

STAT243 Lecture 9.4 Random Number Generation

Logic ▾

仿真研究的核心，是生成随机数与随机变量的能力。

在计算机中，所有随机数实际上都是伪随机数 (pseudo-random numbers) —— 它们由确定性算法生成，但能“看起来”像真正的随机数，并且可以复现。

复现性对于科研而言至关重要，因为它保证了仿真结果的可重复性。

1 Generating Random Uniforms on a Computer

1.1 基本思想 (Fundamental Idea)

所有随机变量的生成都可以追溯到标准均匀分布 $U(0, 1)$ 。

我们只需能生成独立均匀随机数，就能通过变换生成各种分布的随机变量。

在计算机上：

- 随机数由伪随机数生成器 (RNG) 产生；
- 其输出为有限的、离散的数值序列；
- 每个 RNG 由状态 (state) 与输出 (output) 组成；
- 种子 (seed) 决定 RNG 序列的起始位置。

1.2 Sequential Congruential Generators (序列同余生成器)

一种常见的 RNG 类型为序列同余法：

$$x_k = f(x_{k-1}, \dots, x_{k-j}) \mod m$$

通常 $j = 1$ ，其中：

- m ：模数 (modulus)
- f ：递推函数
- 输出随机数 $u_k = x_k/m \in [0, 1)$

1.2.1 一个例子：线性同余法 (Linear Congruential Generator, LCG)

最常见形式：

$$x_k = (ax_{k-1} + c) \mod m$$

其中 a, c, m 为整数，且种子为 x_0 。

性质：

- 序列是有限循环的（周期性）；
- 若 $c = 0$ ，最大周期为 $m - 1$ ；
- 典型模数选取为 Mersenne 素数 $2^p - 1$ 。

Python 示例：

```

Python
1  n = 100
2  a = 171
3  m = 30269
4  x = np.empty(n)
5  x[0] = 7306
6  for i in range(1, n):
7      x[i] = (a * x[i-1]) % m
8  u = x / m

```

1.2.2 RNG 性能评估

一个“好”的 RNG 应满足：

1. 单个样本均匀分布；
2. 各样本独立；
3. 多维情况下呈均匀性（在高维超立方体中）。

然而，LCG 常在高维空间中呈**格状分布 (lattice structure)**，即点落在若干平行超平面上，而非均匀散布在整个空间中。

1.3 Wichmann–Hill 组合生成器

Logic ▾

为改进 LCG 的统计性质，可**组合多个生成器**。

Wichmann–Hill 方法结合了三组参数：

$$a = 171, 172, 170, \quad m = 30269, 30307, 30323$$

生成：

$$u_i = (x_i/30269 + y_i/30307 + z_i/30323) \mod 1$$

R 示例（手动实现）：

```

R
1  RNGkind("Wichmann-Hill")
2  set.seed(1)
3  a <- c(171, 172, 170)
4  m <- c(30269, 30307, 30323)
5  for (i in 2:10) {
6      xyz[i, ] <- (a * xyz[i-1, ]) %% m
7      u[i] <- sum(xyz[i, ]/m) %% 1
8  }

```

该算法提升了多维独立性，是早期 R 默认的 RNG。

1.4 PCG Generators (Permutation Congruential Generators)

由 **Melissa O'Neal (2014)** 提出，是改进型 LCG，现为 **NumPy 默认随机数生成器 (PCG-64)**。

1.4.1 核心思想

1. 使用 **极大模数** $m = 2^k$ (如 64 或 128 位)；
2. 对 LCG 的 states 进行**位操作置换 (permutation)**；
3. 输出经过旋转或位移的结果，提升统计性能。

1.4.2 特性

- 状态长度：64 或 128 bit；
- 周期长度： 2^{128} ；
- 多流机制：支持 2^{127} 个独立随机流；
- 当前 NumPy 默认实现：`numpy.random.default_rng()`。

1.5 Mersenne Twister Generator

Mersenne Twister (MT19937) 是最流行的 RNG：

- R 的默认 RNG；
- NumPy 旧版 (`np.random`) 默认 RNG。

1.5.1 特点

- 周期： $2^{19937} - 1 \approx 10^{6000}$ ；
- 状态由 624 个 32-bit 整数组成；
- 属于 **广义反馈移位寄存器 (GFSR)** 类；
- 通过 **异或操作 (exclusive-or)** 生成伪随机位序列。

1.5.2 实现与可访问性

在 NumPy 中通过：

```
Python
1 np.random.Generator(np.random.MT19937(seed=1))
```

或旧接口：

```
Python
1 np.random.seed(1)
```

1.6 Period vs. Unique Values

虽然 RNG 周期极长，但输出值仍有限：

- PCG-64 输出 64 位 → 最多 2^{64} 种唯一值；
- MT19937 输出 32 位 → 最多 2^{32} 种唯一值。

这意味着可能出现重复值，但不会出现重复状态序列，因此不影响周期特性。

1.7 The Seed and the State

- **Seed (种子)**: RNG 序列的起始位置;
- **State (状态)**: RNG 的内部存储, 包括寄存器、计数器等;
- **重现性**: 通过相同种子保证相同序列;
- **NumPy**: PCG-64 的状态包含两个 128-bit 整数 (状态与增量 c)。



Python

```
1 rng = np.random.default_rng(seed=1)
2 saved_state = rng.bit_generator.state
3 saved_state['state']['state'] # 状态
4 saved_state['state']['inc']   # 增量 c
```

2 RNG in Python

2.1 Choosing a Generator

NumPy 1.17+ 提供新接口:



Python

```
1 rng = np.random.default_rng(seed=1)           # 默认使用 PCG-64
2 rng = np.random.Generator(np.random.MT19937(seed=1)) # 指定 Mersenne Twister
3
4 rng = np.random.Generator(np.random.PCG64(seed = 1)) # PCG-64
```

- `np.random.default_rng()` → 新 API (推荐)
- `np.random` → 旧 API, 仍使用 Mersenne Twister

使用时必须通过 `Generator` 对象调用:



Python

```
1 rng.normal(size=3)           # 使用选定生成器
2 np.random.normal(size=3) # 使用旧接口, 不受影响
```

2.2 Using PCG-64



Python

```
1 rng = np.random.default_rng(seed=1)
2 rng.normal(size=5)
3 # array([ 0.34558419, 0.82161814, 0.33043708, -1.30315723, 0.90535587])
4
5 saved_state = rng.bit_generator.state
6 rng.normal(size=5)
7 # array([ 0.44637457, -0.53695324, 0.5811181 , 0.3645724 , 0.2941325 ])
8
```

```

9  rng.bit_generator.state = saved_state # 恢复状态
10 rng.normal(size=5)
11 # array([ 0.44637457, -0.53695324, 0.5811181 , 0.3645724 , 0.2941325 ])

```

PCG-64 的状态包含：

- `'state'`：主状态；
- `'inc'`：增量参数。



Python

```

1  saved_state
2  # {'bit_generator': 'PCG64', 'state': {'state': 216676376075457487203159048251690499413,
   # 'inc': 194290289479364712180083596243593368443}, 'has_uint32': 0, 'uinteger': 0}
3
4  saved_state['state']['state'] # actual state
5  # 216676376075457487203159048251690499413
6
7  saved_state['state']['inc'] # increment ('c')
8  # 194290289479364712180083596243593368443

```

2.3 Using Mersenne Twister (optional)

旧接口：



Python

```

1  np.random.seed(1)
2  np.random.normal(size=5)
3  # array([ 1.62434536, -0.61175641, -0.52817175, -1.07296862, 0.86540763])

```

保存并恢复状态：



Python

```

1  saved_state = np.random.get_state()
2  tmp = np.random.choice(np.arange(1, 51), size=2000, replace=True)
3  np.random.set_state(saved_state)
4  np.random.normal(size=5)
5  # array([-2.3015387 , 1.74481176, -0.7612069 , 0.3190391 , -0.24937038])

```

3 RNG in Parallel

3.1 并行随机数的潜在问题

并行环境中使用 RNG 时，应避免每个进程使用相同种子，否则所有进程生成相同序列。

错误示例：



Python

```

1  # 在所有进程中
2  np.random.seed(1)

```

会导致各进程得到完全相同的随机数流。

3.2 正确做法

- 为每个进程指定不同且独立的随机流；
- NumPy 提供多流机制支持（PCG64 的 2^{127} 独立流）；
- 可使用 SCF 并行教程中介绍的方法：
[Random Number Generation in Parallel \(Berkeley SCF\)](#)

4 Summary

| 生成器 | 类型 | 周期长度 | 状态大小 | NumPy 支持 | 特点 |
|------------------|---------|-----------------|---------------|---------------|----------|
| LCG | 线性同余 | 中等 | 单一整数 | 手动实现 | 简单但高维性差 |
| Wichmann–Hill | 组合型 | 长 | 三组整数 | R 可用 | 改善独立性 |
| PCG-64 | 置换同余 | 2^{128} | 两个 128-bit 整数 | NumPy 默认 | 高性能、支持并行 |
| Mersenne Twister | 反馈移位寄存器 | $2^{19937} - 1$ | 624×32-bit | NumPy 旧版、R 默认 | 稳定且快速 |