Chapter 4

Generating Discrete Random Variables

Ali Raisolsadat

School of Mathematical and Computational Sciences
University of Prince Edward Island

Table of contents

The Inverse Transform Method

Generate Discrete Random Variable

- 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

$$X = \begin{cases} x_0, & \text{if } U < p_0, \\ x_1, & \text{if } p_0 \leq U < p_0 + p_1, \\ \vdots & \\ x_j, & \text{if } \sum_{i=0}^{j-1} p_i \leq U < \sum_{i=0}^{j} p_i, \\ \vdots & \end{cases}$$

- 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.

Generate Discrete Random Variable

1. The preceding can be written algorithmically as:

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

2. 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 $X = F^{-1}(U)$.
- For this reason, this approach is called the discrete **inverse transform method**.

Generate Discrete Random Variable Algorithm

Algorithm 1 Generate X with probabilities $p_0, \ldots, p_n$ using Uniform$(0, 1)$ numbers

Input: Probabilities $p_0, \ldots, p_n$ and outcomes $x_0, \ldots, x_n$
Draw $U \sim$ Uniform$(0, 1)$
Initialize cumulative probability: $C \leftarrow 0$
for $i = 0$ to n **do** ▷ Loop over outcomes
    Update cumulative probability: $C \leftarrow C + p_i$
    if $U < C$ **then**
        Set $X \leftarrow x_i$ and **stop**
    end if
end for
Output: Sampled value X

Generate Discrete Random Variable – Efficient Algorithm

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

Input: Probabilities $p_1, \dots, p_n$ and outcomes $x_1, \dots, x_n$

Sort outcomes in **descending** order of probability:

$$p_{(1)} \geq p_{(2)} \geq \cdots \geq p_{(n)}, \quad x_{(1)}, \dots, x_{(n)}.$$

Compute cumulative sums $F_{(j)} = \sum_{i=1}^{j} p_{(i)}, \quad j = 1, \dots, n.$

Generate $U \sim \text{Uniform}(0,1)$

for $j = 1$ to n **do**

    if $U < F_{(j)}$ **then**

        Set $X \leftarrow x_{(j)}$ and **stop**

    end if

end for

Output: Sampled value X

- What is different here?
- By ordering outcomes by probability, the algorithm reduces the **expected number of comparisons**.
- Let outcomes be ordered by descending probability:

$$p_{(1)} \geq p_{(2)} \geq \cdots \geq p_{(n)}.$$

- The expected number of comparisons is

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

Example 4a: Simulating a Discrete Random Variable

- **Example 4a:** Simulate X such that

$$p_1 = 0.20, \quad p_2 = 0.15, \quad p_3 = 0.25, \quad p_4 = 0.40, \quad \text{where } p_j = P\{X = j\}$$

- Generate $U \sim \text{Uniform}(0, 1)$ and:

$$\begin{cases} \text{If } U < 0.20 \text{ set } X = 1 \text{ and stop} \\ \text{If } U < 0.35 \text{ set } X = 2 \text{ and stop} \\ \text{If } U < 0.60 \text{ set } X = 3 \text{ and stop} \\ \text{Otherwise set } X = 4 \end{cases}$$

- **More efficient ordering:**

$$\begin{cases} \text{If } U < 0.40 \text{ set } X = 4 \text{ and stop} \\ \text{If } U < 0.65 \text{ set } X = 3 \text{ and stop} \\ \text{If } U < 0.85 \text{ set } X = 1 \text{ and stop} \\ \text{Otherwise set } X = 2 \end{cases}$$

Expected Number of Comparisons

- For the original ordering:

$$p_1 = 0.20,\ p_2 = 0.15,\ p_3 = 0.25,\ p_4 = 0.40$$

$$\mathbb{E}[\text{comparisons}] = 1(0.20){+}2(0.15){+}3(0.25){+}4(0.40) = 2.25 \quad \text{(slower on average)}$$

- For the sequential ordering:

$$p_4 = 0.40,\ p_3 = 0.25,\ p_1 = 0.20,\ p_2 = 0.15$$

$$\mathbb{E}[\text{comparisons}] = 1(0.40) + 2(0.25) + 3(0.20) + 4(0.15) = 1.95$$

- Sorting by probability minimizes expected comparisons and makes simulation faster.

Generating a Discrete Uniform Random Variable

- No searching is necessary when X is discrete uniform.
- If $P\{X = j\} = 1/n,\ j = 1, \ldots, n$, then

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

- Equivalently:

$$X = \mathrm{Int}(nU) + 1$$

  where $\mathrm{Int}(x)$ is the largest integer $\leq x$.

Example 4b: Random Permutation (Fisher–Yates Shuffle)

- Uniform random permutations are essential for simulations, fair sampling, randomized algorithms, and applications like shuffling or task assignment. Each ordering being equally likely ensures unbiased and unpredictable outcomes.
- Suppose we are interested in generating a permutation of numbers $1, 2, \ldots, n$ such that all $n!$ possible orderings are equally likely.
- **Goal:** Produce a uniformly random permutation of the numbers $1, 2, \ldots, n$, ensuring every possible ordering occurs with equal probability.
- **Recall**: $\text{Int}(kU) + 1$ is uniformly distributed over $\{1, 2, \ldots, k\}$ when $U \sim \text{Uniform}(0, 1)$. This allows us to select a random index in a given range.

Example 4b: Random Permutation (Fisher–Yates Shuffle)

- **Procedure:**
  1. Initialize $P_1, P_2, \ldots, P_n$ with any ordering of $1, 2, \ldots, n$ (e.g., $P_j = j$).
  2. Set $k = n$.
  3. Generate $U \sim \text{Uniform}(0, 1)$ and set $I = \text{Int}(kU) + 1$.
  4. Swap P_I and P_k.
  5. Set $k \leftarrow k - 1$; if $k > 1$, return to Step 3.
  6. Output $P_1, \ldots, P_n$ as the uniformly random permutation.
- Randomly choose one of the n numbers and place it in position n.
- Randomly choose one of the remaining $n - 1$ numbers and place it in position $n - 1$.
- Continue until all positions are filled.
- It is more efficient to keep the numbers in an ordered list and swap positions instead of repeatedly searching for the numbers.

Algorithm 3 Random Permutation (Fisher–Yates Shuffle)

Input: Integer n
Initialize $P = [1, 2, \ldots, n]$
for $k = n$ down to 2 **do**
    Generate $U \sim \text{Uniform}(0, 1)$
    $I \leftarrow \mathrm{Int}(kU) + 1$
    if $I \neq k$ **then**
        Swap $P[I]$ and $P[k]$
    end if
end for
Output: P (a uniformly random permutation)

Quick Numerical Example: ($n = 4$)

- Initialize array:

$$P = [1, 2, 3, 4]$$

- Pre-generated Uniform values:

$$U_1 = 0.3, \quad U_2 = 0.7, \quad U_3 = 0.1$$

- **Step 1:** $k = 4$

$$I = \text{Int}(4 \cdot U_1) + 1 = 2, \quad P[2] \leftrightarrow P[4] \implies P = [1, 4, 3, 2]$$

- **Step 2:** $k = 3$

$$I = \text{Int}(3 \cdot U_2) + 1 = 3, \quad I = k \implies \text{skip}$$

- **Step 3:** $k = 2$

$$I = \text{Int}(2 \cdot U_3) + 1 = 1, \quad P[1] \leftrightarrow P[2] \implies P = [4, 1, 3, 2]$$

- Final random permutation:

$$P = [4, 1, 3, 2]$$

Generating a Random Subset of Size r

- Often, in sampling or simulations, we only need a smaller representative set of elements rather than the entire population. Choosing r allows control over sample size and reduces computational cost.
- To generate a random subset of size r from $\{1, 2, \dots, n\}$:
  1. If $r > n/2$, set $r \leftarrow n - r$ and note that the final subset will consist of the elements **not** in the generated subset. This reduces computation when the subset is large.
  2. Initialize $P = [1, 2, \dots, n]$.
  3. Shuffle P using the Fisher–Yates algorithm (or partially, as needed).
  4. Take the first r elements of P as the random subset:

$$S = \{P_1, P_2, \dots, P_r\} \tag{1}$$

- **Observation:** Each subset of size r is equally likely.
- **Efficiency:** Stop the shuffle after r iterations; no need to shuffle the entire array if only the first r elements are needed.

Algorithm: Random Subset of $\{1, \ldots, n\}$ of Size r

Algorithm 4 Random Subset using Fisher–Yates

Input: Integers n (population size) and r (subset size)
if $r > n/2$ **then**
    Set $r \leftarrow n - r$ ▷ Generate the smaller complementary subset first
end if
Initialize $P = [1, 2, \ldots, n]$
for $k = n$ down to $n - r + 1$ **do** ▷ Partial shuffle only
    Generate $U \sim \text{Uniform}(0, 1)$
    $I \leftarrow \text{Int}(kU) + 1$ ▷ Random index in $\{1, \ldots, k\}$
    if $I \neq k$ **then**
        Swap $P[I]$ and $P[k]$
    end if
end for
Output: $S = \{P_1, P_2, \ldots, P_r\}$, a uniform random subset of size r

Application: Random Subsets in Medical Trials

- The ability to generate a random subset is particularly important in medical trials.
- **Example:** A medical center plans to test a new drug designed to reduce blood cholesterol levels.
  - 1000 volunteers have been recruited as subjects for the trial.
  - To account for external factors that might affect blood cholesterol (e.g., weather conditions), the volunteers will be split into two groups of size 500: a treatment group receiving the drug and a control group receiving a placebo.
  - The trial is conducted as a **double-blind study**, meaning that neither the volunteers nor the administrators know who is in each group.

Application: Random Subsets in Medical Trials

- To ensure the treatment and control groups are comparable in all respects except for the drug, the 500 volunteers in the treatment group must be chosen **completely at random**.
- Using a **random subset** ensures:
  - Each volunteer has an **equal chance of being selected** for the treatment group.
  - The comparison between groups is **unbiased**, so that any observed differences in response are due to the drug and not external factors.
- This method is widely used in **randomized clinical trials** and other experimental research designs to maintain fairness and statistical validity.

Example 4c: Calculating Averages

- Suppose we want to approximate

$$\bar{a} = \frac{1}{n}\sum_{i=1}^{n} a(i) \tag{2}$$

where n is large and the values $a(i), i = 1, \ldots, n$ are complicated and not easily calculated.
- One way to accomplish this is to note that if X is a discrete uniform random variable over the integers $1, \ldots, n$, then the random variable $a(X)$ has a mean given by

$$E[a(X)] = \sum_{i=1}^{n} a(i)P\{X = i\} = \frac{1}{n}\sum_{i=1}^{n} a(i) = \bar{a} \tag{3}$$

Example 4c: Calculating Averages

- We can generate k discrete uniform random variables $X_i, i = 1, \ldots, k$ by:
  - Generate k random numbers $U_i \sim \text{Uniform}(0, 1)$
  - Set

$$X_i = \text{Int}(nU_i) + 1 \tag{4}$$

- Then each of the k random variables $a(X_i)$ will have mean $\bar{a}$, and so by the **strong law of large numbers**, as $k \to \infty$ (with $k < n$), the average of these values should approximately equal $\bar{a}$:

$$\bar{a} \approx \frac{1}{k} \sum_{i=1}^{k} a(X_i) \tag{5}$$

Example 4c: Calculating Averages

- The **standard error** of this Monte Carlo approximation is

$$\mathrm{SE}(\hat{\bar{a}}) = \sqrt{\frac{\sigma^2}{k}}, \quad \sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(a(i) - \bar{a})^2 \tag{6}$$

  where σ^2 is the variance of $a(X_i)$.
- This gives a measure of how far the Monte Carlo approximation is expected to deviate from the true mean.
- The larger k, the smaller the standard error, and the more precise the approximation.

Generating Known Discrete Random Variables

Generating a Geometric Random Variable (Part 1)

- Let X be a geometric random variable with parameter p if

$$P\{X = i\} = pq^{i-1}, \quad i \geq 1, \quad \text{where } q = 1 - p$$

- Recall that X can be thought of as representing the time of the first success in independent trials, each of which succeeds with probability p.
- Cumulative probability for the first $j - 1$ trials:

$$\begin{aligned}
\sum_{i=1}^{j-1} P\{X = i\} &= 1 - P\{X > j - 1\} \\
&= 1 - P\{\text{first } j - 1 \text{ trials are all failures}\} \\
&= 1 - q^{j-1}, \quad j \geq 1
\end{aligned}$$

- To generate X, generate $U \sim \text{Uniform}(0, 1)$ and set X equal to the value j such that

$$1 - q^{j-1} \leq U < 1 - q^{j}, \quad \text{or equivalently} \quad q^{j} < 1 - U \leq q^{j-1}.$$

Generating a Geometric Random Variable (Part 2)

- Define X in a single compact expression instead of using inequalities:

$$X = \min\{j : q^j < 1 - U\}$$

- Stepwise procedure:
  - Check $j = 1$: if $q^1 < 1 - U$, then $X = 1$.
  - Otherwise, check $j = 2$: if $q^2 < 1 - U$, then $X = 2$.
  - Continue until the inequality is satisfied.
- This ensures we pick the **smallest integer** j satisfying the inequality, which automatically satisfies the original interval condition.

Generating a Geometric Random Variable (Part 3)

- Using the monotonicity of logarithms:

$$\begin{aligned} X &= \min\{j : j\log(q) < \log(1-U)\} \\ &= \min\left\{j : j > \frac{\log(1-U)}{\log(q)}\right\} \end{aligned}$$

- Note that $\log(q) < 0$ for $0 < q < 1$, so the inequality flips as shown above.
- Using integer part notation:

$$X = \mathrm{Int}\left(\frac{\log(1-U)}{\log(q)}\right) + 1$$

- Since $U \sim \mathrm{Uniform}(0,1)$, then $1 - U \sim \mathrm{Uniform}(0,1)$, giving

$$X = \mathrm{Int}\left(\frac{\log(U)}{\log(q)}\right) + 1$$

- This generates X with geometric parameter p efficiently.
- Homework: How can we generate **Bernoulli** Random Variables?

Generating a Poisson Random Variable (Part 1)

- The random variable X is Poisson with parameter λ if

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

- The key to using the inverse transform method to generate such a random variable is the recursive identity:

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

- Using this recursion, we can build an efficient algorithm for generating $X \sim \text{Poisson}(\lambda)$.

Generating a Poisson Random Variable (Part 2)

- Algorithm steps:
  1. Generate a random number $U \sim \text{Uniform}(0, 1)$.
  2. Initialize $i = 0$, $p = e^{-\lambda}$, $F = p$.
  3. If $U < F$, set $X = i$ and stop.
  4. Update
  $$p \leftarrow \frac{\lambda p}{i+1}, \quad F \leftarrow F + p, \quad i \leftarrow i + 1$$
  5. Repeat Step 3 until $U < F$.
- This procedure generates X with the correct Poisson probabilities using a simple recursive approach.

Algorithm 5 Poisson Random Variable Generation

```
Input: λ > 0
Generate U ~ Uniform(0, 1)
Initialize i = 0, p = e^{-λ}, F = p
while U ≥ F do
    i ← i + 1
    p ← λp / i
    F ← F + p
end while
Output: X = i
```

Example: Generating a Poisson Random Variable

- We illustrate the generation of Poisson random variables with $\lambda = 3$ using the recursive method.
- Recursive formula:

$$p_{i+1} = \frac{\lambda}{i+1} p_i$$

- Case 1: $U = 0$

$$\begin{aligned} &i = 0, \quad p_0 = e^{-3} \approx 0.0498, \quad F = p_0 = 0.0498 \\ &U = 0 < F \implies X = 0 \end{aligned}$$

- Case 2: $U = 0.25$

$$\begin{aligned} &i = 0, \quad p_0 = 0.0498, \quad F = 0.0498 \\ &U = 0.25 \geq F \implies \text{continue} \\ &i = 1, \quad p_1 = \frac{3 \cdot 0.0498}{1} = 0.1494, \quad F = 0.1992 \\ &U = 0.25 \geq F \implies \text{continue} \\ &i = 2, \quad p_2 = \frac{3 \cdot 0.1494}{2} = 0.2241, \quad F = 0.4233 \\ &U = 0.25 < F \implies X = 2 \end{aligned}$$

Efficient Poisson Generation (Improved Method)

- The naive approach requires $1 + X$ comparisons (on average $1 + \lambda$), which is costly for large λ.
- Instead of summing probabilities from $i = 0$ upward until we exceed U, we start the search near $i \approx \lambda$ (where the Poisson mass is concentrated) and search upward or downward from there.
- Let $I = \text{Int}(\lambda)$ and compute

$$F(I) = P(X \leq I) = \sum_{k=0}^{I} p_k$$

  using the recursive formula

$$p_{k+1} = \frac{\lambda}{k+1} p_k.$$

- Generate $U \sim U(0, 1)$ and check:
  - If $U \leq F(I)$, then $X \leq I$ and we search **downward** from I.
  - If $U > F(I)$, then $X > I$ and we search **upward** from $I + 1$.

Algorithm: Efficient Poisson Generation (Search Near λ)

Algorithm 6 Efficient Poisson Random Variable Generation

Input: $\lambda > 0$
$I \leftarrow \text{Int}(\lambda)$ ▷ Closest integer to λ
Compute $F(I) = \sum_{k=0}^{I} p_k$ using recursion
Generate $U \sim \text{Uniform}(0, 1)$
if $U \leq F(I)$ **then**
    $i \leftarrow I$, $F \leftarrow F(I)$
    while $U < F$ **do**
        $i \leftarrow i - 1$
        $p_i \leftarrow p_{i+1} \cdot \frac{i+1}{\lambda}$ ▷ Backward recursion
        $F \leftarrow F - p_i$
    end while
    $X \leftarrow i + 1$
else
    $i \leftarrow I + 1$, $F \leftarrow F(I)$
    while $U > F$ **do**
        $p_i \leftarrow \frac{\lambda}{i} \cdot p_{i-1}$ ▷ Forward recursion
        $F \leftarrow F + p_i$
        $i \leftarrow i + 1$
    end while
    $X \leftarrow i - 1$
end if
Output: $X \sim \text{Poisson}(\lambda)$

Comparison: Average Number of Searches for Poisson Generation

Standard Method (Start at 0)

- **Average number of searches:**

$$\mathbb{E}[\text{searches}] = 1 + \lambda$$

because on average we must sum probabilities up to the mean λ.

Efficient Method (Start near λ)

- Approximately, number of searches:

$$1 + |X - \lambda|$$

where X is the Poisson random variable generated.

- For large λ, $X \sim N(\lambda, \lambda)$ by the Central Limit Theorem.
- Expected number of searches:

$$\begin{aligned}\text{Average number of searches} \approx & 1 + E[|X - \lambda|], \quad X \sim N(\lambda, \lambda \\ &= 1 + \sqrt{\lambda} E\left[\frac{|X - \lambda|}{\sqrt{\lambda}}\right] \\ &= 1 + \sqrt{\lambda} E[|Z|] = 1 + \left(\frac{2}{\pi}\right)^{1/2} \sqrt{\lambda}\end{aligned}$$

Binomial Random Variable and Recursive Identity

- Let $X \sim \text{Binomial}(n, p)$ such that

$$P\{X = i\} = \frac{n!}{i!(n-i)!} p^i (1-p)^{n-i}, \quad i = 0, 1, \ldots, n$$

- We employ the inverse transform method using the recursive identity:

$$P\{X = i+1\} = \frac{n-i}{i+1} \frac{p}{1-p} P\{X = i\}$$

Inverse Transform Algorithm for Binomial (n, p)

1. Generate a random number $U \sim \text{Uniform}(0, 1)$.
2. Initialize: $c = \frac{p}{1-p}$, $i = 0$, $pr = (1-p)^n$, $F = pr$.
3. If $U < F$, set $X = i$ and stop.
4. Otherwise, update:

$$pr \leftarrow \frac{c(n-i)}{i+1}\, pr, \quad F \leftarrow F + pr, \quad i \leftarrow i + 1$$

5. Return to step 3.

Note that:

- i denotes the value currently under consideration,
- $pr = P\{X = i\}$ is the probability that X equals i,
- $F = F(i)$ is the cumulative probability $P\{X \leq i\}$.

Algorithm: Binomial (n, p) Random Variable via Inverse Transform

Algorithm 7 Generate $X \sim \text{Binomial}(n, p)$

Input: Integers n and probability p
Generate $U \sim \text{Uniform}(0, 1)$
Set $c \leftarrow p/(1-p)$, $i \leftarrow 0$, $pr \leftarrow (1-p)^n$, $F \leftarrow pr$
while $U \geq F$ **do**
    $pr \leftarrow \frac{c(n-i)}{i+1} pr$
    $F \leftarrow F + pr$
    $i \leftarrow i + 1$
end while
Output: $X \leftarrow i$

Next Lecture

In the next class, we will cover:

- The Acceptance-Rejection Technique
- The Composition Approach
- More Useful Algorithms and Examples

”Anyone who attempts to generate random numbers by deterministic means is, of course, living in a state of sin.” *John von Neumann*