Chapter 1 - Random Number Generation

Generating Random Variables in R and Python. Discrete Distributions.

Prof. Alex Alvarez, Ali Raisolsadat

School of Mathematical and Computational Sciences
University of Prince Edward Island

Introduction to Random Numbers

Randomness has played an important role for centuries in games, gambling, and statistics.

Historically, random numbers were generated manually or mechanically (e.g., spinning wheels, rolling dice, shuffling cards, or using random number tables).

The main building block for all the simulation algorithms that we will study in this course is the generation of random numbers with a given distribution. This includes both discrete and continuous random variables.

If we are using a powerful statistical software such as **R** (but also **Python** or **MATLAB**, and most used programming languages) there are already some built-in functions that can hep us generate random numbers with well known probability distributions.

On the other hand, we will also learn some techniques to deal with arbitrary probability distributions (generally not supported by any software).

Pseudo Random Numbers Generators

The algorithms used by most software packages to generate random numbers with a given distribution are **deterministic** algorithms, meaning that the generated random numbers are not “truly random”. We call them **pseudo random number generators** (PRNG).

In other words, a PRNG is an algorithm which outputs a sequence of random numbers with properties that are very similar to the properties of sequences of true random numbers. PRNGs are fast and efficient, and for most applications they generate sequences of random numbers that can replace true random numbers without major issues.

Example: The most common method generates numbers starting from an initial value x_0 (the **seed**) and recursively computes successive values x_n for $n \geq 1$ using

$$x_n = a x_{n-1} \mod m$$

where a and m are positive integers. The remainder operation ensures that $x_n \in \{0, 1, \ldots, m-1\}$. On a 32-bit machine (with the first bit as a sign bit), the choice $m = 2^{31} - 1$ and $a = 7^5 = 16807$ produces desirable properties.

Some Probability Distributions Supported by R and Python

Distribution	Name in R	Name in Python (SciPy)
Uniform distribution	`unif`	`uniform`
Normal distribution	`norm`	`norm`
Poisson distribution	`pois`	`poisson`
Binomial distribution	`binom`	`binom`

In **R**, the names of functions associated with these probability distributions are constructed by adding a prefix letter to the distribution name:

- **r**: random number generation (e.g., `runif`)
- **d**: density (or probability mass function for discrete distributions
- **p**: cumulative distribution function (CDF)
- **q**: quantile function

Some Probability Distributions Supported by R and Python

Distribution	Name in R	Name in Python (SciPy)
Uniform distribution	`unif`	`uniform`
Normal distribution	`norm`	`norm`
Poisson distribution	`pois`	`poisson`
Binomial distribution	`binom`	`binom`

In **Python** (**SciPy** and **Numpy**), these distributions are typically accessed via `scipy.stats`. For example:

- `stats.uniform.rvs(size=10)` generates random samples.
- `stats.uniform.pdf(x)` evaluates the probability density function.
- `stats.uniform.cdf(x)` computes the CDF.
- `stats.uniform.ppf(q)` gives the quantile function (percent-point function).

Example: `runif(10)` in **R** or `stats.uniform.rvs(size=10)` in **Python** generate 10 random numbers from a uniform distribution.

From Built-in Distributions to Custom Sampling

In practice, most standard probability distributions are already implemented in statistical software such as **R** and **Python**. These tools allow us to generate random variables with a single command.

It is important to remember that these functions are ultimately based on more fundamental ideas:

- Generating random numbers from the *Uniform*(0,1) distribution
- Transforming those numbers to follow a desired distribution

Understanding these basic mechanisms allows us to:

- Generate random variables from distributions that are *not* built into software
- Modify or customize probability models
- Better understand how simulation algorithms work internally

We now illustrate this idea by constructing a discrete random variable *from scratch* using a uniform random number.

Generation of Random Numbers with Discrete Distributions

Example: Generate a random number from a random variable X with discrete probability distribution given by the p.m.f. $p(i) = i/6$ for $i = 1, 2, 3$. That is $P(X = 1) = 1/6$, $P(X = 2) = 1/3$ and $P(X = 3) = 1/2$.

Algorithm

1. Generate $U \sim U(0, 1)$
2. Define $X = \begin{cases} 1 & \text{if } 0 \leq U < 1/6 \\ 2 & \text{if } 1/6 \leq U < 1/2 \\ 3 & \text{if } 1/2 \leq U \leq 1 \end{cases}$
3. Return X

Generation of Random Numbers with Discrete Distributions

Code

R Code

```r
U <- runif(1, min=0, max=1)
if (U < 1/6) {
    X <- 1
} else if (U < 1/2) {
    X <- 2
} else {
    X <- 3
}
print(X)
```

Python Code

```python
import numpy as np

U = np.random.uniform(0, 1)
if U < 1/6:
    X = 1
elif U < 1/2:
    X = 2
else:
    X = 3

print(X)
```

Generation of Random Numbers with Discrete Distributions

Suppose that we want to generate the value of a discrete random variable X having probability mass function

$$P\{X = x_j\} = p_j, \quad j = 0, 1, \ldots, \quad \sum_{j=0}^{\infty} p_j = 1.$$

To accomplish this, we generate a random number $U \sim \text{Uniform}(0, 1)$ and set

$$X = \begin{cases} x_0, & \text{if } U < p_0, \\ x_1, & \text{if } p_0 \leq U < p_0 + p_1, \\ \vdots & \end{cases}$$

Since for $0 < a < b < 1$, $P\{a \leq U < b\} = b - a$, we have

$$P\{X = x_j\} = P\Big\{\sum_{i=0}^{j-1} p_i \leq U < \sum_{i=0}^{j} p_i\Big\} = p_j$$

and so X has the desired distribution.

Generation of Random Numbers with Discrete Distributions

Example: If $P\{X = j\} = 1/n,\ j = 0, \ldots, n-1$, then

$$X = j \quad \text{if} \quad \frac{j-1}{n} \leq U < \frac{j}{n}$$

If U is uniformly distributed in $[0, 1]$ we can see that by defining $X = \lfloor nU \rfloor$, the random variable X follows the target distribution.

$$X = \lfloor nU \rfloor$$

where $\lfloor \cdot \rfloor$ denotes the rounding down operation.

Generation of random numbers with discrete distributions

Problem: Consider $n = 50$. Generate a random number from a random variable X with discrete probability distribution given by the p.m.f. $p(i) = 1/n$ for $i = 0, 1, \ldots, n-1$.

Algorithm

1. Generate $U \sim U(0,1)$
2. Define $X = \lfloor nU \rfloor$
3. Return X

Code

```r
k <- 50
U <- runif(1, min=0, max=1)
X <- floor(k*U)
print(X)
```

```python
import numpy as np

k = 50
U = np.random.uniform(0, 1)
X = np.floor(k*U)
print(int(X))
```

General Algorithm

The preceding can be written algorithmically as:

$$\begin{cases} \text{Generate a random number } U, \\ \text{If } U < p_0, \text{ set } X = x_0 \text{ and stop,} \\ \text{If } U < p_0 + p_1, \text{ set } X = x_1 \text{ and stop,} \\ \text{If } U < p_0 + p_1 + p_2, \text{ set } X = x_2 \text{ and stop,} \\ \vdots \end{cases}$$

If the $x_i, i \geq 0$ are ordered so that $x_0 < x_1 < x_2 < \dots$, and if we let F denote the distribution function of X, then $F(x_k) = \sum_{i=0}^{k} p_i$, and

$$X = x_j \quad \text{if and only if} \quad F(x_{j-1}) \leq U < F(x_j).$$

In other words, after generating U, we determine X by finding the interval $[F(x_{j-1}), F(x_j))$ in which U lies.

This is equivalent to computing the inverse transform $X = F^{-1}(U)$.

Generate Discrete Random Variable Algorithm

Algorithm 1 Naive Inverse Transform Sampling for Discrete RV

```
1: Input: Probabilities $p_0, \dots, p_{n-1}$, Outcomes $x_0, \dots, x_{n-1}$
2: Draw $U \sim \text{Uniform}(0, 1)$
3: Initialize cumulative probability: $C \leftarrow 0$
4: for $i = 0$ to $n - 1$ do                                  ▷ Loop over outcomes
5:     $C \leftarrow C + p_i$
6:     if $U < C$ then
7:         return $x_i$
8:     end if
9: end for
10: return $x_{n-1}$                                          ▷ Handles rounding edge cases
```

Example: Step-by-Step Calculation

Generate a random number from X with

$$P(X = 1) = 1/6, \quad P(X = 2) = 1/3, \quad P(X = 3) = 1/2$$

Assume we draw $U = 0.42$

- Initialize cumulative probability: $C = 0$
- Step 1: $i = 1$, $x_1 = 1$, $p_1 = 1/6 \approx 0.1667$

$$C \leftarrow C + p_1 = 0 + 0.1667 = 0.1667$$

  Check: $U < C$? $0.42 < 0.1667 \rightarrow$ **No**
- Step 2: $i = 2$, $x_2 = 2$, $p_2 = 1/3 \approx 0.3333$

$$C \leftarrow C + p_2 = 0.1667 + 0.3333 = 0.5$$

  Check: $U < C$? $0.42 < 0.5 \rightarrow$ **Yes!** $\rightarrow X = 2$
- Step 3: Not needed, algorithm stops.

Result: $X = 2$

Efficient Sampling of a Discrete Random Variable

Algorithm 2 Generate X with probabilities $p_1, \dots, p_n$ using sorted cumulative sums

```
1:  Input: Probabilities $p_1, \dots, p_n$, Outcomes $x_1, \dots, x_n$
2:  Sort the pairs $(p_i, x_i)$ in descending order of $p_i$:
        $$(p_{(1)}, x_{(1)}) \geq (p_{(2)}, x_{(2)}) \geq \cdots \geq (p_{(n)}, x_{(n)})$$
3:  Initialize cumulative probability: $C \leftarrow 0$
4:  Draw $U \sim \text{Uniform}(0, 1)$
5:  for $j = 1$ to $n$ do
6:      $C \leftarrow C + p_{(j)}$
7:      if $U < C$ then
8:          return $x_{(j)}$
9:      end if
10: end for
11: return $x_{(n)}$                                  ▷ Handles rounding edge cases
```

Generate Discrete Random Variable – Efficient Algorithm

What is different here?

By ordering outcomes by probability, the algorithm reduces the **expected number of comparisons**.

The expected number of comparisons is

$$\mathbb{E}[\text{comparisons}] = \sum_{i=1}^{n} i \cdot p_{(i)}.$$

Example: Geometric Distribution

In some cases however, we would prefer an explicit expression connecting X and U, instead of listing all possible cases.

Let X be a geometric random variable with parameter p if

$$P\{X = i\} = p_i = p^{i-1}(1-p), \quad i \geq 1$$

From this expression we can see that:

$$\sum_{i=1}^{j} p_i = \sum_{i=1}^{j} p^{i-1}(1-p) = (1-p)\frac{1-p^j}{1-p} = 1 - p^j$$

The event $\left\{U < \sum_{i=1}^{j} p_i\right\}$ is equivalent to the event $\left\{j > \frac{\ln(1-U)}{\ln(p)}\right\}$

This means that by rounding up $\frac{\ln(1-U)}{\ln(p)}$ we find the appropriate value of j.

Note that $\log(p) < 0$ for $0 < p < 1$, so the inequality flips as shown above.

Algorithm to Generate a Random Number From the Geometric Distribution

Algorithm

1. Generate $U \sim U(0,1)$
2. Define $X = \left\lceil \frac{\ln(1-U)}{\ln(p)} \right\rceil$
3. Return X

Remark: The expression $\lceil \cdot \rceil$ denotes the operation of rounding up to the nearest integer.

Algorithm to Generate a Random Number From the Geometric Distribution

Problem: Generate a sample of 10 independent random numbers from the Geometric distribution with parameter $p = 0.4$.

Code

R Code

```r
n <- 10
p <- 0.4
U <- runif(n, min=0, max=1)
X <- ceiling(log(1-U)/log(p
    ))
print(X)
```

Python Code

```python
import numpy as np

n = 10
p = 0.4
U = np.random.uniform(0, 1, n)
X = np.ceil(np.log(1-U) / np.log(p
    )).astype(int)
print(X)
```

Homework Problems

1. Implement Algorithm 1 and Algorithm 2 and test them on the slide 5 example.
2. Consider a discrete random variable taking values $1, 3, 5$ with probabilities $1/9$, $3/9$, and $5/9$ respectively. Write a computer program to generate samples of arbitrary size n from this distribution. Create a histogram with a sample of size 1000.
3. Consider a random variable taking values $1, 3, 5, \ldots, 99$ with probabilities $1/50^2, 3/50^2, \ldots, 99/50^2$. Write a computer program to generate samples of arbitrary size n from this distribution. Create a histogram with a sample of size 1000.
4. For the slide 5 example, compute the expected number of comparisons using Algorithm 1 and Algorithm 2.
5. Write an algorithm to generate samples from a Poisson distribution:

   $$p_i = P\{X = i\} = \frac{e^{-\lambda}\lambda^i}{i!}, \quad i = 0, 1, 2, \ldots$$

   using the recursive identity:

   $$p_{i+1} = \frac{\lambda}{i+1} p_i, \quad i \geq 0$$