

Simulación

1.1 Generación de números pseudoaleatorios.

1.2 Generadores lineales congruenciales.

Jorge de la Vega Góngora

Departamento de Estadística,
Instituto Tecnológico Autónomo de México

Semana 2 (clases: 21-23 agosto 2018)

1.1 Generación de números pseudoaleatorios.

Un poco de historia

Los métodos de Monte Carlo surgieron durante el proceso de investigación de la bomba atómica en la Segunda Guerra Mundial. Este trabajo involucró una simulación directa de los problemas probabilísticos asociados con la difusión aleatoria de neutrones en materiales fisionables.

También se utilizó para criptografía, en donde se requiere de una fuente confiable de números aleatorios.



Figura: Von Neumann, Feynman y Ulam

John Von Neumann y Stanislaw Ulam aplicaron y refinaron las técnicas de simulación directa con técnicas de reducción de varianza. Además, propusieron métodos para la generación de números *pseudoaleatorios*.

Actualmente, la mayoría de los programas computacionales incluyen rutinas o funciones para generar sucesiones de números pseudo-aleatorios, aunque no describen con detalle los algoritmos con las que son generadas esas sucesiones.

Problema:

Generar una muestra aleatoria de una distribución $\mathcal{U}(0, 1)$.

El problema no es fácil, hay que considerar:

- La distribución uniforme es continua. Sin embargo, cualquier mecanismo numérico obtiene sólo números racionales. \mathbb{Q} es denso en los reales y aún así forman un conjunto de medida 0.
- Las observaciones generadas tienen que ser independientes.
- Cualquier método computacional se basa en un *algoritmo* determinístico, y por lo tanto los números no son estrictamente aleatorios.

- Primeros mecanismos para generar números aleatorios: lanzamiento de dados, extracción de cartas, urnas.
- Maurice Kendall & Babington Smith (1938) usan un disco giratorio para generar una tabla de 100,000 dígitos.
- Rand Corporation (1955): utiliza la primera computadora, ERNIE (Electronic Random Number Indicator Equipment) para generar una tabla con un millón de dígitos.
- En los 40's y 50's, se buscó generar números con métodos numéricos o aritméticos, como **el método del cuadrado medio**.

Este método genera una sucesión de números usando el siguiente algoritmo:

1. Toma un número inicial de 4 dígitos (o 5 o 6 o etc), e.g. $Z_0 = 7182$.
2. Elevar al cuadrado para obtener 8 dígitos, $Z_0^2 = 51581124$
3. Tomar los dígitos centrales para generar un nuevo número: $Z_1 = 5811$
4. El siguiente número aleatorio es $u_1 = Z_1/1000 = 0.5811$
5. Repetir con Z_1 los pasos anteriores.

i	Z_i	u_i	Z_i^2
0	7,182	0.7182	51,581,124
1	5,811	0.5811	33,767,721
2	7,677	0.7677	58,936,329
3	9,363	0.9363	87,665,769
4	6,657	0.6657	44,315,649
\vdots	\vdots	\vdots	\vdots

- El problema con este método es que degenera rápidamente a 0 o bien, comienza a repetir un ciclo.
- N. Metropolis hizo pruebas con números en el sistema binario y demostró que con números de 20 bits hay 13 diferentes ciclos en los que las sucesiones degeneran, de las cuáles la más larga tiene periodo 142.
- *“Anyone, who considers arithmetical methods of producing random digits is of course, in a state of sin.”* Von Neumann (1951).
- Por esta razón, se requiere poner condiciones adecuadas sobre los generadores, para que el pecado sea el menor posible.

- Más recientemente, se cuenta con algunos mecanismos físicos, como la desintegración de un material radiactivo o ruido eléctrico.
 - **Hotbits**: a través de decaimiento radioactivo
 - **IDQ**: fuentes de entropía cuántica.
- Los más frecuentes están basados en circuitos eléctricos dotados de una fuente de ruido (una resistencia o un diodo semiconductor), que es amplificada, muestreada y comparada con una señal de referencia para producir secuencias de bits. Por ejemplo, si el ruido es menor que la referencia se produce un 0 y si es mayor un 1.
- a partir de cadenas de 0's y 1's se generan bytes de información que se traducen a números.
- **Generadores de intel**
- Uno de los problemas más comunes con estos mecanismos es que se desconoce la distribución de los números generados, dificultando su aplicabilidad en algunos problemas.

Los números pseudoaleatorios o sus generadores *deben*:

- Al menos, *parecer* distribuirse con una distribución dada y mantenerse de manera estable en esa distribución.
- ser muy rápidos y eficientes de generar (ya que en la práctica se requieren grandes cantidades de números),
- ser capaces de reproducir secuencias (streams) más de una vez (reproducibilidad),
- poder generar más de una secuencia,
- tener periodo largo (en el sentido que se verá más adelante).

Usualmente los mejores generadores son funciones f de la forma:

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

donde k es el *orden* del generador, y los k valores iniciales son la *semilla*. La longitud de la secuencia que no se repite es el *periodo* del generador.

1.2 Generadores lineales congruenciales.

D. H. Lehmer en 1948 introdujo los *generadores lineales congruenciales*, que simulan una especie de ruleta, y que cumplen con los requerimientos mencionados anteriormente, bajo ciertas condiciones.

- Requieren cuatro parámetros (enteros no negativos):

Z_0 : una semilla o valor inicial

m : el módulo o divisor

a : un multiplicador, y

c : el incremento

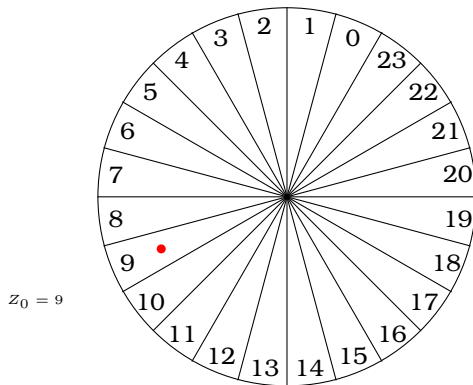
- Se genera una sucesión de valores con la siguiente ecuación:

$$Z_i \equiv (aZ_{i-1} + c) \pmod{m}$$

- De la ecuación anterior, $0 \leq Z_i \leq m - 1$ para cada i , y la serie de números pseudoaleatorios es $u_i = Z_i/m$.
- Cuando $c = 0$, se le llama *generador congruencial multiplicativo*.

Generadores lineales congruenciales

$$m = 24$$
$$Z_i = (3Z_{i-1} + 2) \bmod 24$$



- Los números generados con los GLC's no son aleatorios: hay una fórmula explícita para Z_i en términos de Z_0 , a , c y m (demostrar en la siguiente tarea, utilizando inducción matemática):

$$Z_i \equiv \left(a^i Z_0 + \frac{c(a^i - 1)}{a - 1} \right) \pmod{m}$$

- La calidad del generador depende de la selección de valores a , Z_0 , c y m .
- Cuando se repita un valor de Z_j , los valores comienzan a repetirse. La longitud de valores diferentes generados antes de una repetición se llama **ciclo o periodo**. El ciclo siempre es menor o igual a m .
- Usualmente, se eligen valores de m muy grandes, del orden de 10^9 o mayores.

Ejemplo: $m = 16$, $a = 5$, $c = 3$, $Z_0 = 7$.

$$Z_i \equiv (5Z_{i-1} + 3) \pmod{16}$$

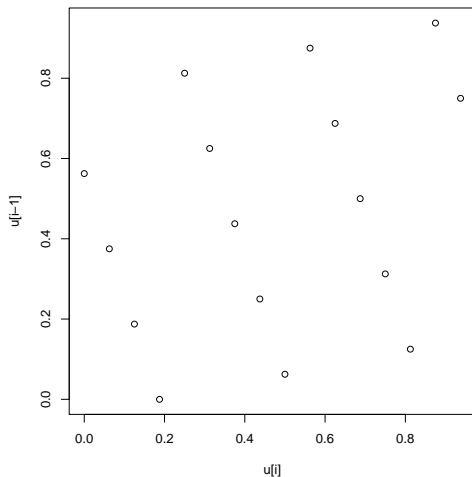
i	Z_i	U_i
0	7	—
1	6	0.375
2	1	0.063
3	8	0.500
4	11	0.688
5	10	0.625
6	5	0.313
7	12	0.750
8	15	0.938
9	14	0.875
10	9	0.563
11	0	0.000
12	3	0.188
13	2	0.125
14	13	0.813
15	4	0.250
16	7	0.438
17	6	0.375
18	1	0.063
19	8	0.500

La calidad de un GLC puede ser vista desde estos ángulos:

- Longitud del periodo del GLC
- Distribución de las muestras : ¿qué distribución tienen los valores generados?
- Independencia de las muestras : las muestras pueden mostrar cierto nivel de dependencia.

Ejemplo: $m = 16$, $a = 5$, $c = 3$, $Z_0 = 7$.

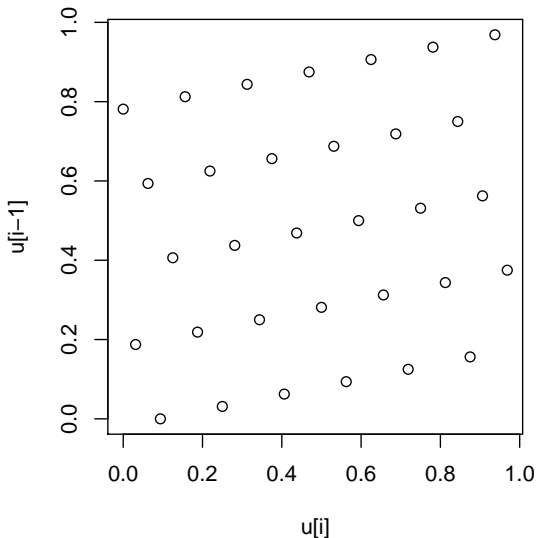
Generador congruencial $m=16,a=5,c=3$



```
lgc <- function(m=16,a=5,c=3,z0=7) {  
  z <- z0  
  i <- 1  
  repeat {  
    i <- i+1  
    z[i] <- (a*z[i-1]+c) %% m  
    if(i>m) break  
  }  
  return(z/m)  
}  
  
z <- lgc()  
par(pty="s")  
plot(z[2:17],z[1:16],xlab="u[i]",ylab="u[i-1]", main =  
      "Generador congruencial m=16,a=5,c=3")
```

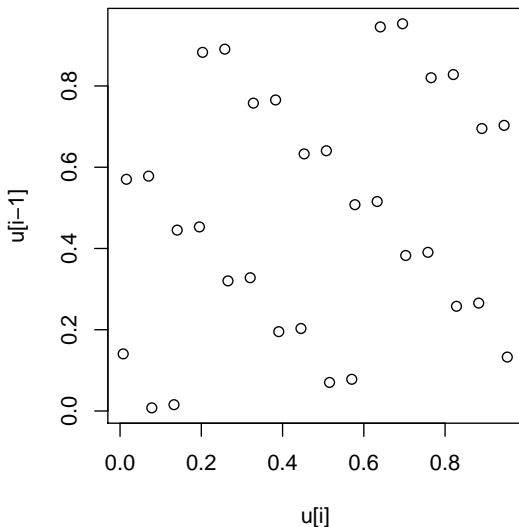

Ejemplo: $m = 32$, $a = 5$, $c = 3$, $Z_0 = 5$.

Generador congruencial $m=32,a=5,c=3$



Ejemplo: $m = 128$, $a = 7$, $c = 3$, $Z_0 = 1$.

Generador congruencial $m=128,a=7,c=3$



- Las gráficas anteriores muestran que es importante hacer una elección adecuada de los parámetros para que el generador tenga características adecuadas.
- Adicionalmente al periodo de los generadores, también hay que hacer múltiples consideraciones.
- Si se ignora el orden secuencial de un GLC de periodo completo, los números parecerán uniformes.
- Sin embargo la estructura de una secuencia de números del GLC es muy regular, y esto puede afectar la correlación serial de los números generados.
- Marsaglia (1968) mostró que cuando se grafican d sucesiones de un GLC con módulo m que se traslapan, los puntos caerán sobre hiperplanos paralelos, y habrá una dirección de algunos hiperplanos en los que no habrá mas de $(d!m)^{1/k}$ hiperplanos paralelos. Las gráficas que mostramos corresponden a $d = 2$.

- ¿Cómo se puede garantizar un periodo completo? El siguiente teorema da la respuesta:

Teorema de periodo completo (Hull & Dobell, 1962)

Un GLC $Z_i \equiv (aZ_{i-1} + c) \pmod m$ tiene periodo completo si y sólo si se cumplen las siguientes tres condiciones:

1. c y m son primos relativos
 2. Si q es un número primo que divide a m , entonces q también divide $a - 1$ ($a \equiv 1 \pmod q$, para cada factor primo de m .)
 3. Si 4 divide a m , entonces 4 divide $a - 1$. ($a \equiv 1 \pmod 4$, si 4 divide a m).
- Noten que si $c = 0$, no se puede conseguir periodo completo (¿porqué?). Por lo tanto, el periodo máximo de un multiplicativo es $m - 1$.
 - No fácil de verificar para los valores relevantes de m , a , c , etc.
 - La prueba del teorema es muy elaborada y basada en varios lemas de teoría de números (Ver: SIAM Review **4** (1962) 230-254).

- $m = 16, a = 5, c = 3, Z_0 = 7$
- $m = 10^{10}, a = 3141592621, c = 2718281829, Z_0 = 5772156648$
- Hay que tener cuidado con la aritmética de enteros grandes en la computadora. Algunos paquetes de R permiten manejar números grandes:
 - **numbers**: funciones de Teoría de Números. en particular las funciones `primeFactors()`, `Primes()`

```
library(numbers)
Primes(1,100)

[1]  2  3  5  7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83
[24] 89 97

primeFactors(2718281829)

[1]      3      157 5771299
```

- **gmp**: Multiple precision arithmetic. Permite hacer operaciones con números grandes.

```
suppressMessages(library(gmp))
factorize(3141592621)

Big Integer ('bigz') object of length 3:
[1] 137 239 95947
```

- Los GLC's son rápidos para generar
- Bajo costo de almacenamiento (computacional)
- Reproducibles (basta conocer la semilla Z_0)
- Se puede reiniciar a la mitad (recordando la última Z_i , y usando Z_0 la siguiente ocasión)
- Múltiples cadenas (salvando semillas separadas)
- Buenas propiedades estadísticas (dependiendo de la elección de los parámetros).

Elección de parámetros desde el punto de vista computacional

- Para lograr una alta densidad en el intervalo $[0, 1]$ los valores de m tienen que ser muy grandes.
- Desde el punto de vista computacional, m puede elegirse de la forma $m = 2^k$ donde k es del tamaño del procesador (en máquinas actuales $k = 64$, antes se tomaba $k = 8, 16, 32$)
- Un truco computacional es aprovechar el desbordamiento numérico para evitar la división numérica por m . En una máquina de $k = 64$ bits, el máximo entero que se puede representar es de la forma $2^k - 1$.
- Para representar un entero mayor H , que ocuparía $h > k$ dígitos binarios, se perderían los $h - k$ dígitos binarios más a la izquierda, y los k dígitos que quedan corresponden precisamente a $H \bmod 2^k$.

Ejemplo

Para $Z_n \equiv 5Z_{n-1} + 3 \bmod 16$, con $Z_0 = 7$, notar que $5Z_9 + 3 = 73$, que en binario es 1001001. Con 4 bits, se pierden los tres primeros dígitos, quedando 1001 que es la representación binaria de $Z_{10} = 9$.

Cuando m tiene una forma particular, algunos resultados se simplifican, como se puede ver en el siguiente teorema.

Teorema

Un GLC con $m = 2^k \geq 4$ tiene periodo completo sí y sólo si c es impar y $a \equiv 1 \pmod{4}$ (esto es, a y 4 son primos relativos).

Ejemplo

Consideren $m = 32$ y $c = 3$, $a_1 = 3$ y $a_2 = 5$. Determinar si este generador tiene periodo completo.

En el caso de los GLC's multiplicativos ($c = 0$), aún se puede obtener un periodo muy grande aunque incompleto eligiendo bien m y a :

Teorema

El periodo máximo de un GLC multiplicativo con $m = 2^k \geq 16$ es 2^{k-2} . Dicho periodo maximal se alcanza si y sólo si Z_0 es impar y $3 = a \pmod 8$ o $5 = a \pmod 8$.

Ejemplo

- Un ejemplo de un mal generador de números aleatorios es RANDU:
 $m = 2^{31}$, $a = 2^{16} + 3$. Este generador se usó mucho tiempo pero se probó en un momento que tiene propiedades estadísticas muy malas.
- Los generadores multiplicativos más famosos utilizados por IBM tomaban $m = 2^{31} - 1$ y $a_1 = 7^5$ o $a_2 = 630360016$.

Ejemplo: generador RANDU

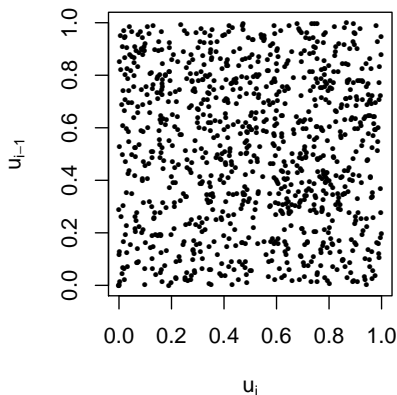
```
options(scipen=9)
seed <- as.double(1)
RANDU <- function() {
  seed <- ((2^16 + 3) * seed) %% (2^31)
  seed / (2^31)
}
randu0 <- NULL
for(i in 1:1000) randu0[i] <- RANDU()
head(randu0, 20)

[1] 0.00003051898 0.00018310966 0.00082398718 0.00329593616 0.01235973230
[6] 0.04449496837 0.15573221957 0.53393860208 0.80204163631 0.00680239918
[11] 0.82243966823 0.87341641681 0.83854148677 0.17050116928 0.47613363480
[16] 0.32229128527 0.64854499837 0.99064842286 0.10698555177 0.72607750492
```

Ejemplo: generador RANDU

Graficamos los pares de números (u_{i-1}, u_i) . Al parecer no forman una retícula.

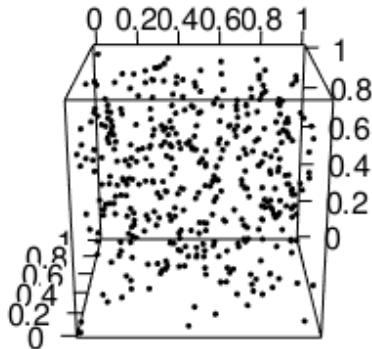
```
par(pty="s")  
plot(randu0[1:999], randu0[2:1000], xlab=expression(u[i]), ylab=expression(u[i-1]), pch=16, cex=0.5)
```



Ejemplo: generador RANDU

Ahora graficamos tripletas de valores consecutivos

```
library(rgl)
seed <- as.double(1)
randul <- NULL
for(i in 1:1200) randul[i] <- RANDU()
plot3d(matrix(randul, ncol=3, byrow=T))
```



Algoritmos con ciclo ampliado

En \mathbb{R} que hay muchos otros algoritmos con ciclos muy amplios. Los algoritmos congruenciales se pueden combinar para aumentar el periodo del ciclo de generación. Estas combinaciones se basan en los siguientes resultados:

Teorema

Si U_1, \dots, U_k son variables aleatorias iid $\mathcal{U}(0, 1)$, entonces la parte fraccional de $U_1 + \dots + U_k$ también sigue una distribución $\mathcal{U}(0, 1)$

$$U_1 + U_2 + \dots + U_k - [U_1 + U_2 + \dots + U_k] \sim \mathcal{U}(0, 1)$$

Demostración: tarea (por inducción)

Teorema

Si u_1, u_2, \dots, u_k están generados por algoritmos congruenciales con ciclos de periodo c_1, c_2, \dots, c_k , respectivamente, entonces la parte fraccional de $u_1 + u_2 + \dots + u_k$ tiene un ciclo de periodo el m.c.m. (c_1, c_2, \dots, c_k) .

Tiene un periodo de orden 10^{12} . El generador se construye a partir de los GLC's:

$$x_i \equiv 171x_{i-1} \pmod{30269}$$

$$y_i \equiv 172y_{i-1} \pmod{30307}$$

$$z_i \equiv 170z_{i-1} \pmod{30323}$$

En este caso se toma

$$u_i = \left(\frac{x_i}{30269} + \frac{y_i}{30307} + \frac{z_i}{30323} \right) - \left\lfloor \frac{x_i}{30269} + \frac{y_i}{30307} + \frac{z_i}{30323} \right\rfloor$$

Este era el algoritmo por default de python 2.3.

- R contiene muchas funciones para generar números aleatorios, basados en los métodos lineales congruenciales, así como en otras familias de generadores.
- El generador por defecto de R es está basado en el algoritmo Mersenne-Twister de Matsumoto y Nishimura (1997), cuyo periodo es de $2^{19937} - 1$. El método está basado en un algoritmo mucho más complejo que el de los GLC's que hemos visto. En python 2.4 y Matlab también es el algoritmo por default (Ver `RandStream.list` para la lista de algoritmos disponibles en Matlab).
- Para cambiar el generador en R:

```
> RNGkind("Super")
```

 #para el super-duper de Marsaglia
- En R el paquete `randtoolbox` tiene otros generadores y pruebas estadísticas. Les recomiendo la lectura de su **documentación**

- Generadores recursivos múltiples (MRG's):

$$Z_i \equiv (a_1 Z_{i-1} + a_2 Z_{i-2} + \cdots + a_{i-k} Z_{i-k}) \pmod{m}$$

- Generadores rezagados de Fibonacci: se escogen r , s y m cuidadosamente:

$$Z_i \equiv (Z_{i-r} + Z_{i-s}) \pmod{m}$$

- Generador congruencial inversivo (ICG): para m primo y con la convención $0^{-1} = 0$:

$$Z_i \equiv \begin{cases} (a_0 + a_1 Z_{i-1}^{-1}) \pmod{m} & \text{si } Z_{i-1} \neq 0 \\ a_0 & \text{si } Z_{i-1} = 0 \end{cases}$$

- Si m es de la forma 2^k , se conocen como linear feedback shift register (LFSR)
- KISS (Keep it simple, stupid!) (Marsaglia y Zaman, 1993) consiste en una combinación de 4 subgeneradores cada uno de 32 bits para alcanzar un periodo de 2^{123} :
 - un GLC mod 2^{32}
 - un generador liner binario
 - dos multiply with carry mod 2^{16}
- **La madre de todos los generadores** (Marsaglia)