aux1 = estminim(&seed);
    printf("%f\n", aux1);
  }

  seed = 1312;
  for(i=0;i<n;i++){
    aux1 = randu(&seed);
    printf("%f\n", aux1);
  }

  seed = -1312;
  for(i=0;i<n;i++){
    aux1 = barreja(&seed);
    printf("%f\n", aux1);
  }
  return 0;
}
```

2.5 Altres mètodes

Els mètodes que hem vist fins ara són els més bàsics dins de la generació de nombres aleatoris. Normalment, els mètodes més sofisticats es generen mitjançant congruèncial lineal. Una eina molt utilitzada és combinar diferents generadors de congruència lineal per generar diferents parts d'un nombre.

Per altra banda, tot i que nosaltres hem utilitzat la mateixa successió per generar els nombres com per mesclar-los en el mètode de la barreja, es pot utilitzar una successió per crear els nombres i una altra per barrejar-los. En aquest nou cas, si es trien correctament les dues successions aconseguim un període molt més gran, però si no ho fem així, ans al contrari, pot resultar la sèrie molt menys aleatòria que en el primer cas (utilitzat en l'apartat anterior).

Finalment, si esteu interessats en aquests mètodes més sofisticats, us recomanem la lectura de [4].

Capítol 3

Tests d'aleatorietat

En aquest tercer capítol, estudiarem els diferents tests d'aleatorietat més bàsics que hi ha per detectar si una successió és de variables aleatòries indepedents amb llei $U([0,1])$.

Mai no podrem afirmar de forma segura que una successió sigui aleatòria ja que nosaltres només analitzarem una propietat d'un nombre finit de termes i no tota la successió infinita, així doncs, el resultat del test s'ha d'entendre com una probabilitat de que sigui aleatòria o no. A més, que compleixi una certa propietat no se'n pot concloure la aleatorietat de la successió. Per tant, els distints tests ens serviran només per deduir successions no aleatòries.

En aquest capítol, la referència bibliogràfica principal són [3] i [11]. A més, el lector interessat es pot endinsar en el tema en [4] o [9].

3.1 Test χ^2

És el test més utilitzat i conegut que s'aplica a les successions discretes. Primerament, es suposa que la successió segueix una llei, és a dir, aquesta serà la nostra hipòtesi a contrastar. Seguidament, es compten cada vegada que surten tots els valors de la successió i es comparen amb els valors esperats. Finalment, mirarem si amb aquests resultats mitjançant un nivell de confiança escollit, acceptem o rebutgem la hipòtesi.

Aquest test el realitzem mitjançant aquest algorisme:
Sigui n el nombre total de termes de la successió.
Anomenem $1, 2, ..., s$ els possibles valors que poden prendre cada terme de la successió. Sigui n_j el nombre de vegades que apareix l'element j en la successió, $1 \leq j \leq s$. Suposem que cada un dels valors té la probabilitat p_j.

Definim

$$V = \sum_{j=1}^{s} \frac{(n_j - np_j)^2}{np_j} = \frac{1}{n} \sum_{j=1}^{s} \left(\frac{n_j^2}{p_j} \right) - n$$

De les definicions anteriors, la igualtat és evident ja que $\sum_{j=1}^{s} n_j = n$ i $\sum_{j=1}^{s} p_j = 1$. Triarem un nivell de confiança, normalment és el 95%, i ho contrastarem mitjançant la taula de la χ^2, per tal d'acceptar o rebutjar la hipòtesi (aquesta taula la ficarem a l'annex).

Implementació χ^2
Nosaltres voldrem aplicar aquest test a una successió de nombres reals a l'interval $[0, 1]$, per comprovar la seva uniformitat, per tant, ens servirà per detectar generadors dolents. Per fer-ho, trencarem l'interval en intervals disjunts $[0, 1] = I_1 \cup I_2 \cup ... \cup I_{10}$, comptarem quantes vegades cau un terme de la successió dins de cada interval i aplicarem l'algorisme definit anteriorment.

Ara, farem el programa en C d'aquest test utilitzant els tres generadors estudiats en el capítol anterior.

```
1
2  /* TEST DE LA CHI QUADRADA */
3
4  #include <stdio.h>
5  #include <stdlib.h>
6
7  float estminim (int *seed);
8  float randu (int *seed);
9  float barreja (int *seed);
10
11 int main (void){
12
13         int i,posicio, compt;
14         int j = 10; /* Hem partit el interval [0,1] en 10 parts */
15         int num=1000000; /* Volem 10^6 nombres aleatoris */
16         int n[j]; /* Vector que ens marca la quantitat de numeros que
       cauen dins de cada subinterval */
17         double p=1./j; /* probabilitat que tenen els diferents nombres
       de caure en cada subinterval */
18         double v=0; /* valor de la chi quadrada */
19         int seed; /* Com que apliquem el metode de la barreja la seed
       ha de ser negativa */
20
21     /* per fer el loop de les diferents llavors, faig un comptador i
       repeteixo aixo 4 vegades */
22         for(compt=0;compt<4;compt++){
23             printf("Dona'm una llavor\n");
24             scanf("%d", &seed);
25                 printf("seed: %d\t", seed);
26                 /* Inicialitzem el vector v a 0 */
```

```
27                  for(i=0;i<j;i++){
28                          n[i]=0;
29                  }
30
31                  for(i=0;i<num;i++){ /* cada numero cau dintre un
    subinterval dels 10*/
32                          posicio = barreja(&seed)*j;
33                          n[posicio] = n[posicio] + 1;
34                  }
35
36                  /* Calculem el valor de la chi quadrada */
37                  for(i=0;i<j;i++){
38                          v = v + n[i]*(n[i]/p);
39                  }
40                  v = v/num;
41          v = v - num;
42                  printf("chi quadrada = %f\n", v);
43          }
44      return 0;
45  }
```

Recordem que en el mètode de la barreja les llavors han de ser negatives i, per tant, hem utilitzat la mateixa llavor que en els altres dos mètodes però amb signe negatiu. També que per comprovar la validesa dels diferents mètodes, ho hem de fer amb diferentes llavors.
Aquests són els resultats obtinguts amb les diferents llavors:

Seeds	*Estàndard mínim*	RANDU	Barreja
1312	11.552440	8.281640	8.273640
1994	13.402752	6.436328	6.409188
2018	7.173533	8.990826	8.982066
36	9.624567	10.413269	10.366969

Com que treballem amb 95% de nivell de confiança i $\nu = j - 1 = 9$, mirem la taula de la χ^2 (annex [2]) i veiem que $V \leq 16.92$ per acceptar la hipòtesi. Així doncs, la generació mitjançant els tres generadors passa el test χ^2.

Finalment, també podríem aplicar el test de la χ^2 en $\mathbb{R}^3$ per comprovar que el generador RANDU no compleix la condició A3 explicada en el capítol anterior, però no volem aprofundir en aquest tema ja que només seria variar quatre detalls d'aquesta implementació en C.

3.2 Test de Kolmogorov - Smirnov

Aquest test s'aplica a successions que segueixen una llei contínua. La finalitat del qual és verificar si la distribució de la successió s'ajusta amb la distribució esperada.

Pel que fa el cas de lleis $U([0,1])$, aquest test és semblant al test χ^2 ja que els dos

estudien la "uniformitat" de la successió en $[0,1]$. Tot i que amb aquest mètode, no ens cal partir $[0,1]$ en diferents subintervals ja que contrastarem la funció de distribució.

Ara, veurem com s'aplica aquest test en el cas d'una llei uniforme $U([0,1])$:
Siguin $x_1, x_2, \ldots, x_n$ els nombres generats mitjançant algun mètode comentat en el capítol anterior. Primerament, ordenarem aquests nombres en ordre creixent i els renombrarem quan ja tinguem la successió ordenada, $x_1 \leq x_2 \leq \ldots \leq x_n$. Llavors, calculem:

$$K_n^+ = \sqrt{n} \max_{1\leq i\leq n} \left(\frac{i}{n} - x_i \right), \; K_n^- = \sqrt{n} \max_{1\leq i\leq n} \left(x_i - \frac{i-1}{n} \right)$$

Finalment triarem un nivell de confiança, normalment s'escull 95%, i mitjançant els resultats obtinguts i la taula de distribució del test (ho ficarem en un annex) contrastarem la hipòtesi.

Implementació test Kolmogorov - Smirnov
Tot seguit, implementem aquest test als tres mètodes de generar nombres aleatoris estudiats al tema anterior on utilitzem la funció *quicksort* amb la finalitat d'ordenar els nombres generats.

```
1  /* TEST KS */
2
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <math.h>
6
7  float estminim (int *seed);
8  float randu (int *seed);
9  float barreja (int *seed);
10 void quicksort(double *vector, int left, int right);
11
12 int main (void){
13
14         int i;
15         int seed, n=1000000, compt=0;
16         double kpos, kneg, maxpos, maxneg, v[n];
17
18
19     /* per fer el loop de les diferents llavors, faig un comptador i
       repeteixo aix  4 vegades */
20         for(compt=0;compt<4;compt++){
21           printf("Dona'm una llavor\n");
22           scanf("%d", &seed);
23             printf("seed: %d\t", seed);
24
25       for(i=0;i<n;i++){
```

```
26                      v[i] = estminim(&seed);
27              }
28
29              /* Utilitzem quicksort per ordenar els nombres */
30              quicksort(v,0,n-1);
31
32              /* Calculem K^+ i K^- */
33              maxpos=0;
34              maxneg=0;
35              for(i=0;i<n;i++){
36                      kpos = (double)(i+1)/n - v[i];
37                      if(kpos>maxpos){
38                      maxpos = kpos;
39                      }
40                      kneg = v[i] - (double)i/n;
41                      if(kneg>maxneg){
42                        maxneg = kneg;
43                      }
44              }
45
46        kpos = sqrt(n)*maxpos;
47              printf("K^+ = %f\t", kpos);
48              kneg = sqrt(n)*maxneg;
49              printf("K^- = %f\n", kneg);
50            }
51        return 0;
52  }
53
54  void quicksort(double *v, int left, int right){
55
56    int i, pivot;
57    double j, aux;
58
59      if(left<right){
60          pivot = left;
61              j = v[pivot];
62              for(i=left+1;i<=right;i++){
63                if(v[i] < j){
64                      pivot = pivot + 1;
65                        aux = v[i];
66                        v[i] = v[pivot];
67                        v[pivot] = aux;
68                    }
69              }
70              aux = v[left];
71              v[left] = v[pivot];
72              v[pivot] = aux;
73              quicksort(v,left,pivot-1);
74              quicksort(v, pivot+1, right);
75      }
76  }
```

Com en el cas del test de la χ^2, per poder comprovar la validesa dels diferents generadors, ho hem tingut de fer amb diferents llavors i, ara també la llavor del mètode de la barreja és el mateix valor que els altres dos però amb signe negatiu. Aquests han sigut els resultats:

Seeds	*Estàndard mínim*	RANDU	Barreja
1312	0.546772/1.017899	0.188614/0.842353	0.187614/0.844353
1994	0.359611/1.140517	0.158813/0.842467	0.157813/0.841467
2018	0.684823/0.275876	0.597287/0.405320	0.597287/0.400320
36	0.739334/0.273951	0.918343/0.565342	0.917343/0.566342

Veient la taula de distribució de Kolmorov - Smirnov (annex [3]) i utilitzant el nivell de confiança 95% i $n = 10^6$, els K_n^+ i K_n^- no han de superar 1.223707 ja que $y_p - \frac{1}{6\sqrt{n}} + 0(\frac{1}{n})$ on $y_p^2 = \frac{1}{2}ln(\frac{1}{1-p})$ ens ha donat aquest valor.

Així doncs, com que en cap dels 3 mètodes distints de generació que hem estudiat supera aquest valor, concluim que els 3 mètodes passen el test Kolmorov - Smirnov.

Finalment, si voleu més informació sobre aquest test, podeu cercar a [4] i [9]

Capítol 4

El mètode Monte Carlo

A vegades, volem fer un càlcul matemàtic el qual ens requereix molt temps en realitzar-lo, per exemple en matemàtica financera, càlculs científics o en l'estadística. Llavors, podem suposar que el resultat d'aquest càlcul és un valor esperat mitjançant un procés estocàstic o aleatori. Per tant, enlloc de calcular el resultat, podem aproximar-lo amb una simulació mitjançant nombres aleatoris. Un dels mètodes més coneguts i importants és el mètode de Monte Carlo.

Aquest mètode va ser creat l'any 1949 pels matemàtics nord-americans J.von Neumann i S.Ulam. El van anomenar Monte Carlo fent referència al casino Monte Carlo de la ciutat de Mònaco on, segons es diu, el tiet de Ulam hi anava a jugar.

Per poder explicar aquest mètode de forma senzilla, usarem el càlcul del valor d'una integral que és una aplicació comuna d'aquest mètode. En aquest capítol, les fonts [5],[2] i [1] seran les nostres principals referències bibliogràfiques. A més, utilitzarem la Llei dels grans nombres i el Teorema del Límit Central.

La llei dels grans nombres diu:

Teorema 4.0.1. *Sigui $x_1, x_2, ..., x_n$ una successió aleatòria amb la corresponent funció de densitat $\mu(x)$. Llavors, sabem que com tota funció de densitat, compleix que $\int_{-\infty}^{+\infty} \mu(x)dx = 1$*
Sigui $I = \int_{-\infty}^{+\infty} \mu(x)f(x)dx$ el valor esperat i $\overline{f_n} = \frac{1}{n}\sum_{i=1}^{n} f(x_i)$ la mitja de la mostra. Donat un $\varepsilon > 0$, tenim

$$\lim_{n\to\infty} Prob\left(I - \varepsilon \leq \overline{f_n} \leq I + \varepsilon\right) = 1$$

Per tant, aquesta llei ens diu que com més gran sigui n, és a dir, com més gran sigui la successió més s'apropararà la mitjana de la mostra amb el valor mig del valor esperat I. Per altra banda, si la successió té longitud infinita, tenim:

$$Prob\left(\lim_{n\to\infty} \overline{f_n} = I\right) = 1$$

Així doncs, nosaltres en el nostre exemple volem calcular la següent integral:

$$\mathrm{I} = \int_a^b f(x)dx$$

Primerament, veiem que el valor mig de $f(x)$ és $\dfrac{\mathrm{I}}{b-a}$ en aquest interval. A més, com que sabem que per la llei dels grans nombres, la mitjana de la mostra s'apropa al valor mig, sabem que:

$$\frac{1}{n}\sum_{i=1}^{n} f(x_i) \approx \frac{\mathrm{I}}{b-a} \Rightarrow \mathrm{I} \approx \frac{b-a}{n}\sum_{i=1}^{n} f(x_i)$$

Tot seguit, calcularem l'error d'aquesta estimació i per fer-ho aplicarem el Teorema del Límit Central. Aquest teorema ens diu que si tenim una successió aleatòria, la mida de la qual és més gran que 30 ($n > 30$), la mitja sempre seguirà una llei normal, sense importar la llei estadística de les variables de la successió. Així doncs,

$$\mathrm{Prob}\left(\mathrm{I} - \frac{\lambda\sigma}{\sqrt{n}} \leq \overline{f_n} \leq \mathrm{I} + \frac{\lambda\sigma}{\sqrt{n}}\right) = \frac{1}{\sqrt{2\pi}}\int_{-\lambda}^{\lambda} e^{\frac{-x^2}{2}}dx$$

on, com ja hem dit I és l'esperança, λ és una constant que depen del nivell de confiança que hi apliquem i $\sigma^2 = \int_{-\infty}^{+\infty}\left(f(x) - \mathrm{I}\right)^2 \mu(x)dx$ és la variància (com sabem per definició).
Llavors, obtenim aquesta taula de probabilitat de l'error en l'estimació depenen del nivell de confiança (aquestes són les probabilitats més utilitzades):

λ	Prob
0.6745	0.50
1.645	0.90
1.96	0.95
2.567	0.99

No obstant, per poder fer una estimació de l'error, hem vist que necessitem el valor σ però com que és desconegut també l'haurem d'estimar. Per fer-ho, utilitzarem la variància mostral per estimar σ^2 :

$$V = \frac{1}{n}\sum_{i=1}^{n} f^2(x_i) - \left(\frac{1}{n}\sum_{i=1}^{n} f(x_i)\right)^2$$

Potser seria millor utilitzar $\dfrac{n}{n-1}V$ per estimar la variància. Això és conseqüència que si tenim una mostra de mida N, podem agafar les $\binom{N}{n}$ mostres possibles de mida

n i llavors, si $\overline{V}$ és la mitjana de les variàncies de les submostres, està demostrat que:

$$\overline{V} = \frac{n-1}{n} V \left(1 + \frac{1}{N-1}\right)$$

A més, si la mida de la mostra és molt gran, tenim que:

$$\overline{V} \approx \frac{n-1}{n} V \Rightarrow V \approx \frac{n}{n-1} \overline{V}$$

Finalment, com que en el mètode de Monte Carlo n és tan gran que $\frac{n}{n-1} \approx 1$, llavors la correcció és gairebé negligible, és a dir:

$$V \approx \overline{V}$$

Ara, que ja hem estimat la desviació estàndard, analitzem l'estimació de l'error:

$$E = \mid \overline{f_n} - \mathrm{I} \mid \leq \frac{\lambda \sigma}{\sqrt{n}}$$

Per tant, sabem que:
Si tenim un nivell de confiança del: 90% , $E \leq \frac{1.645\sigma}{\sqrt{n}}$
Si tenim un nivell de confiança del: 95% , $E \leq \frac{1.960\sigma}{\sqrt{n}}$

Veiem doncs que per una λ constant, és a dir, fixat el nivell de confiança, tenim dues opcions per reduir l'error. Com que σ és directament proporcional, hem de reduir aquest valor i conseqüentment la variància per poder reduir l'error. Aquesta opció l'estudiarem en la següent secció.
Per altra banda, com que n és inversament proporcional per poder reduir l'error hem d'augmentar la mida de la mostra. A més, veiem que si augmentem n en un factor de 10 només disminuïm l'error en un factor de 1. Aquesta segona opció és coneguda com la llei $n^{\frac{-1}{2}}$.

4.1 Exemples aplicació Monte Carlo

Finalment farem dos exemples de càlcul d'integrals, el primer una integral simple i el segon una integral múltiple.

4.1.1 Integral simple

Comprovarem el valor de la integral:

$$I = \int_0^1 x^2 dx = \frac{1}{3}$$

on, com ja hem demostrat, I $\approx \frac{b-a}{n} \sum_{i=1}^{n} f(x_i)$ i $\sigma^2 \approx \frac{1}{n} \sum_{i=1}^{n} f^2(x_i) - \mathrm{I}^2$. Per tant, implementem el programa en C:

Implementació en C

```
/* EL METODE DE MONTECARLO */

#include <stdio.h>
#include <stdlib.h>

float barreja (int *seed);
float estminim (int *seed);
float randu (int *seed);

int main (void){

  int a=0,b=1, n=1000000, seed=-1312;
  double sigma, sumsigma=0, sum=0;
  double x,f,integral;
  int i;

  for(i=0;i<n;i++){
    /* Cridem la funcio que genera la successio */
    x=barreja(&seed);
    /* Calculem la funcio f(x) = x^2 i els sumatoris */
    f=x*x;
    sum = sum + f;
    sumsigma = sumsigma + f*f;
  }

  integral = ((double)(b-a)/n)*sum;
  sigma = ((double)(b-a)/n)*sumsigma - integral*integral;
  printf("El valor de la integral es: %f\n", integral);
  printf("El valor de la variancia es: %f\n", sigma);

  return 0;

}
```

Aquest programa l'hem implementat mitjançant els diferents mètodes de generació de successions aleatòries estudiats en el segon capítol: *estàndard mínim*, el mètode RANDU i el mètode RANDU millorat mitjançant el mètode de la barreja. Aquests han sigut els resultats:

•	*Estàndard mínim*	RANDU	Barreja
Seed	1312	1312	-1312
I	0.333712	0.333662	0.333663
σ^2	0.089013	0.089009	0.089009

4.1.2 Integral múltiple

Tot seguit, comprovarem el valor d'aquesta integral:

$$I = 2^{-7} \int_0^1 \cdots \int_0^1 (x_1 + \cdots + x_8)^2 \, dx_1 \cdots dx_8 = \frac{25}{192}$$

on, aplicarem les mateixes estimacions que en la integral anterior. Per tant, implementarem el programa en C:

Implementació en C

```c
/* EL METODE DE MONTECARLO AMB UNA INTEGRAL MULTIPLE */
#include <stdio.h>
#include <stdlib.h>

float barreja (int *seed);
float estminim (int *seed);
float randu (int *seed);

int main (void){

  int a=0,b=1, n=1000000, seed=1312;
  double sigma, sumsigma=0, sum=0;
  double x,f,integral;
  int i, j;

  for(i=0;i<n;i=i+8){
    x=0;
    for(j=0;j<8;j++){
      x = x + estminim(&seed);
    }
    f=x*x;
    sum = sum + f;
    sumsigma = sumsigma + f*f;
  }
  /* Divideixo per 128 ja que 2^7 = 128 */
  n = n/8;
  integral = (((double)(b-a)/n)*sum)/128;
  sigma = (((double)(b-a)/n)*sumsigma)/(128*128) - integral*integral;
  printf("El valor de la integral es: %f\n", integral);
  printf("El valor de la variancia es: %f\n", sigma);

  return 0;
}
```

Aquesta integral també l'hem calculat mitjançant els 3 mètodes de generació estudiants anteriorment. Aquests han sigut els resultats:

•	*Estàndard mínim*	RANDU	Barreja
Seed	1312	1312	-1312
I	0.130335	0.130364	0.130370
σ^2	0.002657	0.002673	0.002670

4.2 Reduir variància

Per poder reduir la variància, estudiarem dos mètodes distints:

4.2.1 Variable de control

Per poder estudiar aquest mètode de manera senzilla, ho farem mitjançant el següent exemple:
Suposem que volem calcular $I = \int_0^1 f(x)dx$ i volem trobar $g(x)$ tal que

$$\mid f(x) - g(x) \mid \leq \epsilon \ , \ 0 \leq x \leq 1$$

tal que

$$\int_0^1 g(x)dx = J$$

on J és un valor conegut i, $g(x)$ és anomenat variable de control de $f(x)$. Sigui

$$I_1 = \int_0^1 (f(x) - g(x))dx$$

per Monte Carlo. La variància en aquest problema equival a

$$\sigma_1^2 = \int_0^1 (f-g)^2 dx - \left(\int_0^1 (f-g)dx \right)^2$$

i per tant, estem prenent que $\sigma_1 \leq \epsilon$.
Llavors, com a estimador de I agafem:

$$\frac{1}{n} \sum_{i=1}^{n} (f(x_i) - g(x_i)) + J$$

Per tant, podem observar que la variància ha reduït ja que hem doblat el nombre d'avalucions funcionals.
Si el problema es fa de dues maneres diferents, tenint variàncies σ_1^2, σ_2^2 i el número d'avaluacions funcionals N_1, N_2 respectivament, llavors sabem que l'eficàcia relativa de l'utilització d'aquest mètode per reduir la variància és:

$$\frac{\frac{\lambda\sigma_1}{\sqrt{n}\sqrt{\frac{N_2}{N_1}}}}{\frac{\lambda\sigma_2}{\sqrt{n}}} = \frac{\sqrt{N_1}\sigma_1}{\sqrt{N_2}\sigma_2}$$

4.2.2 Mostreig d'importància

En aquest segon mètode, agafem també $I = \int_0^1 f(x)dx$ com a exemple i escrivim:

$$I = \int_0^1 \frac{f(x)}{p(x)} p(x)dx$$

on $p(x) > 0$ i $\int_0^1 p(x)dx = 1$. Ara agafem com a estimador de I:

$$\overline{f_n} = \frac{1}{n}\sum_{i=1}^{n} \frac{f(x_i)}{p(x_i)}$$

on x_i és una variable aleatòria que hem extret de la funció de distribució de $p(x)$ en $x \in [0, 1]$. Ara, podem calcular la variància com:

$$\sigma^2 = \int_0^1 \frac{f^2(x)}{p^2(x)} p(x)dx - \left(\int_0^1 \frac{f(x)}{p(x)} p(x)dx\right)^2$$

Llavors, podem suposar que $f(x) > 0$ ja que si fós negativa, podríem sumar una constant i ja estaríem en el mateix cas. Doncs, seleccionem:

$$p(x) \approx \frac{f(x)}{\int_0^1 f(x)dx}$$

Per tant, finalment arribem a que:

$$\sigma^2 \approx 0$$

Així doncs, acabem de veure que utilitzant una funció $p(x)$ de la qual coneixem el valor de la integral i que s'aproxima a la funció $f(x)$, reduirem la variància. Òbviament, el guany de reduir la variància s'ha de comparar amb el temps emprat en trobar $p(x)$ i calcular tot el mètode explicat.

A més, una altra conseqüència seria que, idealment, la mostra seria proporcional al valor de la funció amb constant de proporcionalitat igual al valor de la integral. I llavors, la variància seria 0 i en conseqüència hauríem de saber el resultat del problema prèviament.

No obstant, si aproximem $cf(x)$ per una funció $p(x)$ constant i definida a trossos, la mostra respecte la funció $p(x)$ ja no ens causarà cap problema. Per altra banda en d dimensions, la situació és més complicada, podríem intentar escollir $p_i(x_i)$ tal que

$$\prod_{i=1}^{d} p_i(x_i) \approx \frac{f(x)}{\int_{C_d} f(x)dV}$$

on $p_i(x_i)$ són d-funcions unidimensionals constants definides a trossos i C_d és el hipercub unitat $[0,1]^d$. Aquestes funcions $p_i(x_i)$ són escollides mitjançant un esquema iteratiu on es té en compte la variabilitat de $f(x)$ en cada dimensió amb l'aproximació de la integral i, després aquestes funcions $p_i(x_i)$, les ajustarem fins que convergeixin. Llavors, un mostreig respecte la distribució final amb funció de densitat ens aproxima al resultat amb una petita variació. Aquesta idea va ser implementada per G.P.Lepage al programa VEGAS.

Sasaki va proposar les funcions d'aquesta forma

$$h_1(x_1, x_2), h_2(x_2, x_3), \cdots, h_{d-1}(x_{d-1}, x_d)$$

pels dos mètodes estudiats per reduir la variància, és a dir, pel mètode variable de control i pel mètode mostreig d'importància. No obstant, aquestes funcions només són útils quan la variació de l'integrant no és gaire gran.

Hi han altres mètodes per reduir la variància com el mostreig estratificat i l'ús de les variables antitètiques. En el mostreig estratificat, dividirem el hipercub unitat $[0,1]^d$ amb hiper-rectangles i escollirem un nombre k fixat de mostrejos en cada hiper-rectangle. Si voleu aprofundir més en aquests dos mètodes, podeu cercar més informació en [2].

4.2.3 Exemple variable de control

Suposem que volem calcular $I = \int_0^1 e^x dx$ i volem trobar $g(x)$ tal que $\mid f(x) - g(x) \mid \leq \epsilon$, $0 \leq x \leq 1$ tal que $\int_0^1 g(x)dx = J$ on J sigui un valor conegut.

Per tant, la nostra variable de control $g(x) = 1 + x$ és la seva sèrie de Taylor.

Llavors, $I = \int_0^1 e^x dx = e^1 - 1 = 1.7182818\ldots$ i $J = \int_0^1 1 + x = \frac{3}{2}$.
Ara, agafem com a estimador de $I \approx \frac{1}{n}\sum_{i=1}^n (f(x_i) - g(x_i)) + J$ i d'estimador de la variància $\sigma^2 \approx \frac{1}{n}\sum_{i=1}^n (f(x_i) - g(x_i))^2 - \left(\frac{1}{n}\sum_{i=1}^n (f(x_i) - g(x_i))\right)^2$.

A continuació, fem la implementació en C d'aquest exemple:

Implementació en C

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

float barreja (int *seed);
float estminim (int *seed);
float randu (int *seed);

int main (void){

  int a=0,b=1, n=1000000, seed=1312;
  double sigma, sumsigma=0, sum=0;
  double x,f,integralf, integralg = 1.5;
  int i;

  for(i=0;i<n;i++){
    x=estminim(&seed);
    f=1 + x;
    f = exp(x) - f;
    sum = sum + f;
    sumsigma = sumsigma + f*f;
  }
  integralf = ((double)(b-a)/n)*sum + integralg;
  sigma = ((double)(b-a)/n)*sumsigma - ((double)(b-a)/n)*sum * ((double
    )(b-a)/n)*sum;
  printf("El valor de la integral es: %f\n", integralf);
  printf("El valor de la variancia es: %f\n", sigma);

  return 0;

}
```

Els resultats han sigut $I \approx 1.718557$ i $\sigma^2 \approx 0.043712$ quan hem generat mitjançant *estàndard mínim*. Per tant, si ho comparem amb el càlcul de la integral mitjançant el mètode de Monte Carlo, sense aplicar el mètode de reduir la variància, veiem que

els resultats són $I \approx 1.718809$ i $\sigma^2 \approx 0.242396$. Aquests càlculs els fiquem a l'Annex [4] ja que la implementació en C és molt semblant a l'exemple de integral simple fet en la secció anterior només variant la funció.

Així doncs podem concluir que aplicant aquest mètode per reduir la variància, ens aproximem més al valor de la integral.

Capítol 5

Tècniques de simulació discreta

En aquest capítol, com els seu nom indica, veurem les tècniques de simulació de processos discrets. Per fer-ho, veurem un exemple senzill del problema que ens interessa que és la gestió de cues i a partir d'ell veurem que amb 4 retocs bàsics la mateixa implementació utilitzada és útil en molts altres casos.

Durant tot aquest capítol, la principal font bibliogràfica utitlitzada ha estat [3] tot i que, per altra banda, la informació més específica ja la detallarem en cada cas.

5.1 Una única cua

El nostre exemple senzill és simular l'evolució d'una única cua davant d'un caixer d'un supermercat. Per tant, haurem de ficar gent a la cua, treure gent de la cua, saber el temps mitjà d'espera, la longitud màxima, etc. Per simplicar la simulació, podem implementar-la com una successió d'esdeveniments.

5.1.1 Esdeveniments

Els esdeveniments són canvis en l'estat del sistema que estem simulant, és a dir, en el nostres cas són l'ARRIBADA d'un nou client a la cua i la SORTIDA del primer client de la cua. Així doncs, quan no succeeix cap esdeveniment, el programa no ha de fer res i per tant, el programa es converteix en un bucle on, en cada passsada es tracta de resoldre el nou esdeveniment o de dur a terme les accions necessàries i també es calcula quan passarà l'esdeveniment següent.

Veiem que en aquesta simulació tenim dos esdeveniments més: l'esdeveniment OBRIR que representa l'esdeveniment inicial quan obrim la caixa i l'esdeveniment TANCAR que, òbviament, és l'esdeveniment final quan tanquem la caixa. Per tant, tenim 4 esdeveniments finalment, els quals estan compostos per dues dades: el QUE passa i el QUAN passa. Conseqüentment, quan implementem el programa per facilitar-nos la feina els definirem com estructures.

Així, definirem un nou tipus de variable que anomenarem **esdev** el qual tindrà dos tipus de variable: una de tipus **float** que serà el temps i l'altra de tipus **int** que serà el tipus d'esdeveniment. Aquesta estructura en llenguatge C serà així:

```
typedef struct {

  float quan;
  int que;

}esdev;
```

5.1.2 Agenda

Ara, aquests esdeveniments per poder fer la simulació els necessitem programar de forma ordenada. Per fer-ho, utilitzarem un conjunt de funcions que s'anomenen **agenda**. La funció principal de l'agenda és simplificar la simulació i així reduir el programa a un bucle molt més senzill. A més, gairebé totes les simulacions es poden programar mitjançant una agenda, per tant, hem d'intentar programar una agenda poc específica per poder utilitzar-la en diferents casos. En conseqüència, per exemple, no fixem la mida de l'agenda.

L'agenda consta de diferents funcions:

(i) **inici agenda**: guarda mèmoria per poder crear l'agenda de qualsevol dimensió n.

(ii) **posa agenda**: introdueix els esdeveniments, afegint-los a la llista de l'agenda dels futurs esdeveniments.

(iii) **treure agenda**: treu els esdeveniments, borrant-los de l'agenda de forma ordenada segons el temps.

(iv) **buida agenda**: reinicia l'agenda per poder fer una altra simulació.

(v) **free agenda**: allibera la mèmoria que hem reservat mitjançant la funció **inici agenda**.

Tot seguit, veurem una implementació en C d'una agenda en general:

Implementació en C

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {

  float quan;
```

```
7   int que;
8 }esdev;
9
10 static esdev *agenda;
11 static int numesd, ara;
12
13 void inici_agenda(int n){
14   numesd = n;
15   agenda = (esdev *)malloc(numesd*sizeof(esdev));
16   if(agenda == NULL){
17     printf("Falta memoria per l'agenda\n");
18     exit(1);
19   }
20   ara = -1;
21 }
22
23 void posa_agenda (esdev e){
24
25   int i;
26   ara = ara + 1;
27   if(ara == numesd){
28     printf("Error: agenda plena\n");
29     exit(1);
30   }
31
32   /* aquest bucle ordena els esdev dins l'agenda */
33   for(i=ara; i>0; i--){
34     if(e.quan <= (agenda[i-1]).quan){
35       break;
36     }
37     agenda[i] = agenda[i-1];
38   }
39   agenda[i] = e;
40 }
41
42 int treure_agenda(esdev *e){
43
44   if(ara == -1){
45     /* agenda buida */
46     return 0;
47   }
48   /* retorna l'esdev que porta mes temps a l'agenda */
49   *e = agenda[ara];
50   ara = ara - 1;
51   return 1;
52 }
53
54 void buida_agenda(void){
55   /* reiniciem l'agenda amb 0 esdev */
56   ara = -1;
57 }
58
59 void free_agenda(void){
```

```
60   /* alliberem la memoria guardada */
61   free(agenda);
62 }
```

Primer de tot, podem observar que hem creat dos variables globals, és a dir, declarades fora de qualsevol funció. Aquestes variables són: un vector d'esdeveniments que és la nostra agenda i una variable entera anomenada ara, la qual és una variable de control del nombre d'esdeveniments que tenim a l'agenda.

Pel que fa les funcions, la gran majoria d'elles no cal fer-ne un anàlisis exhaustiu ja que són elementals i amb l'ajuda de les definicions anteriors al programa i el mateix programa, es comprenen a la perfecció. Així, només analitzarem més detalladament les funcions:

(i) **posa agenda**: primerament, vigila que l'agenda no estigui plena mitjançant la variable de control ara. Si fós el cas, parem el programa ja que no podem guardar el següent esdeveniment. En cas contrari, afegim el nou esdeveniment a l'agenda però de forma ordenada, és a dir, fem un bucle ordenant els esdeveniments de manera que el que trigui més temps a passar estigui en la component 0 i el més immediat estigui en la component més alta del vector.

(ii) **treure agenda**: aquesta funció bàsicament retorna l'esdeveniment que porta més temps a l'agenda mitjançant la variable de control ara la qual, com ja hem dit, ens indica la component del vector més antiga. A més de retornarnos l'esdeveniment, també ens retorna un 1. En cas contrari, és a dir, quan l'agenda és buida, ens retorna un 0.

Ara que ja hem explicat el funcionament de l'agenda, podem tornar al nostre exemple senzill d'una única cua. Per implementar-lo, a part de necessitar l'agenda estudiada, també necessitarem un conjunt de funcions que ens facilitaran la simulació.

5.1.3 Funcions prèvies

Aquest conjunt de funcions per gestionar una cua tenen una certa similitud amb les funcions que hem definit anteriorment per gestionar una agenda. Per exemple, enlloc de posar i treure esdeveniments de l'agenda, treurem i posarem persones a la cua; enlloc de crear una agenda i al final alliberar la memòria reservada per ella, farem exactament el mateix però per la cua. Així doncs, la implementació en C de a continuació, n'és un exemple d'aquest conjunt de funcions:

Implementació en C

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 typedef struct{
5   float temps;
```

```
6   int numprod;
7 }persona;
8
9 static persona *cua;
10 static int maxcua, inicua, fincua;
11
12 void crea_cua (int n){
13
14   maxcua = n;
15   cua = (persona *)malloc(maxcua * sizeof(persona));
16   if(cua == NULL){
17     printf("No tenim memoria suficient per la cua\n");
18     exit(1);
19   }
20
21   inicua = 0;
22   fincua = 0;
23 }
24
25 int posa_cua (persona p){
26
27   if((fincua - inicua) == maxcua){
28     return 0;
29   }
30
31   cua[fincua] = p;
32   fincua = fincua + 1;
33
34   if(fincua == maxcua){
35     fincua =0;
36   }
37   return 1;
38 }
39
40 int treu_cua (persona *p){
41
42   if((fincua - inicua) == 0){
43     return 0;
44   }
45
46   *p = cua[inicua];
47   inicua = inicua + 1;
48
49   if(inicua == maxcua){
50     inicua = 0;
51   }
52   return 1;
53 }
54
55 int llargada (void){
56
57   int longcua;
58   longcua = fincua - inicua;
```

```
59   return(longcua);
60 }
61
62 void free_cua(void){
63   free(cua);
64 }
```

Primer de tot, definim una variable estructura anomenada persona que representa la nova persona que es fica a la cua. Aquesta estructura té dues variables, la variable temps de tipus float i la variable numprod de tipus int.
La variable temps és l'hora en que arriba a la cua la persona, així podrem calcular quant temps ha passat a la cua o la mitjana del temps d'espera, per exemple. I la segona variable numprod indica el número de productes que porta la persona, així podrem saber el temps de pagament i altra informació revellant.

Per altra banda, en la nostra implementació d'aquestes funcions per gestionar la cua tenim un inconvenient, hem de saber prèviament la longitud màxima que pot tenir la cua, o si més no, hem de limitar-la per un nombre finit molt gran tot i que sapiguéssim que mai hi arribaríem. Això és conseqüència del nostre mètode de gestionar com posar i treure els clients de la cua.
Aquest mètode és circular, és a dir, quan arribem a l'última component del vector cua, el següent client el fiquem a la primera component del vector. A més, la variable fincua representa el nou client que fiquem a la cua i, en canvi, la variable inicua és l'últim client que marxa de la cua. Així doncs, aquestes dues variables es van desplaçant pel vector cua, llavors la cua estarà plena quan fincua atrapi a inicua i la cua estarà buida quan inicua atrapi a fincua.
Per tant, tot i l'inconvenient de que hem de saber la longitud màxima prèviament, aquest mètode és molt útil per la seva rapidesa i simplicitat. Ara, analitzarem el funcionament de les diferents funcions utilitzades:

- **crea cua**: Guardem la memòria escollida prèviament de dimensió maxcua pel vector cua. Després, inicialitzem els dos índexs inicua i fincua a 0 que ens serviran per controlar la llargada de la cua.

- **posa cua**: Primer, comprovem que la cua no estigui plena. Si estigués plena retornaríem un 0, en cas contrari, afegirem el nou client a la cua mitjançant l'índex fincua, augmentarem l'índex una posició i retornarem un 1. Finalment, si fincua arribés a l'última component del vector, li donaríem la volta al vector ja que utilitzem el mètode circular explicat anteriorment. Per tant, la funció principal d'aquesta funció és afegir un nou client a la cua quan és possible.

- **treure cua**: La seva funció és retornar el primer client de la cua i ho farà mitjançant la variable inicua. Si la cua no és buida, retorna el primer client, augmenta l'índex inicua una posició i també retorna un 1. En cas contrari, retorna un 0. També revisa que l'índex inicua no hagi arribat a l'última component del vector i si calgués li donaria la volta com hem explicat.

- **llargada**: La funció retorna la llarga de la cua, el seu funcionament és trivial.
- **free cua**: Elimina la memòria reservada pel vector cua.

5.1.4 El programa principal

Després d'haver fet tots els preliminars pertinents per poder realitzar el nostre exemple senzill del funcionament d'una cua.
Ara implementarem el programa principal en C suposant que el temps entre dos clients segueix una exponencial de mitjana 3 minuts i, en canvi, el temps que passa el client davant del caixer varia segons el nombre de productes que porta i aquests depenen de l'hora del dia en que van a comprar.

Implementació en C

```
1
2  #include <math.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  typedef struct{
7                  float quan;
8                  int que;
9  }esdev;
10
11 typedef struct{
12                 float temps;
13                 int numprod;
14 }persona;
15
16
17 #include "agenda.h"
18 #include "cua.h"
19 #include "estminim.h"
20
21 #define OBRIR 1
22 #define ARRIBADA 2
23 #define SORTIDA 3
24 #define TANCAR 4
25
26 int main (void){
27
28   FILE *f;
29   f = fopen("cua.txt", "w");
30
31   float expo(float m, int *seed);
32   int productes (float temps, int *seed);
33   /* ESTRUCTURES */
34   esdev e;
35   persona p;
36   /* tmax temps maxim que una persona ha estat fent cua */
```

```
37  float t, tmax, tmitja;
38  int seed;
39  int open, caixa, ntotal, nocua, j, i;
40  /* open=0 --> caixer tancat, open=1 --> caixer obert  */
41  /* caixa=0 --> client directe caixa, caixa=1 --> client a la cua */
42  inici_agenda(3);
43  crea_cua(100);
44  seed = 1312;
45
46  fprintf(f,"ntotal\t  tmax\t          tmitja\t      numnocua\n");
47  for(i=0;i<10;i++){
48
49  tmitja=0;
50  tmax = 0;
51  open=0;
52  caixa=0;
53  ntotal=0;
54  nocua=1;
55  e.que = OBRIR;
56  e.quan = 0.;
57  posa_agenda(e);
58  e.que = TANCAR;
59  e.quan = 720.;
60  posa_agenda(e);
61  while(treure_agenda(&e) != 0){
62    switch(e.que){
63      case OBRIR:
64        caixa=0;
65        open =1;
66        e.que= ARRIBADA;
67        e.quan=expo(3,&seed);
68        posa_agenda(e);
69        break;
70      case ARRIBADA:
71        /* revisem que la caixa estigui oberta */
72        if(open == 1){
73          t=e.quan;
74          /* client directe a la caixa */
75          if(caixa == 0){
76            nocua = nocua + 1;
77            caixa=1;
78            p.numprod = productes(p.temps,&seed);
79            e.quan = e.quan + (0.5 + p.numprod*0.05);
80            e.que = SORTIDA;
81            posa_agenda(e);
82          }else{ /* client a la cua: agafem l'hora d'arribada i el
    fiquem a la cua */
83            p.temps = t;
84            j = posa_cua(p);
85            if(j == 0){
86              printf("Error: necessitem el vector cua m s gran\n");
87              exit(1);
88            }
```

```
89            }
90            /* seguent client */
91            e.quan = t + expo(3,&seed);
92            e.que= ARRIBADA;
93            posa_agenda(e);
94          }
95          break;
96        case SORTIDA:
97          /* augmenta el num. total de clients atesos */
98          ntotal = ntotal + 1;
99          j= treu_cua(&p);
100         if(j != 0){ /* si j=1 hi ha cua, si j=0 cua buida */
101           t=e.quan - p.temps;
102           tmitja = tmitja + t;
103           if(t > tmax){
104             tmax = t;
105           }
106           p.numprod = productes(p.temps,&seed);
107           e.quan = e.quan + (0.5 + p.numprod*0.05);
108           e.que = SORTIDA;
109           posa_agenda(e);
110         }else{
111           caixa = 0;
112         }
113         break;
114       case TANCAR:
115         open=0;
116         break;
117       default: printf("error: esdeveniment desconegut\n");
118     }
119   }
120
121   fprintf(f,"%d\t", ntotal);
122   fprintf(f,"%f\t", tmax);
123   tmitja = tmitja/ntotal;
124   nocua = (float)nocua/ntotal*100;
125   fprintf(f,"%f\t", tmitja);
126   fprintf(f, "%d\n", nocua);
127
128   buida_agenda;
129
130   }
131
132   free_cua;
133   free_agenda;
134   fclose(f);
135
136   return 0;
137 }
138
139 float expo(float f, int *seed){
140   float aux;
141   aux = -f*log(estminim(seed));
```

```
142   return aux;
143 }
144
145
146 int productes (float temps, int *seed){
147
148   int nprod;
149
150   if(temps<240){
151     nprod = expo(20,seed) + 0.5;
152     return nprod;
153   }else if(temps >= 240 && temps <480){
154     nprod = expo(15,seed) +0.5;
155     return nprod;
156     }else{
157       nprod = expo(10,seed) +0.5;
158       return nprod;
159   }
160
161 }
```

A continuació, comentarem tot el funcionament del programa i per fer-ho, seguirem l'ordre de la implementació en C. Principalment, aquest programa utilitza el gestió d'agendes i les funcions de gestió de cues, implementades i comentades en les seccions anteriors, i també el generador *estàndard mínim* estudiat en el segon capítol.

Primer de tot, definim els 4 esdeveniments que usarem en la simulació: OBRIR, ARRIBADA, SORTIDA i TANCAR; els quals explicarem detalladament quan parlem del funcionament del bucle principal. Seguidament, definim les dues estructures explicades en seccions anteriors: esdeveniment i persona; també moltes variables int i float que la gran majoria tenen la seva utilitat exclusiva en poder obtenir resultats, menys dues que les utilitzarem com a variables de control.

La primera és la variable open que quan sigui 0 voldrà dir que el caixer està tancat, i quan sigui 1, el caixer estarà obert. L'altra és la variable caixa que quan caixa = 0, el client anirà directe a la caixa i si caixa = 1, el client anirà a la cua ja que la caixa estarà plena.

Després, inicialiatzem l'agenda, hi posem l'esdeveniment OBRIR i TANCAR (hem posat un temps de funcionament de 12 hores) i creem la cua de dimensió 100 (si fós massa petit el vector, ens avisaria el programa i llavors crearíem un vector més gran). Tot seguit, implementem el bucle principal que no pararà sempre i quan hi hagi algun esdeveniment a l'agenda. Aquest bucle va agafant esdeveniments i depenen d'ells, farà unes accios o unes altres. Veiem-ho:

- OBRIR: Posem open a 1, caixa a 0 i generem l'arribada del primer client mitjançant la llei estadística corresponent, el qual el posem a l'agenda.
- ARRIBADA: Primer de tot, revisem que la caixa estigui oberta mitjançant

la variable open, si no ho estigués no tenim en compte l'arribada del client. Llavors, depenen de la variable caixa distingim dos casos:

(i) caixa = 0 (la caixa està buida): Augmentem el comptador nocua, canviem la variable caixa (ja que ara estarà plena), calculem el temps que tarda el client en pagar i el fiquem a l'agenda com a esdeveniment SORTIDA.

(ii) caixa = 1 (la caixa està plena): Posem el nou client a la cua guardant l'hora en que ha arribat. També, revisem que el nou client càpiga a la cua, és a dir, revisem que la longitud del vector cua sigui suficientment gran.

Finalment, tant en un cas com en l'altre, preparem l'arribada del següent client posant un esdeveniment ARRIBADA a l'agenda.

- SORTIDA: Primerament, augmentem el número total de clients atesos, calculem el temps d'espera del client per poder comparar-lo amb els altres per saber el temps màxim d'espera i per calcular el temps mitjà d'espera. Finalment, calculem el temps que el client ocupa la caixa de pagament i posem l'esdeveniment SORTIDA a l'agenda. Tanmateix, si fós el cas que no hi hagués ningú a la cua, actualitzarem la variable caixa = 0.
- TANCAR: Per tancar el caixer utilitzarem open = 0, llavors ja no acceptarem l'arribada a la cua de nous clients, però això no vol dir que haguem acabat aquí ja que hem d'acabar d'atendre els clients de la cua.

A més, escrivim **default** per si tenim algun problema de programació i en l'agenda tenim un esdeveniment que no és un dels 4 explicats.

Per acabar, hem implementat la funció expo que justament és la funció de la llei exponencial definida en el primer capítol, i també una funció que hem anomenat productes que, utilitzant la funció expo, et retorna el nombre de productes dels clients depenen de l'hora del dia.

Com heu pogut veure, hem fet la simulació de 10 dies d'obertura de supermercat en els quals hem simulat 12 hores cada dia i aquests han sigut els resultats:

num.total	t.max	t.mitja	num.nocua
253	11.987411	1.187083	56
249	9.620773	0.756744	58
248	6.113190	0.710487	55
221	10.621674	0.990101	64
225	10.383255	1.000132	57
238	5.632294	0.506792	63
224	5.948990	0.661488	58
214	4.107391	0.427149	64
264	5.767761	0.592633	54
254	9.976257	0.693359	58

Així doncs, acabem d'implementar i explicar la simulació d'una única cua. Aquest cas tan senzill, no s'acostuma a simular ja que els mateixos venedors ja veuen a simple vista si el funcionament de la cua és correcte o no. És a dir, no tenen la necessitat d'implementar un programa per poder veure un problema de cues i poder solucionar-lo mitjançant algun alternativa, ja que tant la visió del problema com la gestió de l'alternativa són trivials.

Tot i això, aquest exemple ens ha servit per poder explicar aquesta tècnica de simulació discreta que s'implementa utilitzant l'agenda i els esdeveniments. Aquesta tècnica és molt pràctica ja que mitjançant aquest bucle principal i fent 4 retocs com per exemple les lleis estadístiques o introduint més cues, pots simular una gran varietat de casos diferents.

Per tant, acabem de fer una breu introducció a aquesta tècnica de simulació discreta la qual s'utilitza en molts casos més complexes com ara la simulació de trànsit, diferents tipus de cues de peatges o de supermercat, és a dir, en la gestió de cues.

Capítol 6

Conclusions

Després de tants coneixements teòrics durant la carrera, vaig escollir aquest tema perquè els volia aplicar en un treball en el qual es veiés reflexat la matemàtica que havia après durant aquest temps però sobretot per poder-ho utilitzar en aplicacions més pràctiques en la vida real. En aquest treball doncs, m'he endinsat en temes que em semblaven interessants i no havia "tocat" en el grau i n'he pogut extreure diferentes conclusions:

Primerament, com que les successions "aleatòries" avui dia es generen mitjançant ordenadors i aquests són màquines deterministes, és a dir, la seva funció és implementar ordres preestablertes, conseqüentment aquestes successions mai seran aleatòries i les anomenarem successions pseudoaleatòries.

Aquestes successions pseudoaleatòries les fem servir per generar lleis uniformes les quals són molt utilitzades i, a més a més, a partir d'elles hem estudiat com generar la llei exponencial i normal.
Per poder generar la llei $U([0,1])$ hem vist que es pot usar tant congruència lineal com congruència multiplicativa. Nosaltres hem acabat deduint que si escollim els valors a i m de forma adequada són més pràctics els generadors de congruència multiplicativa i per això, hem utilitzat aquests 3 mètodes: *estàndard mínim*, RANDU i el mètode de la barreja.

El mètode RANDU es va implementar durant molts anys al segle passat tot i que no és un bon generador perquè no compleix la condició d'aleatorietat A3, com hem pogut demostrar mitjançant la gràfica en 3 dimensions o mitjançant el test de la Test χ^2.
Per altra banda, el mètode de la barreja s'aplica a un mètode de generació i la seva principal funció és "barrejar" la successió pseudoaleatòria generada. A més, hem vist que quan l'apliquem a RANDU, corregeix el problema d'aleatorietat A3 que teníem. Per tant, la meva conclusió és que és recomanable utilitzar-lo ja que els inconvenients són ínfims comparats amb la utilitat que aporta.

Mai no podrem afirmar de forma segura que una successió sigui "aleatòria" ja que nosaltres mitjançant els tests d'aleatorietat només hem analitzat una propietat d'un nombre finit de termes i no tota la successió infinita, així doncs, els resultats del tests només ens serviran per deduir les successions no aleatòries.
Hem vist que els 3 generadors estudiats passen el test χ^2 i de Kolmogorov - Smirnov. Tot i que sabem que el generador RANDU en $\mathbb{R}^3$ no passaria el test.

Hem estudiat el mètode de Monte Carlo que serveix per calcular integrals mitjançant una aproximació amb nombres aleatoris. A part, hem vist que aquesta aproximació és més acurada si reduïm la variància i això ho podem fer mitjançant diferents mètodes com ara amb una variable de control o amb mostreig d'importància.

Finalment, hem estudiat una tècnica concreta de simulació discreta que consisteix en utilitzar una agenda amb esdeveniments i així et queda un bucle principal senzill, per fer-ho hem usat un exemple simple de gestió d'una única cua. Un avantatge d'aquesta tècnica és que s'utilitza una implementació estàndard i que a partir d'ella, mitjançant quatre retocs que ens interessin, aquesta implementació ens serveix per poder simular una gran varietat d'aplicacions.

Annexos

Annex 1

Implementem en C el mètode de la barreja al generador randu a $\mathbb{R}^3$.

```c
#include <stdio.h>
#include <stdio.h>

float barreja (int *seed);

int main (void){

  int i,j, seed=-1312, n=50000;
  float aux;
  FILE *fitxer;

  fitxer = fopen("barreja3dim.txt", "w");

  for(i=0;i<n;i++){
    for(j=0;j<3;j++){
      aux = barreja(&seed);
      fprintf(fitxer,"%f\t", aux);
    }
    fprintf(fitxer,"\n");
  }

  fclose(fitxer);
  return 0;
}
```

Annex 2

Percentage Points of the Chi-Square Distribution

Degrees of Freedom	Probability of a larger value of x^2								
	0.99	**0.95**	**0.90**	**0.75**	**0.50**	**0.25**	**0.10**	**0.05**	**0.01**
1	0.000	0.004	0.016	0.102	0.455	1.32	2.71	3.84	6.63
2	0.020	0.103	0.211	0.575	1.386	2.77	4.61	5.99	9.21
3	0.115	0.352	0.584	1.212	2.366	4.11	6.25	7.81	11.34
4	0.297	0.711	1.064	1.923	3.357	5.39	7.78	9.49	13.28
5	0.554	1.145	1.610	2.675	4.351	6.63	9.24	11.07	15.09
6	0.872	1.635	2.204	3.455	5.348	7.84	10.64	12.59	16.81
7	1.239	2.167	2.833	4.255	6.346	9.04	12.02	14.07	18.48
8	1.647	2.733	3.490	5.071	7.344	10.22	13.36	15.51	20.09
9	2.088	3.325	4.168	5.899	8.343	11.39	14.68	16.92	21.67
10	2.558	3.940	4.865	6.737	9.342	12.55	15.99	18.31	23.21
11	3.053	4.575	5.578	7.584	10.341	13.70	17.28	19.68	24.72
12	3.571	5.226	6.304	8.438	11.340	14.85	18.55	21.03	26.22
13	4.107	5.892	7.042	9.299	12.340	15.98	19.81	22.36	27.69
14	4.660	6.571	7.790	10.165	13.339	17.12	21.06	23.68	29.14
15	5.229	7.261	8.547	11.037	14.339	18.25	22.31	25.00	30.58
16	5.812	7.962	9.312	11.912	15.338	19.37	23.54	26.30	32.00
17	6.408	8.672	10.085	12.792	16.338	20.49	24.77	27.59	33.41
18	7.015	9.390	10.865	13.675	17.338	21.60	25.99	28.87	34.80
19	7.633	10.117	11.651	14.562	18.338	22.72	27.20	30.14	36.19
20	8.260	10.851	12.443	15.452	19.337	23.83	28.41	31.41	37.57
22	9.542	12.338	14.041	17.240	21.337	26.04	30.81	33.92	40.29
24	10.856	13.848	15.659	19.037	23.337	28.24	33.20	36.42	42.98
26	12.198	15.379	17.292	20.843	25.336	30.43	35.56	38.89	45.64
28	13.565	16.928	18.939	22.657	27.336	32.62	37.92	41.34	48.28
30	14.953	18.493	20.599	24.478	29.336	34.80	40.26	43.77	50.89
40	22.164	26.509	29.051	33.660	39.335	45.62	51.80	55.76	63.69
50	27.707	34.764	37.689	42.942	49.335	56.33	63.17	67.50	76.15
60	37.485	43.188	46.459	52.294	59.335	66.98	74.40	79.08	88.38

Taula de la χ^2

Annex 3

n \ α	0.001	0.01	0.02	0.05	0.1	0.15	0.2
1		0.99500	0.99000	0.97500	0.95000	0.92500	0.90000
2	0.97764	0.92930	0.90000	0.84189	0.77639	0.72614	0.68377
3	0.92063	0.82900	0.78456	0.70760	0.63604	0.59582	0.56481
4	0.85046	0.73421	0.68887	0.62394	0.56522	0.52476	0.49265
5	0.78137	0.66855	0.62718	0.56327	0.50945	0.47439	0.44697
6	0.72479	0.61660	0.57741	0.51926	0.46799	0.43526	0.41035
7	0.67930	0.57580	0.53844	0.48343	0.43607	0.40497	0.38145
8	0.64098	0.54180	0.50654	0.45427	0.40962	0.38062	0.35828
9	0.60846	0.51330	0.47960	0.43001	0.38746	0.36006	0.33907
10	0.58042	0.48895	0.45662	0.40925	0.36866	0.34250	0.32257
11	0.55588	0.46770	0.43670	0.39122	0.35242	0.32734	0.30826
12	0.53422	0.44905	0.41918	0.37543	0.33815	0.31408	0.29573
13	0.51490	0.43246	0.40362	0.36143	0.32548	0.30233	0.28466
14	0.49753	0.41760	0.38970	0.34890	0.31417	0.29181	0.27477
15	0.48182	0.40420	0.37713	0.33760	0.30397	0.28233	0.26585
16	0.46750	0.39200	0.36571	0.32733	0.29471	0.27372	0.25774
17	0.45440	0.38085	0.35528	0.31796	0.28627	0.26587	0.25035
18	0.44234	0.37063	0.34569	0.30936	0.27851	0.25867	0.24356
19	0.43119	0.36116	0.33685	0.30142	0.27135	0.25202	0.23731
20	0.42085	0.35240	0.32866	0.29407	0.26473	0.24587	0.23152
25	0.37843	0.31656	0.30349	0.26404	0.23767	0.22074	0.20786
30	0.34672	0.28988	0.27704	0.24170	0.21756	0.20207	0.19029
35	0.32187	0.26898	0.25649	0.22424	0.20184	0.18748	0.17655
40	0.30169	0.25188	0.23993	0.21017	0.18939	0.17610	0.16601
45	0.28482	0.23780	0.22621	0.19842	0.17881	0.16626	0.15673
50	0.27051	0.22585	0.21460	0.18845	0.16982	0.15790	0.14886
OVER 50	$\frac{1.94947}{\sqrt{n}}$	$\frac{1.62762}{\sqrt{n}}$	$\frac{1.51743}{\sqrt{n}}$	$\frac{1.35810}{\sqrt{n}}$	$\frac{1.22385}{\sqrt{n}}$	$\frac{1.13795}{\sqrt{n}}$	$\frac{1.07275}{\sqrt{n}}$

Taula de distribució de Kolmorov - Smirnov

Annex 4

Implementem el mètode de Monte Carlo a la funció $f(x) = e^x$.

```
/* EL METODE DE MONTECARLO */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

float barreja (int *seed);
float estminim (int *seed);
float randu (int *seed);

int main (void){

  int a=0,b=1, n=1000000, seed=1312;
  double sigma, sumsigma=0, sum=0;
  double x,f,integral;
  int i;

  for(i=0;i<n;i++){
    /* Cridem la funcio que genera la successio */
    x=estminim(&seed);
    /* Calculem la funcio f(x) = e^x i els sumatoris */
    f=exp(x);
    sum = sum + f;
    sumsigma = sumsigma + f*f;
  }

  integral = ((double)(b-a)/n)*sum;
  sigma = ((double)(b-a)/n)*sumsigma - integral*integral;
  printf("El valor de la integral es: %f\n", integral);
  printf("El valor de la variancia es: %f\n", sigma);

  return 0;

}
```

Bibliografia

[1] Bratley P., Bennett L. F., Schrage L. E., *A Guide to Simulation,* Springer-Verlag, Nova York (1987).

[2] P.J.Davis i P.Rabinowitz, *Methods of Numerical Integration*, Second Edition (1984).

[3] À.Jorba i J.Masdemont, *Introducció a la simulació,* edicions UPC.

[4] Knuth D. E., *Seminumerical Algorithms. The Art of Computer Programming,* vol. 2. Addison-Wesley, Reading (1981).

[5] D.P. Kroese, T.Taimre i Z.I.Botev, *Handbook of Monte Carlo Methods,* Wiley Series in Probability and Statics (2011, Wiley).

[6] Lewis P. A., Goodman A. S., Miller J. M., *A Pseudo-Random Number Generator for the System/360*, IBM Syst. J. 8 (2), pp. 136146 (1969).

[7] David Márquez, *Probabilitats,* Apunts del curs de Probabilitats de la Universitat de Barcelona.

[8] Park S. K., Miller K.W., *Communications of the ACM,* Random Number Generators: Good Ones are Hard to find. vol. 31, pp. 11921201 (1988).

[9] Peña D., *Estadística. Modelos y Métodos. Fundamentos,* vol. 1. Alianza Universidad Tex- tos, Madrid (1989).

[10] Press W. H., Teukolsky S. A., Vetterling W. T., Flannery B. P.:, *Numerical Recipes in C,* (segona edició). Cambridge Univ. Press, Cambridge (1992).

[11] Carles Rovira, *Estadística,* Apunts de l'assignatura d'Estadística de la Universitat de Barcelona

[12] *SAS User's Guide: Basics, Version 5 Edition.* SAS Institute Inc., Cary, N. C., pp. 278-280 (1985).

[13] *System/360 Scientific Subroutine Package, Version III, Programmer's Manual.,* IBM, White Plains, Nova York, pag. 77 (1968).