STAT243 Lecture 9 Simulation

1 回归示例动机 (Motivating Example)

考虑一个**线性回归模型**：

$$Y = X\beta + \varepsilon$$

其中：

- $Y = (y_1, y_2, \ldots, y_n)$ 为响应变量；
- X 为 $n \times p$ 的设计矩阵；
- $\varepsilon \sim N(0, \sigma^2 I)$。

标准最小二乘估计为：

$$\hat{\beta} = (X^\top X)^{-1} X^\top Y$$

根据经典假设（线性期望与独立同方差误差），可得：

$$\begin{aligned} E\hat{\beta} &= \beta \\ \mathrm{Var}(\hat{\beta}) = E((\hat{\beta} - E\hat{\beta})^2) &= \sigma^2 (X^\top X)^{-1} \\ \mathrm{MSPE}(Y^*) = E(Y^* - \hat{Y})^2) &= \sigma^2 (1 + X^{*\top} (X^\top X)^{-1} X^*). \end{aligned}$$

其中 Y^* 为新样本的预测值。

1.1 当模型假设不成立时

如果：

- 均值与 X 的关系**非线性**；
- 误差项**非独立或异方差**；
- 或者我们使用了**稳健估计方法**（例如抗离群点估计）——

则上述解析结果不再适用。
此时我们可使用 **Monte Carlo 模拟** 来研究估计量的性质。

1.2 Monte Carlo 回归模拟的基本思路

若生成 m 个独立样本集 $Y_1, \ldots, Y_m$ 来估计 β，并在第 i 个样本集上得到 $\hat{\beta}_i$，则：

- 估计期望：

$$\widehat{E[\hat{\beta}]} = \bar{\hat{\beta}} = \frac{1}{m} \sum_{i=1}^{m} \hat{\beta}_i$$

- 估计方差：

$$\widehat{\mathrm{Var}}(\hat{\beta}) = \frac{1}{m} \sum_{i=1}^{m} (\hat{\beta}_i - \bar{\hat{\beta}})^2$$

Monte Carlo 方法能帮助我们评估在“非标准假设”下的估计稳定性与偏差。

2 蒙特卡洛方法基础 (Monte Carlo Basics)

2.1 核心思想 (Overview)

我们通常希望估计某个期望：

$$\phi = E_f[h(Y)]$$

其中 $Y \sim f$，$h(Y)$ 为感兴趣的函数。

2.1.1 典型示例

若 $h(Y) = I(Y \leq y)$，则：

$$\phi = P(Y \leq y) = E_f[I(Y \leq y)]$$

表示累积分布函数 $F(y)$。

2.2 蒙特卡洛估计量 (Monte Carlo Estimator)

若从分布 f 独立采样 $Y_1, \ldots, Y_m$，则：

$$\hat{\phi} = \frac{1}{m} \sum_{i=1}^{m} h(Y_i)$$

根据大数定律：

$$\hat{\phi} \xrightarrow[m \to \infty]{} E_f[h(Y)] = \phi$$

在实际模拟中，Y_i 通常是一个完整的数据集，而非单个观测值。
若每个样本有 n 个观测，则 $Y_i = (Y_{i1}, \ldots, Y_{in})$。

2.3 回归问题中的对应关系 (Back to Regression Example)

若我们希望检验回归估计量是否有偏：

$$\phi = E[\hat{\beta}], \quad h(Y) = \hat{\beta}(Y)$$

则 Monte Carlo 估计为：

$$\hat{\phi} = \frac{1}{m} \sum_{i=1}^{m} \hat{\beta}_i$$

若要估计方差：

$$\phi = \mathrm{Var}(\hat{\beta}) = E[(\hat{\beta} - E\hat{\beta})^2]$$

则：

$$\hat{\phi} = \frac{1}{m} \sum_{i=1}^{m} (\hat{\beta}_i - \bar{\hat{\beta}})^2$$

2.4 置信区间覆盖率 (Confidence Interval Coverage)

我们可进一步评估置信区间的覆盖概率：

$$h(Y) = I[\beta \in CI(Y)]$$

Monte Carlo 估计：

$$\hat{\phi} = \frac{1}{m}\sum_{i=1}^{m} I[\beta \in CI(Y_i)]$$

理想情况下，$\hat{\phi} \approx 1-\alpha$（例如 0.95）。

覆盖率偏低的原因：

1. 估计的方差过小；
2. 估计量存在偏差。

---

3 模拟误差 (Simulation Uncertainty)

3.1 Simulation uncertainty

因为 $\hat{\phi}$ 是 $h(Y_i)$ 的样本均值，
其方差为：

$$\mathrm{Var}(\hat{\phi}) = \frac{\sigma^2}{m}, \quad \sigma^2 = \mathrm{Var}(h(Y))$$

样本估计为：

$$\hat{\sigma}^2 = \frac{1}{m-1}\sum_{i=1}^{m}(h(Y_i) - \hat{\phi})^2$$

从而：

$$\widehat{\mathrm{Var}}(\hat{\phi}) = \frac{\hat{\sigma}^2}{m} = \frac{1}{m(m-1)}\sum_{i=1}^{m}(h(Y_i) - \hat{\phi})^2$$

其阶为 $O(1/m)$，标准误差随 $1/\sqrt{m}$ 减小。

3.2 区分模拟误差与统计误差

> ⚠ **Remark**
>
> **Monte Carlo 不确定性 ≠ 统计不确定性**
>
> - Monte Carlo 不确定性：来自模拟随机数的误差
> - 统计不确定性：来自数据生成过程的不确定性

我们可以通过增大 m 来减少 Monte Carlo 方差，但其收敛速度为 $\propto 1/\sqrt{m}$。

3.3 回归示例中的模拟方差 (Regression Example)

可关心的 Monte Carlo 方差包括：

- $\widehat{\mathrm{Var}}(\hat{E}[\hat{\beta}] - \beta)$：偏差估计的不确定性；
- $\widehat{\mathrm{Var}}(\widehat{\mathrm{Var}}(\hat{\beta}))$：方差估计的不确定性；
- $\widehat{\mathrm{Var}}(\widehat{\mathrm{MSPE}}(Y^*))$：预测误差估计的不确定性。

在每种情况下，$\widehat{\mathrm{Var}}()$ 表示估计模拟方差。

---

4 方差缩减技巧 (Variance Reduction, Optional)

4.1 常见方法

- **重要性抽样 (Importance Sampling)**
- **对照变量 (Control Variates)**
- **反对称抽样 (Antithetic Sampling)**

后两种在实际中使用较少，但思想上可减少模拟方差。

4.2 分层模拟 (Stratified Simulation)

若能将总体划分为若干已知比例的分层，可在各层内分别估计均值 μ_h，再根据层权重加权组合。
这种做法能减少层间抽样带来的方差。

4.3 Rao-Blackwell化 (Rao-Blackwellization)

若希望计算：

$$E[h(X)], \quad X = (X_1, X_2)$$

根据迭代期望：

$$E[h(X)] = E[E(h(X) \mid X_2)]$$

若能解析计算 $E(h(X) \mid X_2)$，则无需为 X_1 额外引入随机性。
Rao-Blackwell 化的估计为：

$$\hat{\mu}_{RB} = \frac{1}{m} \sum_{i=1}^{m} E[h(X) \mid X_{2,i}]$$

其方差更小，因为：

$$\mathrm{Var}(E[h(X) \mid X_2]) < \mathrm{Var}(h(X))$$

这是由于：

$$\mathrm{Var}(X) = E[\mathrm{Var}(X|Y)] + \mathrm{Var}(E[X|Y])$$

因此，Rao-Blackwell化本质上利用了**条件期望降低方差**的思想。

---

5 模拟研究的基本步骤 (Basic Steps of a Simulation Study)

1. **定义单次实验 (Specify an Experiment)**
   明确：
   - 样本量 n
   - 数据分布 (e.g., Normal, t, Poisson)
   - 参数值 (e.g., 均值, 方差, 回归系数)
   - 感兴趣的统计量（估计量、检验统计量等）
   - 数据生成机制 (DGP: Data Generating Process)

Example>

生成 $n = 100$ 个 $X_i \sim N(0,1)$，$Y_i = \beta_0 + \beta_1 X_i + \varepsilon_i$，
其中 $\varepsilon_i \sim N(0,1)$。

---

2. **确定要变化的输入因子 (Identify Factors to Vary)**
   - 样本量 (n)
   - 参数 (β_1, σ^2)
   - 分布类型 (Normal / t / skewed)
   - 相关结构 (independent vs correlated)
     每种组合形成一个 **scenario (情景)**。

---

3. **编写实验函数 (Write Simulation Code)**
   - 该函数接收上述输入参数；
   - 返回感兴趣的统计量（如估计偏差、方差等）。

Python

```python
def simulate_once(n, beta, dist='normal'):
    X = np.random.normal(size=n)
    if dist == 't':
        eps = np.random.standard_t(df=3, size=n)
    else:
        eps = np.random.normal(size=n)
    Y = beta * X + eps
    est = np.sum(X * Y) / np.sum(X**2)
    return est
```

---

4. **重复实验 (Repeat the Experiment m Times)**
   - 对每个 scenario 重复模拟 m 次。
   - 这可高度并行化（并行于 scenario 与 replicate 两个维度）。

Python

```python
results = [simulate_once(100, 1.0) for _ in range(1000)]
```

---

5. **总结与不确定性量化 (Summarize Results & Quantify Uncertainty)**
   - 计算均值、标准差、偏差、MSE 等；
   - 模拟标准误差：

$$SE_{sim} = \frac{s}{\sqrt{m}}$$

   - 若 SE_{sim} 远小于效应量，可忽略模拟误差。

---

6. **可视化与报告 (Visualization and Reporting)**
   - 表格呈现 bias、RMSE、coverage；
   - 图形化呈现 (箱线图、误差条、曲线对比)。

6 各种重要考量 (Design Considerations)

1. **效率与可重现性**
   - 代码应结构化、模块化；
   - 记录随机种子以确保可复现。
2. **与真实数据相似的结构**
   - 模拟分布、参数、依赖性应与目标问题相符；
   - 可引入异常值、随机效应等以测试鲁棒性。
3. **控制随机变异 (Paired Designs)**
   - 在比较两种方法时，**使用相同的模拟数据集**，
     即固定随机数种子，使两方法共享相同噪声。
   - 分析上可使用配对差值以降低方差。
4. **控制随机数的一致性**
   - 例如，在比较 t 分布与 Normal 分布数据时，可以通过相同的累积分布概率匹配：

Python

```python
from scipy.stats import t, norm
import numpy as np, matplotlib.pyplot as plt

devs = np.random.normal(size=100)
tdevs = t.ppf(norm.cdf(devs), df=1)

plt.scatter(devs, tdevs)
plt.xlabel('Normal Deviates')
plt.ylabel('t Deviates (df=1)')
plt.plot([min(devs), max(devs)], [min(devs), max(devs)], color='red')
plt.show()
```

> ⚠ **Remark**
>
> 这样可以使两组数据“成对对应”，减少随机误差的影响。

6.1 重复次数 m 的选择 (Choosing m)

- 理论上可用基本功效 (power) 计算决定 m；
- 实践中常采用**顺序方式 (sequential)**：
  - 逐步增加 m，直到模拟结果的精度达到可接受水平；
  - 若 $s/\sqrt{m}$ 已小于差异效应量，可停止。

7 实验设计思想在模拟中的应用 (Optional)

7.1 多因素输入的情景 (Multiple Input Variables)

- 通常需评估多个输入变量对输出结果的影响；
- 避免“一次只变一个变量”的低效方式。

7.2 全因子设计 (Full Factorial Design)

- 若输入较少，可离散化为少量水平（levels），进行全因子设计：
  k 个因子，每个有 L 个水平 $\rightarrow L^k$ 个情景。

> **Example**
>
> 示例：3 个输入 × 3 个水平 = $3^3 = 27$ 个组合。

- 适用于输入数量较少的情况。

7.3 分数因子设计 (Fractional Factorial Design)

- 当输入或水平较多时，无法执行全因子设计；
- 分数因子设计通过选择部分组合来保证：
  - 平衡性；
  - 可估计主效应与部分交互项；
  - 高阶交互项与低阶项混叠（aliasing）。

> **Remark**
>
> 目标：在计算资源有限的前提下，最大化可解释性。

7.4 Latin Hypercube Sampling (LHS)

- 适用于输入维度很高的情况；
- 在多维输入空间内均匀采样，保证覆盖性。

步骤：

1. 假设每个输入变量 $X_j \sim U(0,1)$；
2. 将区间 [0,1] 均分为 m 个区间；
3. 在每个区间内随机采样一个点；
4. 对每个变量独立地随机排列区间顺序；
5. 组合形成 m 个输入点。

> **Remark**
>
> 这样可在有限样本下均匀探索输入空间。
> 分析时关注主效应与一、二阶交互作用。

7.5 结果解释建议

- 重点放在**方差分解与效应大小 (effect magnitude)**，而非显著性检验；
- 在模拟研究中，“零假设成立”通常不具意义。

---

8 Computational Efficiency

8.1 并行计算与仿真研究 (Parallel Processing for Simulations)

- **仿真研究的特点**
  仿真（simulation）通常是高度可并行的任务（embarrassingly parallel）：
  - 每个重复试验（replicate）可以独立运行在不同的 CPU 核上
  - 运行结束后再将结果汇总
  - 理论上，运行速度应与处理器数量近似线性增长
- **C/C++ 编译优化**
  若仿真中包含大量循环或计算密集型部分，可考虑：
  - 使用 **C/C++** 编写核心函数
  - 在 Python 中通过绑定（linking）调用已编译代码
  - 这种方式能显著提升性能
  - 虽然本课程不涉及，但这是常见的高效策略

8.2 Python 中的笛卡尔积工具 (Using `itertools.product`)

在仿真设计中，我们常需要遍历多个参数的所有组合。
Python 提供 `itertools.product` 生成参数笛卡尔积：

Python

```python
import itertools

thetaLevels = ["low", "med", "hi"]
n = [10, 100, 1000]
tVsNorm = ["t", "norm"]
levels = list(itertools.product(thetaLevels, tVsNorm, n))
```

结果：

Python

```python
[("low", "t", 10), ("low", "t", 100), ("low", "t", 1000),
 ("low", "norm", 10), ..., ("hi", "norm", 1000)]
```

应用场景：
用于生成实验条件、超参数组合或多场景仿真设定。

---

9 Analysis and Reporting

9.1 报告结果的方式 (Reporting Simulation Results)

- **表格与图形的选择**
  - 仿真结果可用表格展示（如均值、方差、覆盖率等）
  - 若有多个情境（scenarios），图形可能更直观
  - 但并非所有情况图形都优于表格，应具体分析
- **多维结果展示**
  在多输入变量情境下：
  - **R** 中可使用 `ggplot2::facet_wrap()` 绘制 trellis plots
  - **Python** 中可用 `pandas.plot`，`seaborn`，`plotly` 等工具实现类似功能

9.2 随机性与可复现性 (Reproducibility in Simulations)

- **固定随机种子 (set the seed)**
  在实验开始前设置随机数种子，以确保结果可复现

  Python

```python
import numpy as np
np.random.seed(123)
```

  - 在保存中间结果时，也应保存当前随机数状态
  - 对于 **MCMC** 等逐步更新算法，这样可从中断处继续运行

- **代码与数据的可公开性 (Code and Data Transparency)**
  为促进可复现性，应将：
  - 仿真代码
  - （若可行）仿真数据

  上传至：
  - GitHub
  - 个人网站
  - 期刊附录 (Supplementary Materials)

  他人应能完全再现结果，包括随机数生成与种子设置。

9.3 学术期刊对可复现性的要求 (Reproducibility Standards)

美国统计学会（ASA）对其期刊的计算型论文提出了如下要求：

> “ASA 强烈建议作者提交与论文相关的数据集、代码、程序或附录。这些材料有助于推动可复现研究的专业标准。
>
> - 使用任何数据集时，应完整记录其来源并在网上附录中提供。
> - 出于安全或保密原因，可向编辑申请豁免。
> - 若研究依赖特定代码实现，应尽可能公开该代码。
>
> 对于基于计算结果的论文，应提供足够信息以便读者评估结果质量，包括：
>
> - 结果的估计精度
> - 伪随机数生成器的描述
> - 所用数值算法
> - 编程语言与主要软件组件。”

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。

10 Generating Random Uniforms on a Computer

10.1 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。

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

一种常见的 RNG 类型为**序列同余法**：

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

通常 $j = 1$，其中：

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

10.2.1 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

```python
n = 100
a = 171
m = 30269
x = np.empty(n)
x[0] = 7306
for i in range(1, n):
    x[i] = (a * x[i-1]) % m
u = x / m
```

10.2.2 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。

10.3 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

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

该算法提升了多维独立性，是早期 R 默认的 RNG。

10.4 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)**。

10.4.1 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。

10.4.2 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()`。

10.5 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。

10.5.1 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。

10.5.2 实现与可访问性

在 NumPy 中通过：

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

或旧接口：

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

10.6 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。

10.7 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
rng = np.random.default_rng(seed=1)
saved_state = rng.bit_generator.state
saved_state['state']['state']   # 状态
saved_state['state']['inc']     # 增量 c
```

11 RNG in Python

11.1 Choosing a Generator

NumPy 1.17+ 提供新接口：

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

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

11.2 Using PCG-64

```python
rng = np.random.default_rng(seed=1)
rng.normal(size=5)
# array([ 0.34558419, 0.82161814, 0.33043708, -1.30315723, 0.90535587])

saved_state = rng.bit_generator.state
rng.normal(size=5)
# array([ 0.44637457, -0.53695324, 0.5811181 , 0.3645724 , 0.2941325 ])

rng.bit_generator.state = saved_state  # 恢复状态
rng.normal(size=5)
# 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
saved_state
# {'bit_generator': 'PCG64', 'state': {'state': 216676376075457487203159048251690499413,
'inc': 194290289479364712180083596243593368443}, 'has_uint32': 0, 'uinteger': 0}

saved_state['state']['state'] # actual state
# 216676376075457487203159048251690499413

saved_state['state']['inc'] # increment ('c')
```

```
# 194290289479364712180083596243593368443
```

11.3 Using Mersenne Twister (optional)

旧接口：

Python

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

保存并恢复状态：

Python

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

12 RNG in Parallel

12.1 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

```python
# 在所有进程中
np.random.seed(1)
```

会导致各进程得到完全相同的随机数流。

12.2 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)

13 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

Logic

在仿真研究中，我们需要从各种常见分布（如正态、Gamma、Beta、Poisson、t 等）中生成随机变量。

这些分布的采样方法通常已内置于 **Python** 和 **R** 中，且实现使用了高效且可靠的算法，因此这里仅介绍其核心原理。

许多统计计算与 Monte Carlo 方法的教材（如 Gentle、Robert & Casella）中对这些方法有详细论述。

其中多数方法利用了**分布间的关系**——例如，Beta 随机变量可由两个 Gamma 随机变量构造而得。

---

14 Multivariate Distributions

对于多元分布（如多元正态、多元 t），
`mvtnorm`（R 包）或 `scipy.stats.multivariate_normal`（Python）提供了概率密度与 CDF 计算工具。

14.1 多元正态的生成方法

若协方差矩阵为 Σ，则可通过 **Cholesky 分解** 生成：

Python

```python
L = np.linalg.cholesky(covMat)  # L 为下三角矩阵
x = L @ np.random.normal(size = covMat.shape[0])
```

此时生成的 x 满足 $x \sim \mathcal{N}(0, \Sigma)$。

Remark

若协方差矩阵奇异，可使用带 pivoting 的 Cholesky 分解，将秩亏部分对应行置零，从而生成满足约束的随机向量。但需注意输出向量的重新排序。

---

15 Inverse CDF Method (反CDF法)

要生成 $X \sim F$，其中 F 是累积分布函数 (CDF)，若 F 可逆，则可按以下步骤：

1. 生成 $Z \sim \mathcal{U}(0,1)$；
2. 设 $X = F^{-1}(Z)$。

即：

$$X = F^{-1}(Z), \quad Z \sim \mathcal{U}(0,1)$$

对于离散分布，可使用离散化的 CDF。
对于多元分布，可按条件分布顺序采样：

$$f(x_1)f(x_2|x_1)f(x_3|x_1,x_2)\cdots f(x_k|x_1,\ldots,x_{k-1})$$

该方法在存在解析条件分布时尤为有效。

---

16 Rejection Sampling (拒绝采样)

16.1 基本思想

若目标密度为 $f(x)$，我们希望从中采样。
引入辅助变量 u，则可写作：

$$f(x)=\int_0^{f(x)} du$$

即 (X,U) 的联合分布为：

$$(X,U)\sim\mathcal{U}(x,u):0<u<f(x)$$

但我们无法直接从 $f(x)$ 采样，因此使用“包络函数” $cg(x)$ 来近似 $f(x)$，
其中 $g(x)$ 易于采样，且存在常数 $c>0$ 满足：

$$cg(x)\geq f(x),\quad \forall x$$

16.2 算法步骤

1. 生成 $x\sim g(x)$；
2. 生成 $u\sim\mathcal{U}(0,1)$；
3. 若 $u\leq\dfrac{f(x)}{cg(x)}$，则接受 x；否则拒绝并重采样。

16.3 接受概率

接受率为：

$$P(\text{accept})=\frac{1}{c}=\frac{\int f(x)dx}{\int cg(x)dx}$$

因此，理想的 $g(x)$ 应：

- 具有比 $f(x)$ 更“胖的尾部”；
- 使 c 尽量小，以减少拒绝次数。

16.4 几何直观

我们在 $cg(x)$ 曲线下均匀采样 (x,u)，
仅保留位于 $f(x)$ 下方的点，即 $u<f(x)/cg(x)$。

16.5 拒绝采样中的 Squeezing 技巧

若 $f(x)$ 计算代价高，可使用其下界 $f_{\text{low}}(x)$：

$$u\leq\frac{f_{\text{low}}(x)}{cg(x)}$$

满足条件时可直接接受，避免计算完整 $f(x)$，从而减少计算负担。

16.6 例：截断正态分布

若标准正态分布截断点在 $a > 0$，
直接采样并拒绝效率极低。
Robert (1995) 提出用**平移指数分布**作为包络函数：

$$g(x) = \lambda e^{-\lambda(x-a)}, \quad x > a$$

其中 λ 根据截断点选择，使 $cg(x)$ 能包络正态尾部。
当 $a < 0$ 时，可通过取负对称实现。

---

17 Adaptive Rejection Sampling (自适应拒绝采样) (optional)

RS 的难点在于选取良好的包络函数。
自适应拒绝采样 (ARS) 适用于连续、可微且对数凹的密度。

17.1 基本思想

1. 在 $\log f(x)$ 空间上，用切线与割线构造上下包络；
2. 上包络对应分段指数函数；
3. 从上包络采样，再通过“squeezing”判定是否接受；
4. 每次接受需要计算 $f(x)$ 的新点后，更新包络。

ARS 能动态改进效率，但要求 $f(x)$ 对数凹。

---

18 Importance Sampling

18.1 目标

估计期望：

$$\phi = E_f[h(Y)] = \int h(y)f(y)dy$$

若直接从 $f(y)$ 采样困难，可引入辅助分布 $g(y)$，则：

$$\phi = \int h(y)\frac{f(y)}{g(y)}g(y)dy$$

Monte Carlo 估计量为：

$$\hat{\phi} = \frac{1}{m}\sum_{i=1}^{m} h(y_i)\frac{f(y_i)}{g(y_i)}, \quad y_i \sim g(y)$$

定义权重：

$$w_i = \frac{f(y_i)}{g(y_i)}$$

若 f 仅已知至比例常数（如贝叶斯情境），使用归一化权重：

$$w_i^* = \frac{w_i}{\sum_j w_j}$$

18.2 性质与建议

- 只需 f, g 有**相同 support**，不必要求 $cg(x) \geq f(x)$；

- 若 g 尾部比 f 轻，方差可能爆炸；
- 为降低方差，应确保 w_i 大时 $h(y_i)$ 小，即避免过度权重。

估计量方差为：

$$\mathrm{Var}(\hat{\phi}) = \frac{1}{m}\mathrm{Var}\left(h(Y)\frac{f(Y)}{g(Y)}\right)$$

启发：
若 $h(y)$ 大的区域贡献主要，$g(y)$ 应更多地采样这些区域。
在稀有事件估计中，这意味着**过采样稀有事件**，再用权重修正。

18.3 Sampling Importance Resampling (SIR)

若目标是从 f 中生成样本而非仅估计期望：

1. 从 $g(y)$ 生成样本 y_i；
2. 根据权重 w_i 进行有放回重采样。

此过程称为 **SIR（Sampling Importance Resampling）**。

---

19 Ratio of Uniforms (比值法) (optional)

该方法基于如下构造：

若 (U, V) 在集合

$$C = (u, v) : 0 \le u \le \sqrt{f(v/u)}$$

上均匀分布，则随机变量：

$$X = \frac{V}{U}$$

的密度与 $f(x)$ 成比例。

19.1 算法

1. 选取矩形区域包含 C：

$$0 \le u \le \sup_x \sqrt{f(x)}, \quad \inf_x x\sqrt{f(x)} \le v \le \sup_x x\sqrt{f(x)}$$

2. 在矩形中采样 (u, v)；
3. 若 $u \le \sqrt{f(v/u)}$，接受并令 $x = v/u$。

可根据 $f(x)$ 形状裁剪矩形以提升效率。

推荐阅读：Monahan (2001)，在其第 323 页提出适用于离散分布的改进版比值法。