Enunciado 1 - Generación de Números y Variables Aleatorias

Joaquín Jiménez López de Castro — jo.jimenez@alumnos.upm.es
Alejandro Sánchez de Castro Fernández — alejandro.sanchezdecastro.fernandez@alumnos.upm.es
Ángel Fragua Baeza — angel.fragua@alumnos.upm.es

13 de octubre de 2021

1. Introducción

En esta práctica se pretende tomar contacto con algunas estrategias para probar la eficacia de generadores de números aleatorios. En concreto, se desea probar la eficacia de un generador congruencial multiplicativo de IMSL. Para ello, se utilizará la librería *TestU01* (Simard, 2013), que contiene herramientas para implementar y probar la eficacia de generadores de números aleatorios. Después se analizarán algunos de los contrastes empleados por la librería.

En la Sección 2, se explicará como se ha implementado y probado el generador de IMSL en la librería *TestU01*. En la Sección 3 se explicarán brevemente tres de los métodos que usa *TestU01* para comprobar la eficacia del generador. Finalmente en la Sección 4 se harán unas conclusiones del trabajo realizado.

2. Desarrollo

2.1. Definición del problema

Los generadores congruenciales fueron introducidos por Lehmer (1951), y tienen una definición recursiva a partir del último número generado:

$$x_{n+1} = (ax_n) \mod m$$

Esta función genera números de forma aleatoria entre $[0, m)$, siendo a un multiplicador, b un sesgo, m el módulo y x_0 la semilla inicial, donde si $b = 0$, se denomina generador congruencial multiplicativo. En la práctica se quiere probar la eficacia del generador congruencial multiplicativo de IMSL, que tiene la siguiente definición:

$$x_{n+1} = (16807x_n) \mod (2^{31} - 1)$$

Para ello, se quieren usar los contrastes de una librería desarrollada en C llamada *TestU01*. En esta librería se encuentran implementados algunos de los generadores de números aleatorios más típicos junto con una serie de baterías de tests.

2.2. Implementación

Para poder usar la librería *TestU01* se debe de seguir un proceso de descarga e instalación que se encuentra perfectamente explicado en su guía.

Una vez instalada la librería, se puede utilizar el fichero *Makefile* que se ve en el Código 1 para compilar y ejecutar el programa principal que probará el generador congruencial multiplicativo de IMSL.

```
all: program

program:
    gcc bat1.c -o bat1 -ltestu01 -lprobdist -lmylib -lm

run:
ifndef SEED
    ./bat1
else
    ./bat1 $(SEED) > results/seed-$(SEED).txt
endif

.EXPORT_ALL_VARIABLES:
LD_LIBRARY_PATH = /usr/local/lib
LIBRARY_PATH = /usr/local/lib
C_INCLUDE_PATH = /usr/local/include
```

Código 1: Makefile del programa

Este fichero *Makefile* permite ejecutar las siguientes funcionalidades:

Command Line

```
        Compilar el programa:
$ make
        Ejecutar la batería de tests sobre el generador multiplicativo de
        IMSL, especificando opcionalmente una semilla para el generador:
$ make run [SEED=n]
```

Todo el código necesario para llevar a cabo la generación de números aleatorios utilizando el generador congruencial multiplicativo de IMSL junto con el estudio de aleatoriedad se encuentra contenido en el archivo *bat1.c*, que se muestra en el Código 2.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include "ulcg.h"
4  #include "gdef.h"
5  #include "unif01.h"
6  #include "bbattery.h"
7
8  #define M 2147483647
9  #define A 16807
10 #define C 0
11 #define DEFAULT_SEED 559079219
12
13 int main (int argc, char* argv[])
14 {
15     int seed = (argc > 1) ?  atoi(argv[1]) : DEFAULT_SEED;
16     unif01_Gen *gen;
17     gen = ulcg_CreateLCG (M, A, C, seed);
18     bbattery_SmallCrush (gen);
19     ulcg_DeleteGen (gen);
20
21     printf("Nuestra semilla: %d\n", seed);
22
23     return 0;
24 }
```

Código 2: bat1.c

Se han declarado una serie de constantes. En la Línea 2.8 se establece el valor del módulo m a $2^{31} - 1 = 2147483647$. En la línea 2.9 se establece el multiplicador a. En la línea 2.10 se establece el sesgo b, que como ya se ha mencionado ha de ser 0. Finalmente, en la línea 2.11 se establece un valor de semilla por defecto.

Para crear el generador, se puede usar el tipo *ulcg_CreateLCG*, que provee la librería, tal y como se puede observar en la línea 2.17, a la que se le pasan los parámetros mencionados. Este generador normaliza los valores al rango $[0, 1)$, puesto que busca crear una distribución uniforme.

2.3. Batería de contrastes

En esta práctica se va a ejecutar la batería de contrastes básica de la librería, *bbattery_SmallCrush*. Cabe decir que algunos de los contrastes asumen que la muestra de números tiene al menos 30 bits de precisión, en caso de no ser así es altamente probable que fallen estos contrastes.

En la batería se aplican los siguientes contrastes:

1. smarsa_BirthdaySpacings con $N = 1, n = 5 \times 106, r = 0, d = 230, t = 2, p = 1$.
2. sknuth_Collision con $N = 1, n = 5 \times 106, r = 0, d = 216, t = 2$.
3. sknuth_Gap con $N = 1, n = 2 \times 105, r = 22, \alpha = 0, \beta = \frac{1}{256}$.
4. sknuth_SimpPoker con $N = 1, n = 4 \times 105, r = 24, d = 64, k = 64$.
5. sknuth_CouponCollector con $N = 1, n = 5 \times 105, r = 26, d = 16$.

6. sknuth_MaxOft con $N = 1, n = 2 \times 106, r = 0, d = 105, t = 6$.
7. svaria_WeightDistrib con $N = 1, n = 2 \times 105, r = 27, k = 256, \alpha = 0, \beta = \frac{1}{8}$.
8. smarsa_MatrixRank con $N = 1, n = 20000, r = 20, s = 10, L = k = 60$.
9. sstring_HammingIndep con $N = 1, n = 5 \times 105, r = 20, s = 10, L = 300, d = 0$.
10. swalk_RandomWalk1 con $N = 1, n = 106, r = 0, s = 30, L0 = 150, L1 = 150$.

Para comprobar la aleatoriedad del generador se han usado seis semillas para generar diferentes secuencias de números: 559079219, 4, 3, 2, 1 y 0. Se han sometido estas seis muestras de números a la batería de tests para comprobar el efecto de las semillas sobre el generador. Como era de esperar, si se usa la semilla 0, todos los contrastes rechazan el generador, pues solamente genera ceros. Para el resto de semillas, siempre se cumple que solo rechazan los mismos tres contrastes:

```
========= Summary results of SmallCrush =========

 Version:          TestU01 1.2.3
 Generator:        ulcg_CreateLCG
 Number of statistics:  15
 Total CPU time:   00:00:05.37
 The following tests gave p-values outside [0.001, 0.9990]:
 (eps  means a value < 1.0e-300):
 (eps1 means a value < 1.0e-15):

       Test                          p-value
 ----------------------------------------------
  1  BirthdaySpacings                  eps
  2  Collision                         eps
  6  MaxOft                            eps
 ----------------------------------------------
 All other tests were passed
```

Estos tres contrastes se analizarán en detalle en la siguiente sección.

3. Análisis de contrastes

3.1. Contraste Máximo-de-t

El contraste está descrito en Knuth (1988). Dada una muestra de números $U = \{U_1, U_2, ...U_n\}$, teniendo $U_i \in (0, 1)$ para cualquier i, sirve para rechazar la hipótesis de que U sigue una distribución uniforme en $(0, 1)$. El procedimiento consiste dividir U en $m = \lfloor \frac{n}{t} \rfloor$ *clusters* $C = \{C_1, ..., C_m\}$, con $C_j = \{U_{jt}, U_{jt+1}, ..., U_{jt+t-1}\}$ y $t \geq 1, j = 1, ..., m$ [1]. Se obtiene $V = \{\text{máx}(C_1), ..., \text{máx}(C_m)\} = \{V_1, ...V_m\}$. La hipótesis se rechaza si el test de Kolmogorov-Smirnov rechaza la hipótesis de que V tiene $F(x) = x^t, 0 \leq x \leq 1$ como función de distribución.

El motivo es que si U sigue una distribución uniforme, entonces:

$$\begin{aligned} F(x) = P(V_j \leq x) = \\ &= P(\text{máx}(\{U_{jt}, U_{jt+1}, ..., U_{jt+t-1}\}) \leq x) = \\ &= P(U_{jt} \leq x)P(U_{jt+1} \leq x)...P(U_{jt+t-1} \leq x) = xx...x = x^t \end{aligned}$$

Este contraste admite variaciones. Una implícita es el parámetro t, donde si $t = 1$, se tiene una mera comprobación de que los valores de U son uniformes. En la librería *TestU01* se usa $t = 6$. También se puede intercambiar el test de Kolmogorov-Smirnov por el contraste χ^2, que es lo que hace *TestU01*.

Es sencillo plantear una muestra que pase el test y tenga un patrón fácilmente observable, pero la utilidad de este contraste reside en rechazar generadores de números aleatorios, no en aceptarlos.

[1] No hay que generar explícitamente los *clusters*, se utilizan como recurso para explicar el funcionamiento del contraste.

3.2. Collision

Este contraste también se encuentra en Knuth (1988). Para entender correctamente el test de *Collision*, podemos imaginarlo como un juego, en el que tenemos m urnas y n bolas que lanzamos de forma aleatoria. Si m es mucho mayor que n, lo más probable es que la mayoría de bolas entren en una urna vacía. Si se da el caso de que la urna ya tenía al menos una bola, se considera que se ha producido una colisión. El test de *Collision* cuenta el número total de colisiones producidas, y si este no es ni demasiado grande, ni demasiado pequeño, el generador superará el test.

Utilizando combinatoria básica, se puede deducir que la probabilidad de que una urna tenga k bolas es de:

$$p_k = \binom{n}{k} m^{-k}(1-m^{-1})^{n-k}$$

Partiendo de la fórmula anterior, se puede calcular cuál es el número estimado de colisiones por urna.

$$\sum_{k>=1}(k-1)p_k = \sum_{k>=0} kp_k - \sum_{k>=1} p_k = \frac{n}{m} - 1 + p_0$$

Sabiendo cuál es el número estimado de colisiones por urna, se puede calcular el número de colisiones totales estimadas. Para ello basta con multiplicar el número estimado de colisiones por urna por el número total de urnas. Esta cuenta es algo costosa, puesto que la probabilidad de que una urna tenga 0 bolas $p_0 = (1-m^{-1})^n = 1 - nm^{-1} + \binom{n}{2}m^{-2} - términos\ menores$ es costosa computacionalmente hablando. Por ello, el número de colisiones se suele estimar usando la fórmula $\frac{n^2}{2m}$.

La potencia de este contraste se basa en su capacidad para medir colisiones en múltiples dimensiones. Para ello basta con trocear la muestra inicial en vectores del mismo tamaño que número de dimensiones queramos usar en el test. Cada uno de estos vectores se usará para indexar una tabla de bits de tamaño m. Esta tabla se encontrará inicializada con todo ceros, y en el momento en el que se acceda a una posición cualquiera si hay un 0 se le transforma en 1, y si ya hay un 1 esto significa que ya se ha visitado esa posición, por lo que se mantiene el 1 y se incrementa en uno el número de colisiones.

En la librería *TestU01* la función encargada de llevar a cabo este contraste sería *sknuth_Collision*, cuya implementación se basa en otro test llamado *power divergence*. Este otro contraste está implementado por la función *smultin_Multinomial*, en donde uno de sus parámetros llamado *Sparse* permite que sus parámetros n y k sean muy grandes, lo cual hará que el número de urnas sea muy grande requisito principal del *Collision test*.

3.3. Birthday spacings

Este contraste fue propuesto por Marsaglia (1985). Parte de la premisa de que, en un conjunto de números ordenados escogidos de forma aleatoria, la distribución asintótica del número de valores duplicados entre los espacios de los números es Poisson con media $\lambda = \frac{m^3}{4n}$

Es decir, se parte de una muestra $U = \{U_1, U_2, ..., U_n\}$ de tamaño m, llamada cumpleaños. Los elementos de esta muestra tomarán valores entre $[0, n)$, siendtextito n el tamaño del espacio sobre el que se generan los elementos de la muestra; también se conoce a n como año. Esta muestra U se ordena de forma ascendente y se toman las diferencias entre los elementos consecutivos de forma $x_{j+1} - x_j$ tal que $1 < j < m$ de la muestra para calcular los espacios. Estos espacios se almacenan en otra muestra $K = \{K_1, K_2, ..., K_n\}$ que vuelve a ordenarse para contar el número de colisiones, o lo que es lo mismo, la cantidad de valores duplicados. Esta cantidad de colisiones debe ajustarse al valor de λ. Este proceso se repite hasta obtener una muestra de 5000 valores de colisiones. Por último, se aplica una bondad de ajuste $\sum \frac{2(observado-esperado)}{esperado}$ y ese valor se contrasta usando la distribución χ^2 adecuada para obtener el p-valor y rechazar o no la hipótesis H0: "El número de colisiones es aproximadamente una variable aleatoria de Poisson con media igual a λ".

No hay estudios teóricos sólidos que encuentren m y n óptimos, pero sí hay generaciones de números aleatorios que cumplen esta regla de forma satisfactoria. Así como tampoco hay estudios concluyentes sobre el tamaño que debe tener n para poder comparar los resultados con la distribución de Poisson con media λ, pero la experiencia marca que para poder aplicar estar regla n debe ser al menos 2^{18}.

Para fallar este test el p-valor debe ser próximo a 0, pero fallar una vez es condición necesaria pero no suficiente. Para comprobar si un generador falla en este test es necesario probar con distintas semillas y ver si el p-valor difiere en órdenes de magnitud. Si un generador falla una vez con una semilla x_i con un p-valor de $5x10^{-10}$ y con otra semilla x_j falla con $5x10^{-6}$ no puede afirmarse que el generador haya fallado el test. En cambio, si falla las dos veces con p-valor $5x10^{-10}$ puede afirmarse que el generador no ha pasado el test Mccullough (2006). Este es uno de los test más difíciles de pasar, si un conjunto pasa este test es muy probable que pase todos los demás.

En cuanto a la implementación, la librería *TestU01* ofrece la función *smarsa_BirthdaySpacings* para poder ejecutar esta prueba individualmente. La función incluye los siguientes parámetros:

$N \rightarrow$ Cantidad de veces que se repite el test sobre muestras de datos distintas del mismo conjunto de número aleatorios.

$n \rightarrow$ Número de "cumpleaños". Tamaño de la muestra que se escoge del conjunto de números.

$d \rightarrow$ Tamaño del espacio.

$t \rightarrow$ Número de dimensiones del espacio.

$p \rightarrow$ Forma de ordenar el espacio. Puede tomar valores en $[1, 2]$.

$r \rightarrow$ Toma valores entre $[0, b]$ siendo b el número de bits que tiene el conjunto números aleatorios. Según este valor se ignoran los r bits más significantes del conjunto de número aleatorios.

En el caso de este trabajo se ha utilizado la batería de pruebas *bbattery_SmallCrush* en el que se ejecuta *smarsa_BirthdaySpacings* con unos valores concretos de sus parámetros. Estos valores son:

$N \rightarrow 1$

$n \rightarrow 5x10^{-10}$

$d \rightarrow 2^{30}$

$t \rightarrow 2$

$p \rightarrow 1$

$r \rightarrow 0$

4. Conclusión

En esta práctica se ha podido utilizar una librería de test de generadores de números aleatorios para comprobar la eficacia del generador congruencial multiplicativo de IMSL. Para ello, se ha pasado la batería de contrastes pequeña y tres tests han rechazado el generador como adecuado. Se han escogido estos tres contrastes para ser analizados y explicados brevemente.

Se concluye que el generador, aunque puede servir para algunos propósitos básicos, no es adecuado para funcionalidades más exigentes, como la criptografía.

Referencias

Knuth, D. E. (1988). *Art of computer programming, volume 2: Seminumerical algorithms*. Addison-Wesley Professional, págs. 70-71.

Lehmer, D. H. (1951). Mathematical Methods in Large-scale Computing Units. En: *Proceedings of the Second Symposium on Large Scale Digital Computing Machinery*. Cambridge, United Kingdom: Harvard University Press, págs. 141-146.

Marsaglia, G. (1985). A current view of random number generators. En: *Computer Science and Statistics: The Interface*, págs. 3-10.

Mccullough, B. D. (2006). A review of TESTU01. En: *Journal of Applied Econometrics* 21.5, págs. 677-682. DOI: 10.1002/jae.917.

Simard, R. J. (2013). A Software Library in ANSI C for Empirical Testing of Random Number Generators. En: