Implementación de rand10() utilizando rand7()

Adam Bourbahh Romero

22 de Junio de 2025

1 Introducción y Definiciones

Imagine que tenemos a nuestra disposición una función, rand7(), que nos entrega un número entero aleatorio X, perfectamente distribuido de forma uniforme, dentro del conjunto $\{1, 2, 3, 4, 5, 6, 7\}$. Esto significa que cada uno de estos siete números tiene exactamente la misma probabilidad de ser elegido, es decir, $P(X = k) = \frac{1}{7}$ para cualquier k en ese rango.

Nuestro desafío es diseñar una nueva función, rand10(), que nos dé un número entero aleatorio Y, también uniformemente distribuido, pero esta vez en el rango $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$. Para lograrlo, cada m en este conjunto de diez números debe tener una probabilidad de $P(Y = m) = \frac{1}{10}$.

2 Construcción de un Espacio Muestral Uniforme Ampliado

Una sola llamada a rand7() no es suficiente para generar un número en el rango $[1, 10]$ de manera uniforme; simplemente, 7 es menor que 10. Para sortear esto, necesitamos una fuente de aleatoriedad más amplia. Lo haremos combinando el resultado de varias llamadas a rand7().

Pensemos en dos llamadas independientes a nuestra función base: $R_1 = \text{rand7()}$ y $R_2 = \text{rand7()}$. Con estas, podemos construir una nueva variable aleatoria Z de la siguiente manera:

$$Z = (R_1 - 1) \cdot 7 + R_2$$

Veamos cómo se comporta Z:

- $R_1 - 1$ transforma el resultado de rand7() al rango $\{0, 1, \ldots, 6\}$, cada uno con probabilidad $\frac{1}{7}$.
- Al multiplicar por 7, $(R_1 - 1) \cdot 7$ nos da valores en $\{0, 7, 14, \ldots, 42\}$, también con probabilidad $\frac{1}{7}$.
- R_2 sigue en su rango original $\{1, 2, \ldots, 7\}$, con probabilidad $\frac{1}{7}$.

La variable Z resultante tomará valores enteros desde el más bajo, $(1 - 1) \cdot 7 + 1 = 1$ (cuando $R_1 = 1, R_2 = 1$), hasta el más alto, $(7 - 1) \cdot 7 + 7 = 42 + 7 = 49$ (cuando $R_1 = 7, R_2 = 7$). Dado que R_1 y R_2 son completamente independientes y uniformemente distribuidas, cada combinación posible (R_1, R_2) tiene una probabilidad de $\frac{1}{7} \cdot \frac{1}{7} = \frac{1}{49}$. Y como cada valor de Z se corresponde de forma única con un par (R_1, R_2), esto significa que Z se distribuye uniformemente en el conjunto $\{1, 2, \ldots, 49\}$. Así, para cualquier $k \in \{1, \ldots, 49\}$, $P(Z = k) = \frac{1}{49}$.

3 Muestreo por Rechazo (Rejection Sampling)

El rango de Z es $[1, 49]$, pero este no es un múltiplo exacto de 10. Si intentáramos simplemente usar una operación de módulo 10 sobre Z, la distribución resultante no sería uniforme en $[1, 10]$. Esto se debe a que algunos números (del 1 al 9) aparecerían 5 veces en el rango $[1, 49]$ (por ejemplo, $1, 11, 21, 31, 41$ todos darían 1 al aplicar el mapeo), mientras que el 10 solo aparecería 4 veces $(10, 20, 30, 40)$. Esto introduciría un molesto sesgo.

Para asegurar una distribución uniforme, emplearemos una técnica ingeniosa llamada **muestreo por rechazo**. Necesitamos un rango de valores de Z que sí sea un múltiplo de 10. El múltiplo de 10 más grande que no excede 49 es 40. Entonces, si el valor de Z que generamos cae dentro del rango $[1, 40]$, lo aceptamos. Si Z resulta estar en el rango $[41, 49]$, lo descartamos y simplemente repetimos todo el proceso desde el principio.

El procedimiento detallado es el siguiente:

1. Genere $Z = (R_1 - 1) \cdot 7 + R_2$, donde R_1 y R_2 son llamadas a rand7().
2. Si Z está entre 1 y 40 (ambos inclusive), entonces Z es un valor que nos sirve.
3. Si Z está entre 41 y 49, descarte Z y vuelva al paso 1.

Llamemos a $Z_{aceptado}$ la variable Z una vez que ha sido aceptada (es decir, $Z \leq 40$). La probabilidad de que un Z inicial sea aceptado es $P(Z \leq 40) = \frac{40}{49}$. Para cualquier valor $k \in \{1, \dots, 40\}$, la probabilidad condicional de que $Z_{aceptado}$ sea k, dado que ha sido aceptado, es:

$$P(Z_{aceptado} = k) = P(Z = k | Z \leq 40) = \frac{P(Z = k \cap Z \leq 40)}{P(Z \leq 40)} = \frac{1/49}{40/49} = \frac{1}{40}$$

Esto nos demuestra que, una vez que logramos un valor de Z que es aceptado, este valor está uniformemente distribuido en el rango $[1, 40]$. ¡Perfecto!

4 Mapeo al Rango Deseado $[1, 10]$

Ahora que tenemos un $Z_{aceptado}$ que se distribuye uniformemente en $[1, 40]$, el último paso es transformarlo al rango $[1, 10]$ utilizando una operación modular. La fórmula que necesitamos es:

$$Y = ((Z_{aceptado} - 1) \pmod{10}) + 1$$

Veamos por qué esta transformación nos da una distribución uniforme en $[1, 10]$: El rango $[1, 40]$ puede dividirse en exactamente 4 bloques completos de números del 1 al 10:

- $\{1, 2, \dots, 10\}$
- $\{11, 12, \dots, 20\}$
- $\{21, 22, \dots, 30\}$
- $\{31, 32, \dots, 40\}$

Para cada número $m \in \{1, \dots, 10\}$, existen exactamente 4 valores de $Z_{aceptado}$ que, al aplicar la fórmula, se mapean a m. Por ejemplo:

- Si queremos el 1: los valores de $Z_{aceptado}$ que nos lo darán son $1, 11, 21, 31$.
- Si queremos el 10: los valores de $Z_{aceptado}$ que nos lo darán son $10, 20, 30, 40$.

Dado que cada uno de estos 4 valores de $Z_{aceptado}$ tiene una probabilidad de $\frac{1}{40}$ de ser elegido (una vez que ha pasado el filtro de rechazo), la probabilidad de que nuestra función Y nos dé un número $m \in \{1, \dots, 10\}$ es la suma de las probabilidades de esos 4 valores:

$$P(Y = m) = 4 \cdot \frac{1}{40} = \frac{1}{10}$$

Esta probabilidad es, como queríamos, constante para todos los números del 1 al 10, lo que confirma que nuestra función Y está uniformemente distribuida en el rango deseado.

5 Implementación en C++

Aquí tiene la implementación de la función rand10() en C++, siguiendo rigurosamente el algoritmo que acabamos de describir:

```
// La API rand7() está predefinida para usted.
// int rand7();
// @return un entero aleatorio en el rango 1 a 7.

class Solution {
public:
    int rand10() {
```

```
        // Usaremos un bucle para el muestreo por rechazo.
        // Seguiremos intentando hasta que generemos un número en el rango [1, 40].
        using namespace std;
        int val;
        while (true) {
            // Generamos un número en el rango [1, 49] de forma uniforme.
            // (rand7() - 1) produce un número en [0, 6].
            // Multiplicarlo por 7 lo escala a {0, 7, 14, ..., 42}.
            // Sumarle otro rand7() (que está en [1, 7]) nos da un resultado en [1, 49].
            val = (rand7() - 1) * 7 + rand7();

            // Si 'val' cae en el rango aceptable [1, 40], lo usamos.
            if (val <= 40) {
                // Ahora, mapeamos este valor de [1, 40] a [1, 10] usando el módulo.
                // (val - 1) lo convierte a un rango [0, 39].
                // (val - 1) % 10 lo lleva a [0, 9].
                // Finalmente, sumamos 1 para obtener el rango deseado [1, 10].
                return (val - 1) % 10 + 1;
            }
            // Si 'val' está en [41, 49], lo rechazamos y el bucle se repetirá automáticamente.
        }
    }
};
```

—

6 Generalización

Podemos llevar este ingenioso método un paso más allá y generalizarlo para construir una función randN(). Esta nueva función generaría un entero aleatorio uniformemente distribuido en el rango $[1, N]$, basándose en una función existente randM() que ya nos da un entero aleatorio uniforme en el rango $[1, M]$.

6.1 Principio General

La meta es transformar la distribución de randM() para que se ajuste perfectamente a una distribución uniforme en el rango $[1, N]$. El proceso sigue los mismos dos pilares que ya hemos explorado:

1. **Expansión del Rango:** Tomar múltiples resultados de randM() y combinarlos para generar un entero aleatorio Z uniformemente distribuido en un rango intermedio $[1, R_{total}]$. Este rango R_{total} debe ser lo suficientemente amplio, al menos N, y si es posible, un múltiplo de N, o un valor que nos permita extraer un múltiplo de N.

2. **Muestreo por Rechazo y Mapeo:** Aplicar el muestreo por rechazo para limitar nuestro Z a un subrango $[1, R_{util}]$. Este R_{util} será el múltiplo de N más grande que sea menor o igual a R_{total}. Finalmente, el $Z_{aceptado}$ se mapea al rango deseado $[1, N]$ utilizando la operación módulo, tal como hicimos antes.

6.2 Construcción del Rango Intermedio R_{total}

Para construir un rango suficientemente amplio, la clave es combinar las llamadas a randM(). Si realizamos k llamadas a randM() (digamos $R_1, R_2, \dots, R_k$), podemos formar un número en base M:

$$Z = (R_1 - 1) \cdot M^{k-1} + (R_2 - 1) \cdot M^{k-2} + \cdots + (R_k - 1) \cdot M^0 + 1$$

Este Z estará uniformemente distribuido en el rango $[1, M^k]$. Cada valor $x \in \{1, \dots, M^k\}$ tendrá una probabilidad $P(Z = x) = \frac{1}{M^k}$. Necesitamos elegir el menor k que garantice $M^k \geq N$. Para una eficiencia óptima, el M^k idealmente debería ser lo más cercano posible a un múltiplo de N. Un enfoque práctico, como vimos con rand7() $\rightarrow$ rand10(), es simplemente multiplicar los rangos. Si combinamos

$C_1 = \text{randM}()$ y $C_2 = \text{randM}()$, podemos generar un número $Z = (C_1 - 1) \cdot M + C_2$, que se distribuye uniformemente en $[1, M^2]$. Podemos seguir este patrón, acumulando: $Z_{nuevo} = (Z_{anterior} - 1) \cdot M + \text{randM}()$, hasta que el rango total R_{total} sea igual o mayor que N.

En términos formales, los valores que nos da randM() están en $\{1, \ldots, M\}$. Para obtener un número en $[1, N]$, necesitamos un espacio de resultados con al menos N elementos, y que todos sean equiprobables. Sea k el entero positivo más pequeño tal que $M^k \geq N$. Definimos Z como el número $X_k + 1$, donde $X_k = \sum_{i=1}^{k} (\text{randM}_i() - 1) M^{k-i}$. Este Z estará uniformemente distribuido en $[1, M^k]$. Llamaremos a este M^k como R_{total}.

6.3 Muestreo por Rechazo y Mapeo

Una vez que tenemos nuestro Z uniformemente distribuido en $[1, R_{total}]$ (donde $R_{total} = M^k$), el siguiente paso es aplicar el muestreo por rechazo. Calculamos el límite superior $L = N \cdot \lfloor R_{total}/N \rfloor$. Este L es el múltiplo de N más grande que no excede R_{total}. El algoritmo para nuestra función randN() es el siguiente:

1. **Generar Z:** Entramos en un bucle donde generamos un valor Z utilizando repetidamente randM() hasta que Z caiga en el rango $[1, L]$.
   - Una manera de generar Z eficientemente es iniciar con $Z = \text{randM}()$ y $R_{total} = M$. Luego, en cada iteración, si R_{total} aún no es lo suficientemente grande o si Z supera L: $Z = (Z - 1) \cdot M + \text{randM}()$. $R_{total} = R_{total} \cdot M$.
2. **Mapear $Z_{aceptado}$:** Una vez que hemos obtenido un valor $Z_{aceptado}$ que se distribuye uniformemente en $[1, L]$, lo transformamos al rango $[1, N]$ con la fórmula:

$$Y = ((Z_{aceptado} - 1) \pmod N) + 1$$

La probabilidad de que cualquier número $m \in \{1, \ldots, N\}$ sea generado es $\frac{1}{N}$. Esto se debe a que cada m se corresponde con $\lfloor R_{total}/N \rfloor$ valores dentro del rango $[1, L]$, y todos esos valores son equiprobables.

6.4 Eficiencia

La eficiencia de este método depende directamente de la cantidad de valores que tengamos que rechazar. Cuanto más cerca esté la relación $\frac{L}{R_{total}}$ de 1, menos rechazos tendremos. La probabilidad de que un valor sea aceptado es $\frac{L}{R_{total}} = \frac{N \cdot \lfloor M^k/N \rfloor}{M^k}$. Esta probabilidad es óptima (cercana a 1) si M^k es un múltiplo exacto de N. Para minimizar las llamadas a randM() y los rechazos, debemos elegir k de forma inteligente, buscando que M^k sea lo más cercano posible a un múltiplo de N (aunque no siempre se pueda) y, por supuesto, que sea mayor o igual a N.

6.5 Ejemplo Generalizado en C++: randM() → randN()

Aquí se presenta una implementación generalizada en C++ para la función randN() utilizando randM():

```
// Supongamos que randM() está definido y devuelve un int en [1, M]
// int randM();
// const int M = ...; // Este sería el valor fijo de la función fuente (ej. 7 para rand7())

class Solution {
public:
    // Esta función nos permite generar un número en el rango [1, N]
    // usando una función randM() preexistente que genera en [1, M].
    // int randM(); // Asumimos que esta función está accesible globalmente o como miembro.

    int randN_func(int N) {
        int generated_val;
        int current_max_range;

        while (true) {
```

```
            generated_val = randM();
            current_max_range = M;

            // Continuamos expandiendo el rango de 'generated_val'
            // hasta que 'current_max_range' sea lo suficientemente grande.
            // Necesitamos que 'current_max_range' sea al menos N,
            // y preferiblemente que el límite de rechazo sea grande.
            // Multiplicamos por M hasta que el 'current_max_range'
            // sea mayor o igual que 'N' y 'N' no sea cero.
            while (current_max_range < N || (current_max_range % N != 0 && (current_max_range / N) *
                // Para evitar un bucle infinito si N es 0 o negativo, aunque el problema asume N >
                if (N == 0) return 0; // O lanzar una excepción, según el contexto.
                generated_val = (generated_val - 1) * M + randM();
                current_max_range *= M;
            }

            int limit_for_rejection = (current_max_range / N) * N;
            if (generated_val <= limit_for_rejection) {
                return (generated_val - 1) % N + 1;
            }
        }
    }

    // NOTA: Para que el código compile, necesitarías definir randM()
};
```