伪随机数生成器

GitHub Copilot

2024 年 5 月 8 日

摘要

这篇文章详细介绍了伪随机数以及伪随机数生成器的概念。我们将探讨各种伪随机数生成算法，包括线性同余生成器、梅森旋转、Xorshift 生成器、维纳尔-克特、PCG 和中间平方法。对于每种算法，我们都将介绍其原理，可能的 C/C++ 实现，以及生成的随机数的质量。最后，我们将讨论如何评估伪随机数的质量，包括统计测试、周期长度和均匀分布。

1 伪随机数简介

1.1 什么是伪随机数

伪随机数是由确定性算法生成的数字序列，尽管这些数字看起来是随机的，但实际上它们是由初始的” 种子” 值完全决定的。这意味着如果你知道生成伪随机数的算法和种子值，你就可以预测所有的随机数。这与真随机数不同，真随机数是由物理过程（如放射性衰变）生成的，无法预测 [2]。

伪随机数在许多领域都有应用，包括但不限于计算机图形学、密码学、和统计抽样。尽管它们不是真正的随机数，但在许多情况下，伪随机数的随机性已经足够好，可以用于各种需要随机性的应用 [3]。

伪随机数生成器（PRNG）是一种算法，它可以生成看起来像随机数的长序列。然而，由于 PRNG 是完全由算法驱动的，所以它们实际上是确定的和可重复的。这意味着如果你知道 PRNG 的内部状态，你就可以预测它将生成的所有未来的数字。

PRNG 的一个关键特性是它们的周期性。周期是 PRNG 在重复其输出序列之前可以生成的唯一数字的数量。理想的 PRNG 将有一个非常长的周期，这样在实际应用中就不太可能看到输出序列的重复。

尽管伪随机数不能用于所有需要随机性的应用，但它们在许多情况下是非常有用的。例如，在模拟和建模、游戏、艺术、和密码学中，伪随机数通常是一个很好的选择，因为它们可以快速生成，且具有可预测和可重复的特性。

1.2 伪随机数的应用

伪随机数在许多领域都有广泛的应用。以下是一些主要的例子：

- **计算机图形学**：在计算机图形学中，伪随机数被用于生成纹理和噪声，以增加视觉效果的复杂性和真实感 [5]。
- **密码学**：在密码学中，伪随机数被用于生成密钥和初始化向量。这些数必须具有高度的不可预测性，以防止攻击者猜测它们 [4]。
- **统计抽样**：在统计抽样中，伪随机数被用于选择样本，以确保样本的代表性 [6]。
- **模拟和建模**：在模拟和建模中，伪随机数被用于模拟随机事件，如天气模式、股票价格和物理实验 [6]。

1.3 伪随机数与真随机数的区别

伪随机数和真随机数的主要区别在于它们的生成方式和预测性。

伪随机数是由确定性算法生成的，这意味着给定相同的初始种子值，算法将生成相同的数字序列。因此，伪随机数是可预测的，如果你知道算法和种子值，你就可以预测所有的随机数 [2]。

真随机数，另一方面，是由非确定性的物理过程生成的，如放射性衰变或大气噪声。这些过程是不可预测的，因此生成的随机数也是不可预测的。真随机数生成器（TRNG）通常需要专门的硬件，并且生成速度较慢 [1]。

尽管伪随机数不是真正的随机数，但在许多情况下，它们的随机性已经足够好，可以用于各种需要随机性的应用。然而，在需要高度不可预测性的应用中，如密码学，真随机数可能是更好的选择 [4]。

1.4 线性同余生成器（LCG）

线性同余生成器（LCG）是一种常见的伪随机数生成算法。它的工作原理基于线性同余方程，这是一种简单但有效的方法来生成伪随机数序列。

1.4.1 算法原理

LCG 的工作原理可以用以下的数学公式来描述：

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

其中，X_n 是当前的数值，X_{n+1} 是下一个数值，a、c 和 m 是常数。这些常数需要被仔细选择，以确保生成的数字序列具有良好的随机性和长的周期。

LCG 的优点是它的实现简单，运行速度快。然而，它也有一些缺点。例如，如果常数 a、c 和 m 没有被正确选择，那么生成的数字序列可能会有明显的模式。此外，LCG 生成的数字序列在高维空间中的分布可能会有问题。

1.4.2 生成的随机数质量

LCG 生成的随机数质量取决于常数 a、c 和 m 的选择。如果这些常数被正确选择，那么 LCG 可以生成具有良好随机性和长周期的数字序列。

然而，LCG 也有一些已知的问题。例如，它生成的数字序列在高维空间中的分布可能会有问题。这是因为 LCG 生成的数字序列在高维空间中可能会形成超平面，这可能会影响到需要高维随机数的应用。

总的来说，尽管 LCG 有一些已知的问题，但由于其简单和高效，它仍然是一种常用的伪随机数生成算法。

1.5 梅森旋转（Mersenne Twister）

梅森旋转是一种常用的伪随机数生成算法。它的工作原理基于有限二进制字段上的线性递归，可以生成具有极长周期的数字序列。

1.5.1 算法原理

梅森旋转的工作原理可以用以下的步骤来描述：

1. 初始化一个长度为624的状态向量。
2. 通过线性递归关系生成下一个状态向量。
3. 通过混淆函数将状态向量转换为一个32位的输出。
4. 重复步骤2和步骤3，生成更多的随机数。

1.5.2 C/C++ 实现

这个过程可以用以下的 C++ 代码来实现:

```
#include <iostream>
#include <fstream>

#define N 624                    // 状态向量的长度
#define M 397                    // 用于生成下一个
   状态向量的位移量
#define MATRIX_A 0x9908b0dfUL    // 用于生成下一个
   状态向量的常数
#define UPPER_MASK 0x80000000UL // 用于取状态向量
   的上半部分
#define LOWER_MASK 0x7fffffffUL // 用于取状态向量
   的下半部分

static unsigned long mt[N]; // 状态向量
static int mti = N + 1;     // 状态向量的索引

// 初始化状态向量
void init_genrand(unsigned long s) {
    mt[0] = s & 0xffffffffUL;
    for (mti = 1; mti < N; mti++) {
        mt[mti] = (1812433253UL * (mt[mti - 1] ^ (
           mt[mti - 1] >> 30)) + mti);
        mt[mti] &= 0xffffffffUL;
    }
}

// 生成一个32位的随机数
unsigned long genrand_int32(void) {
    unsigned long y;
    static unsigned long mag01[2] = {0x0UL,
       MATRIX_A};
```

```
        if (mti >= N) { // 如果状态向量已经用完，就生
            成下一个状态向量
            int kk;
            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 & 0x1UL];
            }
            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 & 0x1UL];
            }
            y = (mt[N - 1] & UPPER_MASK) | (mt[0] &
                LOWER_MASK);
            mt[N - 1] = mt[M - 1] ^ (y >> 1) ^ mag01[y
                 & 0x1UL];
            mti = 0;
        }
        y = mt[mti++];
        y ^= (y >> 11);                 // 混淆函数
        y ^= (y << 7) & 0x9d2c5680UL;  // 混淆函数
        y ^= (y << 15) & 0xefc60000UL; // 混淆函数
        y ^= (y >> 18);                 // 混淆函数
        return y;
    }

    // 生成一个64位的随机数
    unsigned long long genrand_int64(void) {
        return ((unsigned long long)genrand_int32() <<
            32) | genrand_int32();
```

```
}
```

1.5.3 生成的随机数质量

梅森旋转生成的随机数质量通常非常好。它可以生成具有极长周期（$2^{19937}-1$）的数字序列，且通过了许多统计测试，包括 DIEHARD 和 TestU01。然而，它并不适合用于密码学，因为观察到足够多的输出后，可以预测其后续的输出。

1.6 Xorshift 生成器

Xorshift 生成器是一种基于位运算的伪随机数生成算法。它的工作原理简单但效果却非常好，能够生成具有良好随机性和长周期的数字序列。

1.6.1 算法原理

Xorshift 生成器的工作原理可以用以下的步骤来描述：

1. 将当前的数值向左（或向右）进行位移运算。2. 将结果与原始的数值进行异或运算。3. 将结果向右（或向左）进行位移运算。4. 将结果与原始的数值进行异或运算，得到下一个数值。

这个过程可以用以下的C++代码来实现：

```
uint32_t xorshift32(uint32_t state) {
    // Xorshift algorithm from George
       Marsaglia's paper
    state ^= state << 13;
    state ^= state >> 17;
    state ^= state << 5;
    return state;
}
```

这段代码首先将当前的数值向左移动 13 位，然后将结果与原始的数值进行异或运算。接着，它将结果向右移动 17 位，然后将结果与原始的数值进行异或运算。最后，它将结果向左移动 5 位，然后将结果与原始的数值进行异或运算，得到下一个数值。

1.6.2 生成的随机数质量

Xorshift 生成器生成的随机数质量通常非常好。它可以生成具有良好随机性和长周期的数字序列。然而，它也有一些已知的问题。例如，它生成的数字序列的最低位的随机性可能会比较差。这是因为在位移运算中，最低位的信息可能会被丢失。

总的来说，尽管 Xorshift 生成器有一些已知的问题，但由于其简单和高效，它仍然是一种常用的伪随机数生成算法。

1.7 维纳尔-克特（WELL）

维纳尔-克特（WELL）生成器是一种改进的伪随机数生成算法。它的工作原理基于线性反馈移位寄存器（LFSR），并引入了额外的非线性操作来提高生成的数字序列的随机性。

1.7.1 算法原理

WELL 生成器的工作原理可以用以下的步骤来描述：

1. 从当前的状态向量中选择几个元素。
2. 将这些元素进行位移和异或运算，得到一个新的元素。
3. 将新的元素插入到状态向量的开始位置，同时移除状态向量的最后一个元素。
4. 将状态向量中的所有元素进行一次非线性转换，得到下一个状态向量。

这个过程可以用以下的 C++ 代码来实现：

```
uint32_t WELLRNG512a(void) {
    uint32_t a, b, c, d;
    a = STATE[state_i];
    c = STATE[(state_i+13)&15];
    b = a^c^(a<<16)^(c<<15);
    c = STATE[(state_i+9)&15];
    c ^= (c>>11);
    a = STATE[state_i] = b^c;
```

```
        d = a^((a<<5)&0xDA442D24UL);
        STATE[state_i] = a = d^b^((d<<2)^(b<<18))
            ^(c<<28);
        state_i = (state_i + 15)&15;
        return a;
    }
```

1.7.2 生成的随机数质量

WELL 生成器生成的随机数质量通常非常好。它可以生成具有良好随机性和长周期的数字序列。然而，它也有一些已知的问题。例如，它生成的数字序列的最低位的随机性可能会比较差。这是因为在位移运算中，最低位的信息可能会被丢失。

总的来说，尽管 WELL 生成器有一些已知的问题，但由于其改进了 LFSR 的随机性，它仍然是一种常用的伪随机数生成算法。

1.8 PCG (Permuted Congruential Generator)

PCG 是一种新型的伪随机数生成算法。它的工作原理基于线性同余生成器（LCG），并引入了额外的排列操作来提高生成的数字序列的随机性。

1.8.1 算法原理

PCG 的工作原理可以用以下的步骤来描述：

1. 使用LCG生成一个新的数值。
2. 将这个数值进行位移和异或运算，得到一个新的数值。
3. 返回这个新的数值作为下一个随机数。

这个过程可以用以下的 C++ 代码来实现：

```
struct pcg32_random_t {  // Internals are *Private
    *.
     uint64_t state;  // RNG state.  All values are
         possible.
```

```
    uint64_t inc;    // Controls which RNG
       sequence (stream) is selected. Must *always
       * be odd.
};

uint32_t pcg32_random_r(pcg32_random_t* rng) {
    uint64_t oldstate = rng->state;
    // Advance internal state
    rng->state = oldstate * 6364136223846793005ULL
        + (rng->inc|1);
    // Calculate output function (XSH RR), uses
       old state for max ILP
    uint32_t xorshifted = ((oldstate >> 18u) ^
       oldstate) >> 27u;
    uint32_t rot = oldstate >> 59u;
    return (xorshifted >> rot) | (xorshifted <<
       ((-rot) & 31));
}
```

1.8.2 生成的随机数质量

PCG 生成的随机数质量通常非常好。它可以生成具有良好随机性和长周期的数字序列。然而，它也有一些已知的问题。例如，它生成的数字序列的最低位的随机性可能会比较差。这是因为在位移运算中，最低位的信息可能会被丢失。

总的来说，尽管 PCG 有一些已知的问题，但由于其改进了 LCG 的随机性，它仍然是一种常用的伪随机数生成算法。

1.9 中间平方法

中间平方法是一种早期的伪随机数生成算法。它的工作原理基于数字的平方运算，通过取平方数的中间部分作为下一个随机数。

1.9.1 算法原理

中间平方法的工作原理可以用以下的步骤来描述：

1. 取一个初始的数值。
2. 将这个数值进行平方运算。
3. 取平方数的中间部分作为下一个随机数。
4. 重复步骤2和步骤3，生成更多的随机数。

这个过程可以用以下的 Python 代码来实现：

```
def middle_square_method(seed, digit=4):
    square = str(seed ** 2)
    while len(square) < 2 * digit:  # pad with
       zeros if necessary
        square = '0' + square
    next_seed = int(square[len(square) // 4: 3 *
       len(square) // 4])
    return next_seed
```

1.9.2 生成的随机数质量

中间平方法生成的随机数质量通常较差。它生成的数字序列的周期通常很短，而且可能会陷入循环。此外，它还有一些已知的问题，例如，如果初始数值的平方是一个全零的数，那么它将永远只能生成零。

总的来说，尽管中间平方法在早期的计算机系统中得到了一些应用，但由于其生成的随机数质量较差，现在已经很少使用了。

2 如何评估伪随机数的质量

评估伪随机数的质量通常涉及到几个关键的方面：统计测试、周期长度和分布均匀性。以下是对这些评估方法的详细介绍：

2.1 统计测试

统计测试是评估伪随机数质量的主要方法之一。这些测试检查生成的数字序列是否具有随机数应具有的统计特性。例如，一个好的伪随机数生成

器生成的数字序列应该没有明显的模式，数字的出现频率应该大致相等，且数字之间应该没有明显的相关性 [2]。

有许多不同的统计测试可以用来评估伪随机数的质量，包括频率测试、序列测试、游程测试、熵测试等。这些测试可以帮助我们检测生成的数字序列中的模式和偏差，从而评估伪随机数生成器的质量。

2.2 周期长度

伪随机数生成器的周期长度是指生成器在重复其输出序列之前可以生成的唯一数字的数量。理想的伪随机数生成器应该有一个非常长的周期，这样在实际应用中就不太可能看到输出序列的重复 [2]。

周期长度是评估伪随机数生成器质量的重要指标。如果生成器的周期太短，那么生成的数字序列可能会在短时间内重复，这可能会影响到需要随机性的应用。

2.3 均匀分布

伪随机数应该均匀分布在其输出范围内。这意味着在长期运行中，每个可能的输出值都应该出现大致相同的次数 [2]。

一种直观的评估伪随机数分布均匀性的方法是生成一个大的数字序列，然后将这些数字映射到一个二维图像上。例如，我们可以使用 C++ 生成一系列的伪随机数，然后将这些数值映射到一个 BMP 图像的像素上。如果生成的图像看起来像是随机噪声，那么我们可以认为伪随机数生成器的输出是均匀分布的。如果图像中出现明显的模式或者结构，那么这可能表明生成器的输出并不均匀分布。

我们可以使用以下的 C++ 代码来评估梅森旋转算法生成的随机数的质量。这段代码将生成一系列的随机数，并将这些数值映射到一个 BMP 图像的像素上，以此来评估生成的随机数是否均匀分布。

```
// BMP图像的宽度和高度
#define WIDTH 512
#define HEIGHT 512

// 用于生成BMP图像的RGB颜色结构体
struct RGB {
```

```
    unsigned char r;
    unsigned char g;
    unsigned char b;
};

// 初始化梅森旋转算法
void init_mersenne_twister(unsigned long seed) {
    init_genrand(seed);
}

// 生成一个随机的RGB颜色
RGB generate_random_color() {
    RGB color;
    unsigned long long rand_value = genrand_int64();
    color.r = rand_value % 256;
    rand_value >>= 8;
    color.g = rand_value % 256;
    rand_value >>= 8;
    color.b = rand_value % 256;
    return color;
}

// 生成一个BMP图像
void generate_bmp(const std::string& filename) {
    std::ofstream file(filename, std::ios::binary);

    // BMP文件头
    unsigned char fileHeader[14] = {
        'B', 'M',  // 魔数
        0, 0, 0, 0,  // 文件大小
        0, 0,  // 保留字段
        0, 0,  // 保留字段
        54, 0, 0, 0  // 偏移量到像素数据
```

```
};

// BMP信息头
unsigned char infoHeader[40] = {
    40, 0, 0, 0,  // 信息头大小
    0, 0, 0, 0,  // 图像宽度
    0, 0, 0, 0,  // 图像高度
    1, 0,  // 颜色平面数
    24, 0,  // 每像素位数
    0, 0, 0, 0,  // 压缩类型
    0, 0, 0, 0,  // 图像大小
    0, 0, 0, 0,  // 水平分辨率
    0, 0, 0, 0,  // 垂直分辨率
    0, 0, 0, 0,  // 颜色表中的颜色数
    0, 0, 0, 0,  // 重要颜色数
};

// 设置图像的宽度和高度
*(int*)&infoHeader[4] = WIDTH;
*(int*)&infoHeader[8] = -HEIGHT;  // 负的高度表示
    像素顺序从上到下

// 写入文件头和信息头
file.write((char*)fileHeader, 14);
file.write((char*)infoHeader, 40);

// 生成并写入像素数据
for (int y = 0; y < HEIGHT; ++y) {
    for (int x = 0; x < WIDTH; ++x) {
        RGB color = generate_random_color();
        file.write((char*)&color, 3);
    }
}
```

```
    file.close();
}

int main() {
    init_mersenne_twister(123456789);  // 使用一个种子初始化梅森旋转算法
    generate_bmp("random32.bmp");  // 生成32位随机颜色的BMP图像
    generate_bmp("random64.bmp");  // 生成64位随机颜色的BMP图像
    return 0;
}
```

这段代码首先初始化梅森旋转算法，然后生成一个 512x512 的 BMP 图像，每个像素的颜色由梅森旋转算法生成的随机数决定。生成的图像保存为”random.bmp” 文件。

通过观察生成的图像，我们可以直观地评估梅森旋转算法生成的随机数的分布均匀性。如果图像看起来像是随机噪声，那么我们可以认为梅森旋转算法的输出是均匀分布的。如果图像中出现明显的模式或者结构，那么这可能表明梅森旋转算法的输出并不均匀分布。

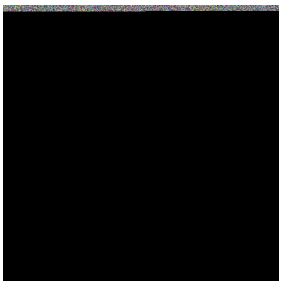

图 1: 中间平方法生成的随机数图像

从图 1、图 2 和图 3 可以看出，梅森旋转算法生成的随机数图像比中间平方法生成的随机数图像更接近于随机噪声，这表明梅森旋转算法生成的随机数的分布更均匀，因此，梅森旋转算法比中间平方法更优秀。

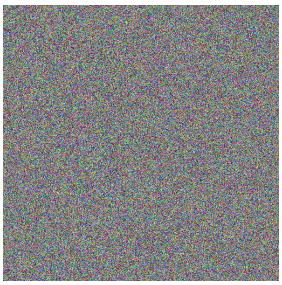

图 2: 梅森旋转算法生成的 32 位随机数图像

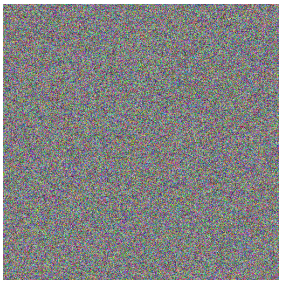

图 3: 梅森旋转算法生成的 64 位随机数图像

3 总结

伪随机数在许多领域都有着广泛的应用，包括但不限于模拟、加密、计算机图形学等。因此，理解并能够评估伪随机数的质量是非常重要的。

我们介绍了几种评估伪随机数质量的方法，包括统计测试、周期长度和分布均匀性。统计测试可以帮助我们检测生成的数字序列中的模式和偏差，周期长度可以帮助我们理解生成器在重复其输出序列之前可以生成的唯一数字的数量，而分布均匀性则可以帮助我们理解生成的数字是否均匀分布在其输出范围内。

我们还展示了如何使用 C++ 生成一系列的伪随机数，并将这些数值映射到一个 BMP 图像的像素上，以直观地评估伪随机数的分布均匀性。这种方法可以帮助我们直观地理解生成的数字序列的分布情况，从而评估伪随机数生成器的质量。

总的来说，评估伪随机数的质量是一个复杂但重要的任务。通过理解和应用上述方法，我们可以更好地评估和选择伪随机数生成器，以满足我们的特定需求。

参考文献

[1] Andreas Holzinger, Georg Langs, Helmut Denk, Kurt Zatloukal, and Heimo Müller. Machine learning for health informatics: state-of-the-art and future challenges. In *Machine Learning for Health Informatics*, pages 271–293. Springer, 2018.

[2] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, 1981.

[3] Pierre L'Ecuyer. Uniform random number generation. *Annals of Operations Research*, 86:213–233, 1994.

[4] Alfred J Menezes, Paul C van Oorschot, and Scott A Vanstone. *Handbook of Applied Cryptography*. CRC press, 2018.

[5] Art B Owen. A survey of quasi-random point sequences. *ACM SIGSAM Bulletin*, 37(2):20–29, 2003.

[6] Christian P Robert and George Casella. *Monte Carlo methods*. Springer, 2004.