Generación de números aleatorios

Generadores congruenciales lineales

El generador de números aleatorios más común, fácil de entender e implementar se llama Generador Congruencial Lineal (GCL) y se define mediante una recursión de la siguiente manera:

$$Z_{n+1} = (aZ_n + c) \mod m, \quad n \geq 0$$
$$U_n = \frac{Z_n}{m}$$

donde $0 < a < m, 0 \leq c < m$ son enteros constantes, y $\mod m$ significa módulo m que implica dividir por m y quedarse con el residuo. Por ejemplo, $6 \mod 4 = 2, 2 \mod 4 = 2, 7 \mod 4 = 3, 12 \mod 4 = 0$. Así, todos los Z_n caen entre 0 y $m-1$; los U_n están, por lo tanto, entre 0 y 1.

Se llama semilla a $0 \leq Z_0 < m$. Se elige m para que sea muy grande, usualmente de la forma $m = 2^{32}$ o $m = 2^{64}$ porque la arquitectura de tu computadora se basa en 32 o 64 bits por palabra; el cálculo del módulo simplemente implica la truncación por la computadora, por lo tanto, es inmediato.

Por ejemplo, si $m = 2^3 = 8$, entonces en binario, hay 8 números que representan $\{0, 1, 2, 3, 4, 5, 6, 7\}$ dados por una terna de $0s$ y $1s$, denotados por

$$(i_0, i_1, i_2) = i_0 2^0 + i_1 2^1 + i_2 2^2, \quad i_j \in \{0, 1\}, \quad j = 0, 1, 2.$$

Así, $(0,0,0) = 0, (1,0,0) = 2^0 = 1, (1,1,0) = 2^0 + 2^1 = 3, (0,0,1) = 2^2 = 4$ y $(1,1,1) = 2^0 + 2^1 + 2^2 = 7$, y así sucesivamente.

Nota cómo $7+1 = 8 = 0 \mod 8$ se calcula mediante truncamiento: $(1,1,1)+(1,0,0) = (0,0,0,1) = 2^3$. El último componente se trunca, resultando en $(0,0,0) = 0$. $10 = (0,1,0,1)$ se trunca a $(0,1,0) = 2$, y así sucesivamente.

Por ejemplo, si $m = 2^3 = 8$, entonces en binario, hay 8 números que representan $\{0, 1, 2, 3, 4, 5, 6, 7\}$ dados por una 3-tupla de 0s y 1s, denotados por

$$(i_0, i_1, i_2) = i_0 2^0 + i_1 2^1 + i_2 2^2, i_j \in \{0, 1\}, j = 0, 1, 2.$$

Así, $(0,0,0) = 0, (1,0,0) = 2^0 = 1, (1,1,0) = 2^0 + 2^1 = 3, (0,0,1) = 2^2 = 4$ y $(1,1,1) = 2^0 + 2^1 + 2^2 = 7$, y así sucesivamente.

Nótese cómo se calcula $7+1 = 8 = 0 \mod 8$ mediante truncamiento: $(1,1,1)+(1,0,0) = (0,0,0,1) = 2^3$. El último componente se trunca dando como resultado $(0,0,0) = 0$. $10 = (0,1,0,1)$ se trunca a $(0,1,0) = 2$, y así sucesivamente.

Aquí hay un ejemplo más típico:

$$Z_{n+1} = (1664525 \times Z_n + 1013904223) \bmod 2^{32}$$

Así $a = 1664525$ y $c = 1013904223$ y

$$m = 2^{32} = 4,294,967,296$$

más de 4.2 billones.

Los números a, c, m deben ser cuidadosamente escogidos para obtener un generador de números 'aleatorios' bueno, en particular querríamos que todos los valores c $0, 1, \ldots, c-1$ sean generados en cuyo caso decimos que el GLC tiene un periodo completo de longitud c. Tales generadores recorrerán cíclicamente los números una y otra vez.

Para ilustrar, considera

$$Z_{n+1} = (5Z_n + 1) \bmod 8, n \geq 0$$

con $Z_0 = 0$. Entonces

$$(Z_0, Z_1, \ldots, Z_7) = (0, 1, 6, 7, 4, 5, 2, 3)$$

y $Z_8 = 16 \bmod 8 = 0$, causando así que la secuencia se repita.

Si aumentamos c a $c = 16$,

$$Z_{n+1} = (5Z_n + 1) \bmod 16, n \geq 0,$$

con $Z_0 = 0$, entonces

$$(Z_0, Z_1, \ldots, Z_{15}) = (0, 1, 6, 15, 12, 13, 2, 11, 8, 9, 14, 7, 4, 5, 10, 3)$$

y $Z_{16} = 16 \bmod 16 = 0$, causando así que la secuencia se repita.

Escoger buenos números a, c, m involucra el uso sofisticado de la teoría de números; números primos y tal, y ha sido extensamente investigado/estudiado por científicos de la computación y matemáticos por muchos años.

Nótese que los números generados son completamente deterministas: Si conoces los valores a, c, m, entonces una vez que conoces un valor (Z_0, digamos) los conoces todos. Pero si te entregan una larga secuencia de los U_n, ciertamente parecen aleatorios, y ese es el punto.

Hay dos tipos principales de GCL:

- **GCL Multiplicativo:** Cuando $c = 0$, se simplifica a $Z_{n+1} = aZ_n \mod m$.
- **GCL Mixto:** Cuando $c \neq 0$, puede ofrecer un período más largo y mejores propiedades estadísticas, dependiendo de los parámetros.

Período completo

Un GCL ideal debería tener un período completo, generando todas las secuencias posibles dentro de su rango antes de repetir cualquier número. Según el Teorema de Hull-Dobell, un GCL tiene un período completo si:

1. c y m son coprimos.
2. $a - 1$ es divisible por todos los factores primos de m.
3. Si m es múltiplo de 4, $a - 1$ debe ser múltiplo de 4.

Método de los cuadrados medios (John von Neumann, 1940s)

Este método fue propuesto por Von Neumann. En este método, tenemos una semilla y luego la semilla se eleva al cuadrado y su término medio se toma como el número aleatorio. Consideremos que tenemos una semilla con N dígitos, elevamos ese número al cuadrado para obtener un número de $2N$ dígitos; si no se convierte en un número de $2N$ dígitos, agregamos ceros antes del número para hacerlo de $2N$ dígitos. Un buen algoritmo es básicamente aquel que no depende de la semilla y el período también debe ser máximamente largo, es decir, debe tocar casi todos los números en su rango antes de empezar a repetirse; como regla general, recuerda que cuanto más largo sea el período, más aleatorio será el número.

Ejemplo:

Considera que la semilla es 14 y queremos un número aleatorio de dos dígitos.

Número	Cuadrado	Término medio
14	0196	19
19	0361	36
36	1296	29
29	0841	84
84	7056	05
05	0025	02
02	0004	00
00	0000	00

En el ejemplo anterior, podemos notar que obtenemos algunos números aleatorios 19, 36, 29, 84, 05, 02, 00 que parecen ser elecciones aleatorias, de esta manera obtenemos múltiples números aleatorios hasta que encontramos una cadena que se repite a sí misma. También nos damos cuenta de una desventaja de este método, que es si encontramos un 0, entonces obtenemos una cadena de 0s a partir de ese punto. Además, considera que obtenemos un número aleatorio 50, el cuadrado será 2500 y los términos medios son 50 de nuevo, y entramos en esta cadena de 50, y a veces podemos encontrar tales cadenas más a menudo lo cual actúa como una desventaja y debido a estas desventajas este método no se utiliza prácticamente para generar números aleatorios.

Implementación en R

```
cuadradosMedios <- function(semilla, n, digitos) {
 numeros <- numeric(n)  # Vector para almacenar los números generados
 actual <- semilla
 for (i in 1:n) {
  # Cuadrado de la semilla
  cuadrado <- actual^2
  # Convertir a cadena y rellenar con ceros a la izquierda para garantizar 2*digitos longitud
  cuadradoStr <- sprintf("%0*d", 2 * digitos, cuadrado)
  # Asegurar que cuadradoStr tenga exactamente 2*digitos de longitud
  if (nchar(cuadradoStr) < 2 * digitos) {
   cuadradoStr <- paste0(rep("0", 2 * digitos - nchar(cuadradoStr)), cuadradoStr)
  }
  # Extraer dígitos del medio
  inicio <- max(1, floor((nchar(cuadradoStr) - digitos) / 2) + 1)
  fin <- inicio + digitos - 1
  numero <- as.numeric(substr(cuadradoStr, inicio, fin))
  # Normalizar y almacenar
  numeros[i] <- numero / 10^digitos
  # Actualizar semilla
  actual <- numero
 }
 return(numeros)
}
cuadradosMedios(14,8,2)
```

Generador de cuadrados medios (Middle-square method)

Comparado con la versión anterior, este algoritmo ofrece un mejor control sobre la longitud de los números involucrados y añade una comprobación para evitar ciclos indeseados, lo que mejora la fiabilidad del método de cuadrados medios para generar números. Sin embargo, sigue inherente al método de cuadrados medios la posibilidad de entrar en ciclos cortos o generar secuencias de ceros, lo cual limita su utilidad en aplicaciones prácticas que requieren una generación de números aleatorios de alta calidad y sin patrones predecibles.

Implementación en R

```
generadorCuadradoMedio <- function(semilla, n, digitos) {
 numeros <- numeric(n)  # Vector para almacenar los nUmeros generados
 actual <- semilla
 for (i in 1:n) {
  # Cuadrado de la semilla
  cuadrado <- actual^2
  # Convertir a cadena y asegurarse de que tenga 2*digitos, rellenando con ceros si es necesario
  cuadradoStr <- sprintf("%0*s", 2*digitos, cuadrado)
  # Longitud deseada del numero (debe ser par)
  longitudDeseada <- 2*digitos
  # Asegurarse de que el cuadrado tenga la longitud deseada, rellenando con ceros
  if (nchar(cuadradoStr) < longitudDeseada) {
   cuadradoStr <- paste0(rep("0", longitudDeseada - nchar(cuadradoStr)), cuadradoStr)
  }
  # Extraer digitos del medio
  inicio <- (nchar(cuadradoStr) / 2) - (digitos / 2) + 1
  final <- (nchar(cuadradoStr) / 2) + (digitos / 2)
  medioStr <- substr(cuadradoStr, inicio, final)
  medio <- as.numeric(medioStr)
  # Verificar si se ha llegado a un ciclo o cero
  if (medio == 0 || medio == actual) {
   warning(paste("Se detecto un ciclo o cero en la iteracion", i))
   break
  }
  # Almacenar el nUmero y actualizar la semilla
  numeros[i] <- medio
  actual <- medio
 }
 return(numeros)
}
```

Método congruencial lineal (Derrick Lehmer, 1951) en R

```
MCL <- function(a, c, m, semilla, n) {
 numeros <- numeric(n)  # Vector para almacenar los números generados
 actual <- semilla  # Inicializa 'actual' con el valor de la semilla

 numeros[1] <- (a*semilla + c) %% m

 for (i in 2:n) {
  numeros[i] <- (a *  numeros[i-1] + c) %% m
 }
 return(numeros)
}



MCL(1.4,20,2,3,100)
```

Método congruencial multiplicativo

Este es un caso especial del Método Congruencial Lineal donde el incremento c es cero.

```
  MCM <- function(a, m, semilla, n) {
   numeros <- numeric(n)
   actual <- semilla
   for (i in 1:n) {
    actual <- (a * actual) %% m
    numeros[i] <- actual / m
   }
   return(numeros)
  }
```

Introducción al generador congruencial cuadrático

El generador congruencial cuadrático es un tipo de generador de números pseudoaleatorios no lineal que introduce un término cuadrático en la relación de recurrencia. La forma general de este generador es:

$$X_{n+1} = (aX_n^2 + bX_n + c) \mod m$$

donde a, b y c son coeficientes constantes, y m es el módulo.

Ventajas

- Mayor complejidad en la secuencia generada en comparación con los generadores lineales.
- Posibilidad de ciclos más largos si se eligen adecuadamente los parámetros.

Desafíos

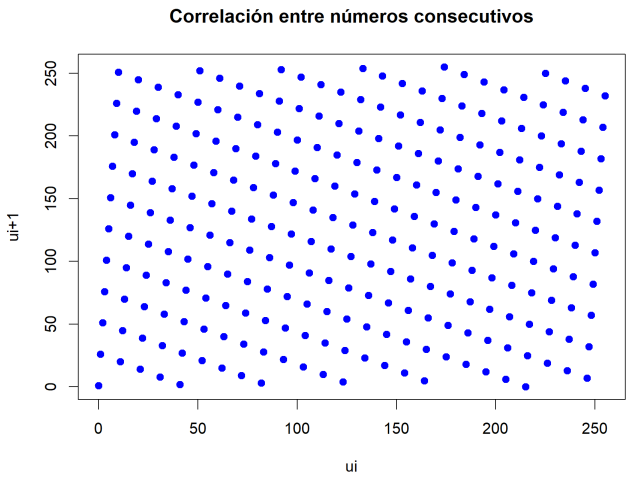
- La elección de los parámetros a, b, c y m es crítica para asegurar la calidad de la secuencia pseudoaleatorias.
- Puede ser más computacionalmente intensivo que los métodos lineales debido a la operación cuadrática.

```
generadorCongruencialCuadratico <- function(a, b, c, m, semilla, n) {
 numeros <- numeric(n)  # Vector para almacenar los números generados
 numeros[1] <- (a*semilla^2 + b*semilla + c) %%m
 for (i in 2:n) {
  numeros[i] <- (a * numeros[i-1]^2 + b * numeros[i-1] + c) %% m
 }
 return(numeros)
}

# Ejemplo de uso
a <- 0.3  # Coeficiente del término cuadrático
b <- 0.7  # Coeficiente del término lineal
c <- 0.5  # Término constante
m <- 5  # Módulo
semilla <- 1.2  # Semilla inicial
n <- 10  # Número de valores pseudoaleatorios a generar

# Llamada a la función
numerosGenerados <- generadorCongruencialCuadratico(a, b, c, m, semilla, n)
print(numerosGenerados)
```

Marsaglia

Los algoritmos deterministas numéricos no producirán números aleatorios verdaderos y que siempre habrá alguna prueba que nuestro generador no superará. Aquí presentamos una prueba interesante que un simple generador aleatorio congruencial no pasa. Tomemos la relación de recurrencia con $M = 2^8$, $a = 25$, $c = 1$. Generamos números aleatorios $(u_0, u_1, u_2, \ldots)$ y los organizamos en pares (u_i, u_{i+1}). Cada par se traza entonces en un plano bidimensional. El resultado se muestra en la figura. Si los números u_i fueran

verdaderamente independientes entre sí, los puntos parecerían distribuirse uniformemente en el plano, mientras que es claro que hay un patrón subyacente. La primera reacción es que deberíamos culpar a nuestra elección de los números (a, c, M), que no fue lo suficientemente buena. De hecho, si repetimos la trama usando los números $(69069, 1, 2^{32})$, el resultado se ve mucho mejor.



Reproducir gráfico en R

```
# 1. Define la función del generador congruencial
generador_congruencial <- function(a, c, M, semilla, n) {
 numeros <- numeric(n) # Vector para almacenar los números
 numeros[1] <- semilla
 for (i in 2:n) {
  numeros[i] <- (a * numeros[i-1] + c) %% M
 }
 return(numeros)
}

# 2. Genera los números aleatorios
# Utilizando los parámetros del ejemplo: a = 25, c = 1, M = 2^8
n <- 1000 # Número de puntos a generar
semilla <- 123 # Semilla inicial para la reproducibilidad
numeros <- generador_congruencial(a = 25, c = 1, M = 2^8, semilla = semilla, n = n)

# 3. Organiza los números en pares
pares <- embed(numeros, 2) # Crea pares de números consecutivos

# 4. Grafica los pares
plot(pares[,2], pares[,1], xlab = "ui", ylab = "ui+1",
```

```
main = "Correlación entre números consecutivos",
pch = 19, col = 'blue') # Utiliza puntos azules pequeños para el trazado
```

¿Existe una manera a priori de elegir el trío (a, c, M) de tal manera que no existan estas correlaciones entre números consecutivos?

Una respuesta sorprendente y negativa fue dada por Marsaglia en un artículo muy interesante con el título sugestivo 'Los números aleatorios caen principalmente en los planos'. En resumen, Marsaglia ha demostrado que todos los generadores congruenciales tendrán correlaciones sutiles. Estas correlaciones se manifiestan cuando organizamos la secuencia de números aleatorios $(u_0, u_1, u_2, \ldots)$ en grupos de d para generar puntos en un espacio d-dimensional $z_1 = (u_0, u_1, \ldots, u_{d-1})$, $z_2 = (u_d, u_{d+1}, \ldots, u_{2d-1})$, y así sucesivamente. El teorema demuestra que existe una dimensión d para la cual todos los puntos $z_1, z_2, \ldots$ caen en un hiperplano de dimensión $d-1$. Esto es bastante molesto y muestra que los generadores congruenciales no pasan una prueba particular para la independencia. El compromiso aceptado es usar un generador que tenga una gran dimensión d para los planos de Marsaglia. Solo entonces podemos aceptar que la presencia de estas correlaciones entre los números no será, esperemos, de ninguna significancia para nuestro cálculo.

Feedback Shift Register

Los generadores de números aleatorios *Feedback Shift Register* (FSR) utilizan ideas similares a los generadores congruenciales pero organizan los números resultantes de una manera diferente. La idea es usar un generador congruencial módulo 2 (así solo produce 0 y 1) pero para aumentar la memoria de la relación de recurrencia. Si denotamos por z_i los diferentes bits, la relación de recurrencia es

$$z_i = c_1 z_{i-1} + c_2 z_{i-2} + \cdots + c_p z_{i-p} \mod 2$$

donde $c_k = 0, 1$ para $k = 1, \ldots, p$, es un conjunto dado de constantes binarias y necesitamos establecer los valores iniciales $(z_0, z_1, \ldots, z_{p-1})$. Los números aleatorios reales se obtienen primero construyendo enteros de b bits uniendo los bits $m_0 = z_0 z_1 \cdots z_{b-1}, m_1 = z_b z_{b+1} \cdots z_{2b-1}$, y así sucesivamente. Los números reales son, como antes, $u_i = \frac{m_i}{2^b}$.

Está claro a partir de la relación de recurrencia que los números z_i necesariamente se repiten después de un período máximo de $L = 2^p$ y por lo tanto el número máximo disponible de números aleatorios distintos es 2^{p-b}. ¿Cómo podemos obtener un período máximo? Un teorema nos dice que un período casi máximo de $2^p - 1$ se obtiene en la relación de recurrencia si y solo si el polinomio

$$f(z) = 1 + c_1 z + c_2 z^2 + \cdots + c_p z^p$$

no puede factorizarse como $f(z) = f_1(z) f_2(z)$ (todas las operaciones son módulo 2). Técnicamente, $f(z)$ se dice que es primitivo en GF(2), el campo de Galois.

Resulta que buenas propiedades estadísticas pueden obtenerse utilizando los polinomios primitivos más simples, aquellos de la forma

$$f(z) = 1 + z^q + z^p$$

con valores adecuados para p y q. La relación de recurrencia se convierte en

$$z_i = z_{i-p} + z_{i-q} \mod 2$$

.

Si, en lugar de aritmética binaria, usamos operaciones lógicas binarias, esta relación es equivalente a

$$z_i = z_{i-p} \oplus z_{i-q}$$

donde $\oplus$ es la operación lógica de 'o exclusivo' (recordemos su tabla: $0 \oplus 0 = 0$, $0 \oplus 1 = 1$, $1 \oplus 0 = 1$, $1 \oplus 1 = 0$). La operación de 'o exclusivo' está incluida en muchos compiladores y tiene la ventaja de que es realmente rápida.

Una posible forma de implementar el algoritmo es trabajar directamente al nivel de los enteros m_i. Primero, se establecen p valores iniciales $m_0, m_1, \ldots, m_{p-1}$, donde m_i son enteros con la precisión deseada (por ejemplo, $b = 31$ bits, para evitar la presencia de números negativos). Esto se implementa utilizando otro generador de números aleatorios. Luego se usa la relación de recurrencia $m_i = m_{i-p} \oplus m_{i-q}$, que el compilador entiende como la operación $\oplus$ realizada al nivel de cada bit de los números enteros. Finalmente, se establece $u_i = \frac{m_i}{2^b}$.

Una vez asegurado el período máximo, ¿qué pares (p, q) darán buenas propiedades estadísticas? Una primera elección sugerida fue $p = 250$, $q = 103$. El generador resultante se llama R250. Su período es $2^{250} \approx 1.8 \times 10^{75}$, lo cual es suficientemente grande para cualquier aplicación. Fue muy popular durante un tiempo hasta que se demostró que algunas correlaciones espurias aparecían al calcular las propiedades de equilibrio del modelo de Ising en redes regulares, probablemente debido a la existencia de planos de Marsaglia de dimensión no lo suficientemente grande. Otros valores que se han sugerido como productores de buenas propiedades estadísticas son $p = 1279$, $q = 418$.

Campo de Galois

En matemáticas, un campo finito o campo de Galois (nombrado en honor a Évariste Galois) es un campo que contiene un número finito de elementos. Como cualquier campo, un campo finito es un conjunto en el cual las operaciones de multiplicación, adición, sustracción y división están definidas y satisfacen ciertas reglas básicas. Los ejemplos más comunes de campos finitos son dados por los enteros módulo p cuando p es un número primo.

El orden de un campo finito es su número de elementos, que es un número primo o una potencia de un primo. Para cada número primo p y cada entero positivo k existen campos de orden p^k, todos los cuales son isomorfos.

Los campos finitos son fundamentales en varias áreas de la matemática y la informática, incluyendo teoría de números, geometría algebraica, teoría de Galois, geometría finita, criptografía y teoría de la codificación.

Ejemplo en Python

```
state = [1,1,1]
fpoly = [3,2]
L = LFSR(initstate=state,fpoly=fpoly,counter_start_zero=False)
print('count \t state \t\toutbit \t seq')
print('-'*50)
for _ in range(15):
print(L.count,L.state,'',L.outbit,L.seq,sep='\t')
L.next()
print('-'*50)
```

```
 count   state   outbit   seq
 --------------------------------------------------
 1       [1 1 1]  1 [1]
 2       [0 1 1]  1 [1 1]
 3       [0 0 1]  1 [1 1 1]
 4       [1 0 0]  0 [1 1 1 0]
 5       [0 1 0]  0 [1 1 1 0 0]
 6       [1 0 1]  1 [1 1 1 0 0 1]
 7       [1 1 0]  0 [1 1 1 0 0 1 0]
 8       [1 1 1]  1 [1 1 1 0 0 1 0 1]
 9       [0 1 1]  1 [1 1 1 0 0 1 0 1 1]
 10      [0 0 1]  1 [1 1 1 0 0 1 0 1 1 1]
 11      [1 0 0]  0 [1 1 1 0 0 1 0 1 1 1 0]
 12      [0 1 0]  0 [1 1 1 0 0 1 0 1 1 1 0 0]
 13      [1 0 1]  1 [1 1 1 0 0 1 0 1 1 1 0 0 1]
 14      [1 1 0]  0 [1 1 1 0 0 1 0 1 1 1 0 0 1 0]
 15      [1 1 1]  1 [1 1 1 0 0 1 0 1 1 1 0 0 1 0 1]
 --------------------------------------------------
 Output:  [1 1 1 0 0 1 0 1 1 1 0 0 1 0 1 1]
```

El LFSR se inicializa con un estado y una función de retroalimentación basada en los bits en las posiciones 3 y 2. A continuación, se detallan los pasos para cada iteración:

1. **Estado inicial:** [**1 1 1**]
2. **Función de retroalimentación:** $f(x) = x_3 \oplus x_2$

3. **Iteraciones:**

Iteración	Estado anterior	Nuevo bit	Nuevo estado	Secuencia generada
1	[1 1 1]	$1 \oplus 1 = 0$	[0 1 1]	[1]
2	[0 1 1]	$0 \oplus 1 = 1$	[1 0 1]	[1 1]
3	[1 0 1]	$1 \oplus 0 = 1$	[1 1 0]	[1 1 1]
4	[1 1 0]	$1 \oplus 1 = 0$	[0 1 1]	[1 1 1 0]
5	[0 1 1]	$0 \oplus 1 = 1$	[1 0 1]	[1 1 1 0 0]

- En la iteración 8, el LFSR regresa a su estado inicial [1 1 1], lo que indica que la secuencia se repetirá a partir de este punto.
- El período de esta secuencia es 7, que es la cantidad de iteraciones únicas antes de que la secuencia se repita.
- La secuencia generada por el LFSR es pseudoaleatoria y determinista, basada en la función de retroalimentación y el estado inicial.

Ejemplo en R

```
LFSR <- function(initial_state, bit_pos1, bit_pos2) {
 # Verificar la longitud del estado inicial
 if (length(initial_state) != 3) {
  stop("El estado inicial debe tener exactamente 3 bits.")
 }

 # Verificar la validez de las posiciones de bits
 if (bit_pos1 < 1 || bit_pos1 > 3 || bit_pos2 < 1 || bit_pos2 > 3) {
  stop("Las posiciones de bits deben estar entre 1 y 3.")
 }

 # Inicializar variables
 state <- initial_state
 sequence <- list(initial_state)
 period <- 0

 repeat {
  # Calcular el nuevo bit con la función XOR
  new_bit <- xor(state[bit_pos1], state[bit_pos2])
  # Desplazar los bits y añadir el nuevo bit
  state <- c(new_bit, head(state, -1))
  period <- period + 1
```

```
  # Verificar si el estado se repite
  if (all(state == initial_state)) {
   break
  }
  # Añadir el estado a la secuencia
  sequence[[length(sequence) + 1]] <- state
 }

 return(list(sequence = sequence, period = period))
}

# Ejemplo de uso
initial_state <- c(1, 0, 1)
bit_pos1 <- 3
bit_pos2 <- 2

result <- LFSR(initial_state, bit_pos1, bit_pos2)
print(result$sequence)
print(paste("Período de la secuencia:", result$period))
```

RCARRY y Lagged Fibonacci Generators

La familia de generadores de Fibonacci rezagados utiliza la relación:

$$m_i = (m_{i-p} \oplus m_{i-q}) \mod M$$

donde $\oplus$ denota una suma, una resta, una multiplicación o un OR exclusivo (realizado bit a bit, en este caso recuperamos los generadores FSR). Al usar una suma o una resta, hay una mezcla de bits y, supuestamente, las propiedades estadísticas son mejores.

Basándose en las ideas de Marsaglia y otros, James propuso el generador llamado RCARRY. Comienza con una secuencia de Fibonacci rezagada con una resta pero añadiendo una resta adicional si m_i es un número negativo. El algoritmo es:

$$m_i = m_{i-r} - m_{i-s} - c_i \mod (M)$$

donde $r > s$ y el residuo es $c_i = 1$ si $m_i \leq 0$ (antes de la operación módulo) y $c_i = 0$ en caso contrario. La resta mezcla los diferentes bits del número entero m_i, y el uso del residuo c_i tiene como objetivo destruir la mayoría de las correlaciones en la secuencia de números aleatorios. Para la inicialización del algoritmo, es necesario dar una secuencia de r números enteros $m_i, i = 1, \ldots, r$. Una elección conveniente es $M = 2^{24}$, $r = 24$, $s = 10$. El período del generador es 48 veces menor que el número de diferentes estados que pueden representarse usando 24 números con 24 bits, o $(2^{24})^{24} \approx 10^{173}$.