Generadores de Números Pseudo-Aleatorios

Pablo Brianese

Julio 2020

1. Introducción

En este trabajo exploraremos la librería estándar `random` de C++, y particularmente uno de los algoritmos que nos ofrece para generar números pseudo-aleatorios: el motor Mersenne Twister `mt19937`.

2. Usos de la aleatoriedad

¿Por qué estamos interesados en el estudio de estos temas?

- Existe una gran familia de algoritmos que utiliza generadores de números pseudo-aleatorios para alcanzar buenos resultados (al menos en un caso promedio), en ella encontramos encontramos:
  - El llamado *método de Monte Carlo* abarca una amplia clase de algoritmos que se basan en técnicas de muestreo aleatorio para obtener soluciones numéricas a problemas de *optimización*, de *integración numérica* y de *muestreo* de distribuciones de probabilidad;
  - La familia de *algoritmos genéticos* utiliza técnicas probabilísticas para obtener soluciones aproximadas a problemas de optimización y de *búsqueda*;
  - El algoritmo Zobrist (utilizado en programas que juegan juegos como el ajedrez y Go) y otros algoritmos de *hashing* que transforman datos de tamaño variable en datos de tamaño fijo.
- Generar *permutaciones aleatorias* es de interés, por ejemplo se utilizan para *evaluar algoritmos* de ordenamiento (`sortbenchmark.org`).
- Las aplicaciones en *estadística* son importantes:
  - control de calidad en la industria manufacturera;
  - testeo de presencia de drogas en una muestra biológica;
  - exámenes para nuevos tratamientos médicos o drogas;
  - auditoría;

    - encuestas.

- También son importantes las aplicaciones en *criptografía*, pero nosotros no veremos técnicas relevantes.

En la página web `random.org` pueden encontrarse testimonios de muy diversos usuarios de números aleatorios.

3. La librería <random> de C++

La librería estándar `<random>` provee herramientas para la generación de números aleatorios, su uso en aplicaciones, y su estudio por expertos.

En la conceptualización de esta librería, no hay simples generadores de números aleatorios. En cambio, su diseño fomenta la separación de dos tipos de funcionalidades. Por un lado encontramos los llamados *motores*. Su finalidad es generar bits (ceros y unos) de forma impredecible, asegurando que la probabilidad de conseguir un bit 0 sea siempre igual a de conseguir un bit 1. Por el otro encontramos las *distribuciones*, objetos que transforman secuencias generadas por un motor (u objetos similares, ya esclareceremos más adelante) en una sucesión de números que siga una distribución de probabilidad específica.

La librería contiene mucho material, pero un usuario casual sólo necesita conocer algunos elementos para poder usarla. De la generación de bits se encargara el motor `mt19937`, que implementa el algoritmo Mersenne Twister. Y las distribuciones básicas que podemos necesitar son `uniform_int_distribution` y `uniform_real_distribution`, que devuelven valores enteros o reales (respectivamente) con distribución uniforme en el rango que especifiquemos. El siguiente ejemplo sencillo hace uso de estos e imprime 16 números aleatorios

```
#include <iostream>
#include <random>
int main() {
  // Inicialización del motor mt19937 con una semilla 1729
  std::mt19937 mt{1729};
  // Creación de una variable aleatoria
  // con distribución uniforme
  // en el rango de números enteros [0, 99]
  std::uniform_int_distribution<int> dist{0, 99};

  for (int i = 0; i < 16; ++i) {
    // Obtenemos los valores de la distribución dist
    // pasandole el motor mt como argumento.
    std::cout << dist(mt) << " ";
  }
  std::cout << std::endl;
}
```

Este programa que presentamos tiene salida *determinista*, y esto es así porque la sucesión generada por `mt19937` es *pseudo-aleatoria*. Es decir, dada una semilla, la sucesión de números aleatorios generada por el motor será siempre la misma. Si este comportamiento no es deseable, queremos intencionalmente que la salida del algoritmo sea *realmente estocástica*, la clase `random_device` ayudará con la generación de semillas *verdaderamente aleatorias* para nuestro motor.

```
#include <iostream>
#include <random>

int main() {
  // Creamos una instancia rd de la clase random_device
  std::random_device rd;
  // y la llamamos para dar una semilla
  // al generador de números pseudo-aleatorios.
  std::mt19937 mt{rd()};
  std::uniform_int_distribution<int> dist{0, 99};
  for (int i = 0; i < 16; ++i) {
    std::cout << dist(mt) << " ";
  }
  std::cout << std::endl;
}
```

Esta clase `random_device` que hemos utilizado es particular. Es un *URBG (Uniform Random Bit Generator)*, pero no es un motor al igual que `mt19937`. Este esta diseñado para ser una interfaz estándar para cualquier *fuente ambiental/física de aleatoriedad* (e.g., `/dev/random` o `/dev/urandom` en Linux; e.g., un instrumento que capta ruido atmosférico; e.g, un generador de ruido por resistencia). Su funcionalidad sigue siendo similar a la de los motores, pero carecen de algunas características.

4. Motores

La librería incluye una cantidad de tipos de motor predefinidos. El tipo `default_random_engine` depende de la implementación, es una caja negra para nosotros. Este es el que por diseño está orientado a un uso casual e inexperto, pero su uso no tiene ventajas claras por sobre `mt19937`. Los demás 9 algoritmos son portables *bit por bit* a través de toda plataforma:

- generadores lineales congruenciales:
  - `minstd_rand0`
  - `minstd_rand`
- motores de tipo Mersenne twister:
  - `mt19937`

    - `mt19937_64`
- motores de tipo substracción con acarreo:
    - `ranlux24_base`
    - `ranlux48_base`
- motores con descarte de bloques:
    - `ranlux24`
    - `ranlux48`
- motores con orden permutado:
    - `knuth_b`

Estos motores comparten varias propiedades, y las comparten en particular con `mt19937`. Todos ellos son pseudo-aleatorios, su salida es determinista. Una instancia `e` de un tipo de motor `E`, produce una *cadena de bits* de longitud $n > 0$ cada vez que se la llama `e()` (por ejemplo, para `mt19937` el número de bits es 32). Esta cadena de bits se codifica como un valor entero positivo de tipo (no-signado) `E::result_type` para minimizar el tiempo de ejecución (para `mt19937` este tipo es `std::uint_fast32_t`). Es más eficiente hacerlo así que entregar los bits uno a uno. Además este entero pertenecerá al rango $[$`E::min()`$, \ldots,$ `E::max()`$]$ (para `mt19937` este intervalo es $[0, \ldots, 2^{32} - 1]$). Pero estas propiedades raramente son necesarias, un objeto motor debería ser usado casi exclusivamente como una *fuente de aleatoriedad* (por ejemplo, para un objeto de *distribución*).

¿Por qué fueron elegidos estos algoritmos? Porque sus características (tiempo de ejecución, tamaño, calidad, etcétera) han sido estudiados cuidadosamente y están bien descriptos en referencias estándar (e.g. Knuth: TAOCP vol. 2). Más aún, sus características son diversas, y dependiendo de la aplicación en cuestión algunas pueden resultar más importantes que otras. Por eso, la elección de un tipo de motor involucra compromisos. Por ejemplo, los generadores congruenciales lineales son pequeños y veloces. Sin embargo, las sucesiones que producen son de menor calidad en comparaciones con las resultantes de generadores de tipo Mersenne twister, cuyo estado es $> 600x$ mayor.

5. Distribuciones

Por el otro encontramos las *distribuciones*, objetos que transforman secuencias generadas por un motor (u objetos similares, ya esclareceremos más adelante) en una sucesión de números que siga una distribución de probabilidad específica. Un objeto de *ditribución* produce una variable aleatoria. Estas moldean las secuencias generadas por los URBG (y también por motores) en sucesiones de números que siguen una distribución de probabilidad específica. Al momento de instanciarlas debemos especificar el *tipo* de los números que formaran la sucesión, y argumentos de construcción que especifican los *parámetros* precisos

que distinguen a esta variable aleatoria de otras en la misma familia. A modo de ejemplo, el siguiente código produce dos secuencias de 16 valores, de tipo double, que siguen distribuciones normales con distintos parámetros

```
#include <iostream>
#include <random>

int main() {
  // Utilizamos un URBG como fuente de aleatoriedad,
  // no un motor
  std::random_device rd;

  // Variable con distribución normal estándar
  // que toma valores de tipo double
  double mean1 = 0, stddev1 = 1;
  std::normal_distribution<double> d1{mean1, stddev1};

  // Variable con distribución normal
  // que toma valores de tipo float
  float mean2 = -2, stddev2 = 0.1415;
  std::normal_distribution<float> d2{mean2, stddev2};

  std::cout << "d1: \n";
  for (int i = 0; i < 16; ++i)
    std::cout << d1(rd) << " ";
  std::cout << std::endl;

  std::cout << "d2: \n";
  for (int i = 0; i < 16; ++i)
    std::cout << d2(rd) << " ";
  std::cout << std::endl;
}
```

La librería contiene una cantidad de distribuciones predefinidas, organizadas en cinco grupos:

- Distribuciones uniformes:
  - `uniform_int_distribution`
  - `uniform_real_distribution`
- Distribuciones de tipo Bernoulli:
  - `bernoulli_distribution`
  - `binomial_distribution`
  - `negative_binomial_distribution`
  - `geometric_distribution`

- Distribuciones de tipo Poisson:
  - `poisson_distribution`
  - `exponential_distribution`
  - `gamma_distribution`
  - `weibull_distribution`
  - `extreme_value_distribution`
- Distribuciones de tipo Normal:
  - `normal_distribution`
  - `lognormal_distribution`
  - `chi_squared_distribution`
  - `cauchy_distribution`
  - `fisher_f_distribution`
  - `student_t_distribution`
- Distribuciones de muestreo:
  - `discrete_distribution`
  - `piecewise_constant_distribution`
  - `piecewise_linear_distribution`

Mientras que la mayoría de los motores de la librería estándar son portables bit por bit a través de las distintas plataformas, los resultados de las distribuciones no están determinados de ese modo. Esto permite, en cada implemetación, elegir los algoritmos apropiados para cada plataforma.

6. Funcionalidades importantes: interoperabilidad y extensibilidad

Un aspecto interesante de la biblioteca es que por diseño es extensible. Para quienes deseen ampliarla con sus propios motores y distribuciones, la biblioteca incluye seis plantillas de clases configurables de *motores / adaptadores de motores*, además de una plantilla de función para nuevas *distribuciones*. En la misma dirección, para que estas creaciones puedan interactuar sin fricciones con las presentes el la librería estándar, por diseño cualquier URBG (incluído cualquier motor) puede ser usado con cualquier distribución, sean estos y estas parte de la librería o de diseño personalizado.

7. Mersenne Twister

El algoritmo *Mersenne Twister*, un generador de números pseudo aleatorios de propósito general muy popular, fue desarrollado en 1997 por Makoto Matsumoto y Takuji Nishimura.

Una de las propiedades que dan testimonio de su calidad, es que está *k-distribuído* con precisión de 32 bits para todo $1 \leq k \leq 623$. De forma precisa, denotemos por x_i a la sucesión pseudo-aleatoria de enteros de w-bits ($w = 32$ en nuestro caso) generada y por P a su período ($P = 2^{19937} - 1$). Si $\text{trunc}_v(x)$ es el número formado por los v bits más significativos de x ($1 \leq v \leq 32$), y consideramos P de los vectores formados por $v \cdot k$ bits ($1 \leq k \leq 623$) dados por

$$(\text{trunc}_v(x_i), \text{trunc}_v(x_{i+1}), \ldots, \text{trunc}_v(x_{i+k-1})) \qquad (0 \leq i < P) \qquad (1)$$

Entonces, cada una de las 2^{vk} posibles combinaciones de bits ocurre el mismo número de veces en un período, excepto por la combinación nula $(0 \cdots 0)$ que ocurre una vez menos.

8. Descripción del algoritmo

Utilizamos notación vectorial, como en $\mathbf{x}$ y $\mathbf{a}$, para denotar *palabras*, vectores fila de dimensión w sobre el cuerpo de dos elementos $\mathbb{F}_2 = \{0, 1\}$. Además, las identificamos con palabras de bits de longitud w (estando el bit menos significativo del lado derecho), la unidad de datos propia de un procesador.

El algoritmo MT genera una sucesión de palabras, que son consideradas enteros (no signados) pseudo-aleatorios entre 0 y $2^w - 1$.

El algoritmo esta basado en la siguiente relación de recurrencia lineal

$$\mathbf{x}_{k+n} := \mathbf{x}_{k+m} \oplus ((\mathbf{x}_k^u \,||\, \mathbf{x}_{k+1}^l)A) \qquad (k = 0, 1, \ldots) \qquad (2)$$

La notación es como sigue. Varios símbolos representan constantes: un entero n, que es el grado de la recurrencia, un entero $r \in [0, \ldots, w[$ (escondido en la definición de $\mathbf{x}_k^u$), un entero $m \in [1, \ldots, n]$, y una matriz $A \in \mathbb{F}_2^{w \times w}$. Damos $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{n-1}$ como semillas iniciales. Luego, el generador produce $\mathbf{x}_n$ siguiendo la recurrencia 2 con $k = 0$. Reemplazando por $k = 1, 2, \ldots$, el generador determina $\mathbf{x}_{n+1}, \mathbf{x}_{n+2}, \ldots$. En el lado derecho de la recurrencia 2, $\mathbf{x}_k^u$ hace referencia a los $w - r$ bits más significativos de $\mathbf{x}_k$, y $\mathbf{x}_{k+1}^l$ a los r bits menos significativos de $\mathbf{x}_{k+1}$. Así, si $\mathbf{x} = (x_{w-1}, x_{w-2}, \ldots, x_0)$, entonces por definición $\mathbf{x}^u$ es el vector de $w - r$ bits $(x_{w-1}, \ldots, x_r)$ y $\mathbf{x}^l$ es el vector de r bits $(x_{r-1}, \ldots, x_0)$. La expresión $(\mathbf{x}_k^l \,||\, \mathbf{x}_{k+1}^l)$ representa su concatenación; a saber, esta es la palabra obtenida concatenando los $w - r$ bits más significativos de $\mathbf{x}_k$ y los r bits menos significativos de x_k en este orden. Luego la matriz A multiplica por derecha a este vector. Finalmente sumamos $\mathbf{x}_{k+m}$ a este vector ($\oplus$ es la adición bit a bit módulo 2), y obtenemos el siguiente vector $\mathbf{x}_{k+n}$.

La forma de la matriz A fue elegida para que el cálculo del producto sea muy veloz. Específicamente, tomando la matriz $w \times w$

$$A = \begin{pmatrix} & & 1 & \\ & & & \ddots \\ & & & 1 \\ a_{w-1} & a_{w-2} & \cdots & a_0 \end{pmatrix} \tag{3}$$

el cálculo de $\mathbf{x}A$ puede hacerse utilizando operaciones sobre bits

$$\mathbf{x}A = \begin{cases} (\mathbf{x} \gg 1) & \text{si } x_0 = 0 \\ (\mathbf{x} \gg 1) \oplus \mathbf{a} & \text{si } x_0 = 1 \end{cases} \tag{4}$$

donde $\mathbf{a} = (a_{w-1}, a_{w-2}, \ldots, a_0)$, $\mathbf{x} = (x_{w-1}, x_{w-2}, \ldots, x_0)$. Además, los vectores $\mathbf{x}_k^u$ y $\mathbf{x}_{k+1}^l$ de la recurrencia 2 pueden ser calculados con la operación de conjunción bit a bit.

Para mejorar la k-distribución a una precisión de v bits, multiplicamos cada palabra generada por una matriz $w \times w$ invertible T adecuada (llamada *templamiento*). Esta matriz queda determinada como transformación lineal $\mathbf{x} \mapsto \mathbf{x}T$ por las sucesivas transformaciones

$$\mathbf{y} := \mathbf{x} \oplus (\mathbf{x} \gg u) \tag{5}$$
$$\mathbf{y} := \mathbf{y} \oplus ((\mathbf{y} \ll s) \,\&\, \mathbf{b}) \tag{6}$$
$$\mathbf{y} := \mathbf{y} \oplus ((\mathbf{y} \ll t) \,\&\, \mathbf{c}) \tag{7}$$
$$\mathbf{z} := \mathbf{y} \oplus (y \gg l) \tag{8}$$

donde l, s, t, y u son enteros, $\mathbf{b}$ y $\mathbf{c}$ son máscaras de bits adecuadas con el tamaño de una palabra, y $\mathbf{x} \gg u$ denota un corrimiento a derecha ($\mathbf{x} \ll u$ un corrimiento a izquierda) de u bits.

9. Implementación en C

La implementación depende de dos clases de parámetros: 1. *parámetros de período* que determinan el período: parametros enteros w (tamaño de palabra), n (grado de la recursión), m (término medio), r (punto de separación de una palabra), y un parámetro vectorial $\mathbf{a}$ (la matriz A), y 2. *parámetros de templamiento* para conseguir k-distribución con precisión de v bits: parámetros enteros l, u, s, t, y los parámetros vectoriales $\mathbf{b}$, $\mathbf{c}$.

Los parámetros de período que especifican la relación de recurrencia 2 se ingresan como directivas al preprocesador. El grado de la recursión es $n = 624$ y el término medio es $m = 397$. El punto de separación de una palabra elegido es $31 = 32-1$. Esto lo podemos ver en las elecciones de las máscaras superiores e inferiores como $\mathbf{u}$ igual a `0x80000000` igual a 10000000000000000000000000000000, y $\mathbf{l}$ igual a `0x7fffffff` igual a 01111111111111111111111111111111. Con estas

máscaras, dadas palabras $\mathbf{x}$ y $\mathbf{z}$, podemos calcular el vector $\mathbf{x}^u \,||\, \mathbf{z}^l$. Por un lado $\mathbf{x} \,\&\, \mathbf{u}$ es igual a $(x_{w-1} \,\&\, 1)(x_{w-2} \,\&\, 0) \cdots (x_0 \,\&\, 0)$ es igual a $x_{w-1}0 \cdots 0$. Por otro $\mathbf{z} \,\&\, \mathbf{l}$ es igual a $(z_{w-1} \,\&\, 0)(z_{w-2} \,\&\, 1) \cdots (z_0 \,\&\, 1)$ es igual a $0z_{w-2} \cdots z_0$. Entonces $(\mathbf{x} \,\&\, \mathbf{u}) \,|\, (\mathbf{z} \,\&\, \mathbf{l})$ es igual a $(x_{w-1} \,|\, 0)(0 \,|\, z_{w-2}) \cdots (0 \,|\, z_0)$ es igual a $x_{w-1}z_{w-2} \cdots z_0$. El vector $\mathbf{a}$ que determina la matriz A es 10011001000010001011000011011111.

```
/* Period parameters */

#define N 624
#define M 397
#define MATRIX_A 0x9908b0df   /* constant vector a */
#define UPPER_MASK 0x80000000 /* most significant w-r bits */
#define LOWER_MASK 0x7fffffff /* least significant r bits */
```

Los parámetros de templamiento que especifican la transformación lineal T también se ingresan como directivas al preprocesador. Los parámetros enteros son $u = 11$, $s = 7$, $t = 15$, $l = 18$. Los parámetros vectoriales son $\mathbf{b}$ igual a 10011101001011000101011010000000, y $\mathbf{c}$ igual a 11101111110001100000000000000000.

```
/* Tempering parameters */
#define TEMPERING_MASK_B 0x9d2c5680
#define TEMPERING_MASK_C 0xefc60000
#define TEMPERING_SHIFT_U(y)  (y >> 11)
#define TEMPERING_SHIFT_S(y)  (y << 7)
#define TEMPERING_SHIFT_T(y)  (y << 15)
#define TEMPERING_SHIFT_L(y)  (y >> 18)
```

Para ejecutar la recurrencia necesitamos un área de trabajo, lugar en la memoria, de n palabras ($624 \times 32 = 19968$ bits). Esto es, un arreglo $\mathbf{X} = (\mathbf{x}_0, \ldots, \mathbf{x}_{n-1})$ de enteros largos no-signados. Aquí estamos suponiendo que trabajamos con una arquitectura de 32 bits ($w = 32$). Para inicializar el algoritmo con una única palabra (entero no-signado largo) ¡y evitarnos econtrar una fuente de 623 de ellas! pasamos esta a un generador congruencial multiplicativo $\mathbf{x}_{i+1} = 69069\mathbf{x}_i \bmod 2^{32}$ implementado en una función `sgenrand`.

Observar que la operación de conjunción lógica & con el entero no-signado `0xffffffff` no tiene ningún efecto, porque este último es idéndicamente igual a 1 como vector de bits.

```
static unsigned long mt[N];  /* the array for the state vector */
static int mti=N+1;  /* mti==N+1 means mt[N] is not initialized */

/* initializing the array with NONZERO seed */
void
sgenrand(seed)
     unsigned long seed;
{
  /* setting initial seeds to mt[N] using         */
```

```
  /* the generator Line 25 of Table 1 in          */
  /* [KNUTH 1981, The Art of Computer Programming */
  /* Vol. 2 (2nd Ed.), pp102]                     */
  mt[0] = seed & 0xffffffff;
  for (mti = 1; mti < N; mti++)
    mt[mti] = (69069 * mt[mti-1]) & 0xffffffff;
}
```

Ahora implementamos la función `genrand` que actualiza el estado del generador y devuelve un número pseudo-aleatorio `y`

```
unsigned long /* for integer generation */
genrand()
{
  unsigned long y;
```

La alternativa en el cálculo del producto matricial 4 se codifica como un arreglo `mag01` con dos entradas, una corresponde a la acción sobre un vector $\mathbf{x} = (x_{w-1}, x_{w-2}, \ldots, x_0)$ en el caso $x_0 = 0$ y la otra al caso $x_0 = 1$. De este modo $x_0 = \mathbf{x} \,\&\, 1$ (interpretando vectores de bits como enteros no-signados y viceversa) y el producto xA puede calcularse como $(\mathbf{x} \gg 1) \oplus \texttt{mag01}[\mathbf{x} \,\&\, 1]$.

```
  static unsigned long mag01[2]={0x0, MATRIX_A};
  /* mag01[x] = x * MATRIX_A for x=0,1 */
```

Para evitar el uso de una operación numérica compleja como tomar módulo, la función actualiza el estado sólo cada n $(= 624)$ llamadas. La primera vez que se la llama lo hace, porque al finalizar la inicialización con la función `sgenrand` la variable auxiliar `mti` tiene valor n. Además, se verifica la existencia de una semilla y de no haberla se escoge una por defecto.

```
  if (mti >= N) { /* generate N words at one time */
    int kk;

    if (mti == N+1)   /* if sgenrand() has not been called,  */
      sgenrand(4357); /* a default initial seed is used      */
```

Cada elemento del estado $\mathbf{X} = (\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}) = (\mathbf{x}_k)$ es actualizado. Primero se actualizan aquellos tales que $\mathbf{k} + m < n$, es decir para los cuales $\mathbf{x}_{k+m}$ es un elemento del estado $\mathbf{X}$. Luego se actualizan aquellos tales que $n-m \leq \mathbf{k} < n-1$, es decir para los cuales $\mathbf{x}_{k+(m-n)}$ es un elemento del estado $\mathbf{X}$. Finalmente se actualiza el vector $\mathbf{x}_{n-1}$ de forma individual porque los índices $n-1$ y 0 a los que tenemos que acceder no son contiguos.

Expliquemos de qué modo las actualizaciones siguen la recurrencia $\mathbf{x}_{k+n} := \mathbf{x}_{k+m} \oplus ((\mathbf{x}^u_k \,||\, \mathbf{x}^l_{k+1})A)$. Para hacerlo escribimos $\mathbf{X}' = (\mathbf{x}'_0, \mathbf{x}'_1, \ldots, \mathbf{x}'_{n-1})$ para referirnos al nuevo estado que reemplazará a $\mathbf{X}$. Antes que nada observamos es que $\mathbf{X}, \mathbf{X}'$ siguen un orden $\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}, \mathbf{x}'_0, \ldots, \mathbf{x}'_{n-1}$ dentro de la sucesión de números pseudo-aleatorios generada por el algoritmo. Esto nos permite orientarnos a la hora de expresar la recurrencia.

En el primer bucle `for` tenemos $0 \leq k < n - m$. Aquí para calcular $\mathbf{x}'_k$ según la recurrencia necesitamos del miembro del orden que lo precede en n posiciones $\mathbf{x}_k$ y de su sucesor $\mathbf{x}_{k+1}$ (con $k + 1 \leq n - 1$ porque $k < n - m$); también necesitamos del elemento $\mathbf{x}_{k+m}$ que se encuentra m posiciones por delante de $\mathbf{x}_k$ (con $k + m \leq n - 1$ porque $k < n - m$). Luego $\mathbf{x}'_k$ es igual a $\mathbf{x}_{k+m} \oplus (\mathbf{y} \gg 1) \oplus \texttt{mag01}[\mathbf{y} \,\&\, 1]$ siendo $\mathbf{y} = \mathbf{x}^u_k \,||\, \mathbf{x}^l_{k+1}$.

En el segundo bucle `for` tenemos $n - m \leq k < n - 1$. Al igual que en el paso anterior, para calcular $\mathbf{x}'_k$ según la recurrencia necesitamos del miembro del orden que lo precede en n posiciones $\mathbf{x}_k$ y de su sucesor $\mathbf{x}_{k+1}$ (con $k+1 \leq n-1$ porque $k < n - 1$). A diferencia del caso anterior, el elemento que se encuentra m posiciones por delante de $\mathbf{x}_k$ (siendo $n - m \leq k$) no pertenece al estado $\mathbf{X}$ si no a $\mathbf{X}'$ y es $\mathbf{x}'_{k+(m-n)}$ (con $0 \leq k + (m - n)$ porque $n - m \leq k$, y $k + (m - n) \leq n - 1$ porque $k \leq n - 1$). Aquí no hay ningún error y ningún problema, porque el primero de estos vectores que necesitamos es $\mathbf{x}'_{(n-m)+(m-n)} = \mathbf{x}'_0$ y se calcula en el paso anterior; y porque a partir de ahí el vector $\mathbf{x}'_{k+(m-n)}$ con $k + (m - n) < k$ fue calculado antes de llegar a $\mathbf{x}'_k$. Luego $\mathbf{x}'_k$ es igual a $\mathbf{x}'_{k+(m-n)} \oplus (\mathbf{y} \gg 1) \oplus \texttt{mag01}[\mathbf{y} \,\&\, 1]$ siendo $\mathbf{y} = \mathbf{x}^u_k \,||\, \mathbf{x}^l_{k+1}$.

Finalmente, cuando para calcular el último vector $\mathbf{x}'_{n-1}$ necesitamos del miembro del orden que lo precede en n posiciones $\mathbf{x}_{n-1}$ y de su sucesor $\mathbf{x}'_0$ (en este caso sus índices no son contiguos). También necesitamos el elemento $\mathbf{x}'_{m-1}$ que se encuentra m posiciones por delante de $\mathbf{x}_{n-1}$. Luego $\mathbf{x}'_{n-1}$ es igual a $\mathbf{x}'_{m-1} \oplus (\mathbf{y} \gg 1) \oplus \texttt{mag01}[\mathbf{y} \,\&\, 1]$ siendo $\mathbf{y} = \mathbf{x}^u_{n-1} \,||\, \mathbf{x}'^l_0$.

```
    for (kk = 0; kk < N-M; kk++) {
      y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
      mt[kk] = mt[kk+M] ^ (y >> 1) ^ mag01[y & 0x1];
    }
    for(; kk < N-1; kk++) {
      y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
      mt[kk] = mt[kk + (M-N)] ^ (y >> 1) ^ mag01[y & 0x1];
    }
  y = (mt[N-1]&UPPER_MASK) | (mt[0]&LOWER_MASK);
  mt[N-1] = mt[M-1] ^ (y >> 1) ^ mag01[y & 0x1];
```

El condicional termina reiniciando la variable auxiliar `mti` a 0, lo cual hará que la proxima vez que llamemos a la función `genrand` no volvamos a actualizar el estado.

```
  mti = 0;
  }
```

Guardamos la palabra `mt[mti]` en la variable `y`, luego aumentamos la variable auxiliar `mti` (que determina cuando volvemos a actualizar el estado del generador). Aplicamos la transformación de templamiento a `mti` y devolvemos el resultado.

```
  y = mt[mti++];
```

```
  y ^= TEMPERING_SHIFT_U(y);
  y ^= TEMPERING_SHIFT_S(y) & TEMPERING_MASK_B;
  y ^= TEMPERING_SHIFT_T(y) & TEMPERING_MASK_C;
  y ^= TEMPERING_SHIFT_L(y);

  return y; /* for integer generation */
}
```

Para explorar un poco estas funciones que describimos incluímos un pequeño programa que genera 1000 números aleatorios

```
/* this main() outputs first 1000 generated numbers */
int main()
{
  sgenrand(4357); /* any nonzero integer can be used as a seed */
  for (int j = 0; j < 1000; j++) {
    printf("%lu ", genrand());
    if (j % 8 == 7) printf("\n");
  }
  printf("\n");
}
```

10. Referencias

Tipo: presentación; Título: *rand() Considered Harmful*; Autor: Stefan T. Lavavej; Dónde: GoingNative 2013; Filminas: `https://onedrive.live.com/view.aspx?resid=E66E02DC83EFB165!312&cid=e66e02dc83efb165&lor=shortUrl`

Tipo: presentación; Título: *What C++ Programmers Need to Know about Header <random>*; Autor: Walter E. Brown; Dónde: CppCon 2016; Filminas: `https://github.com/cppcon/cppcon2016`

Tipo: presentación; Título: *I Just Wanted a Random Integer!*; Autor: Cheinan Marks; Dónde: CppCon 2016; Filminas: `https://github.com/cppcon/cppcon2016`

Tipo: Seminario; Título: *PCG: A Family of Better Random Number Generators*; Autor: Melissa O'Neill; Dónde: Colloquium on Computer Systems Seminar Series, Universidad de Stanford;

Tipo: página web; URL: `https://www.random.org` Autor: Mads Haar;

Tipo: publicación académica; Título: *Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator*; Autores: Makoto Matsumoto y Takuji Nishimura; Dónde: ACM Transactions on Modeling and Computer Simulations: Special Issue on Uniform Random Number Generation (1998); URL: `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/mt.pdf`

Tipo: artículo; Título: *Random Number Generation in C++11*; Autor: Walter E. Brown; URL: `https://isocpp.org/files/papers/n3551.pdf`.