Simulating random variables

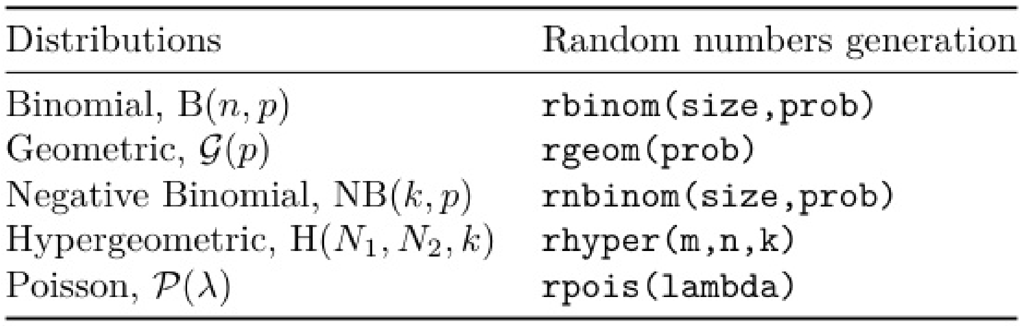
Outline

Main distributions in R

Note: In R consult

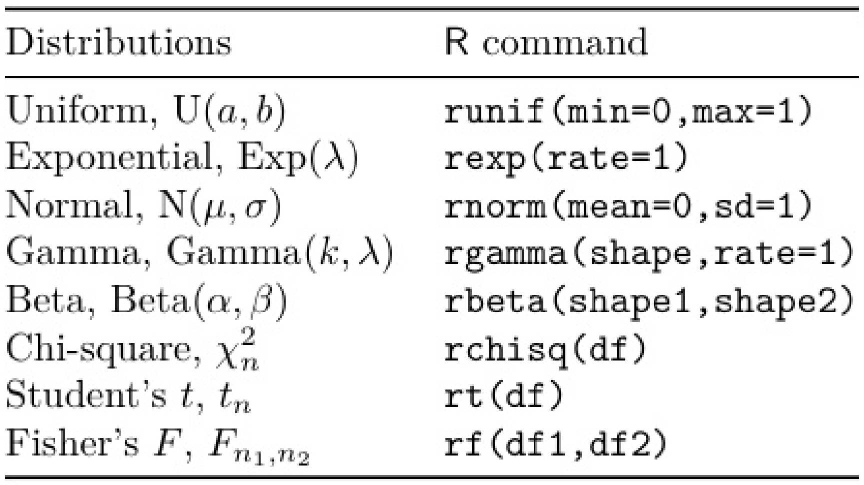
```
https://cran.r-project.org/web/views/Distributions.html
```

Main discrete distributions in R



Distributions	Random numbers generation
Binomial, $B(n, p)$	`rbinom(size,prob)`
Geometric, $\mathcal{G}(p)$	`rgeom(prob)`
Negative Binomial, $NB(k, p)$	`rnbinom(size,prob)`
Hypergeometric, $H(N_1, N_2, k)$	`rhyper(m,n,k)`
Poisson, $\mathcal{P}(\lambda)$	`rpois(lambda)`

Main continuous distributions in R



Distributions	R command
Uniform, $\mathrm{U}(a,b)$	`runif(min=0,max=1)`
Exponential, $\mathrm{Exp}(\lambda)$	`rexp(rate=1)`
Normal, $\mathrm{N}(\mu,\sigma)$	`rnorm(mean=0,sd=1)`
Gamma, $\mathrm{Gamma}(k,\lambda)$	`rgamma(shape,rate=1)`
Beta, $\mathrm{Beta}(\alpha,\beta)$	`rbeta(shape1,shape2)`
Chi-square, χ^2_n	`rchisq(df)`
Student's t, t_n	`rt(df)`
Fisher's F, F_{n_1,n_2}	`rf(df1,df2)`

Inverse transform

Let X be a random variable with a cumulative distribution function (CDF) denoted by $F(x) = P(X \leq x)$.

By definition, the CDF is a non-decreasing function. This property allows the definition of an inverse function, F^{-1}, which is referred to as the quantile function. For a given value u in the interval $[0,1]$, the quantile function is defined as:

$$F^{-1}(u) = \inf\{x : F(x) \geq u\}.$$

This definition means that $F^{-1}(u)$ is the smallest value of x such that the CDF $F(x)$ is at least u.

Next, consider a uniformly distributed random variable U, where $U \sim U(0,1)$. For a uniform random variable on the interval $[0,1]$, the property is that $P(U \leq u) = u$ for any $u \in [0,1]$.

Now, define a random variable X as $X = F^{-1}(U)$. To determine the CDF of X, we compute:

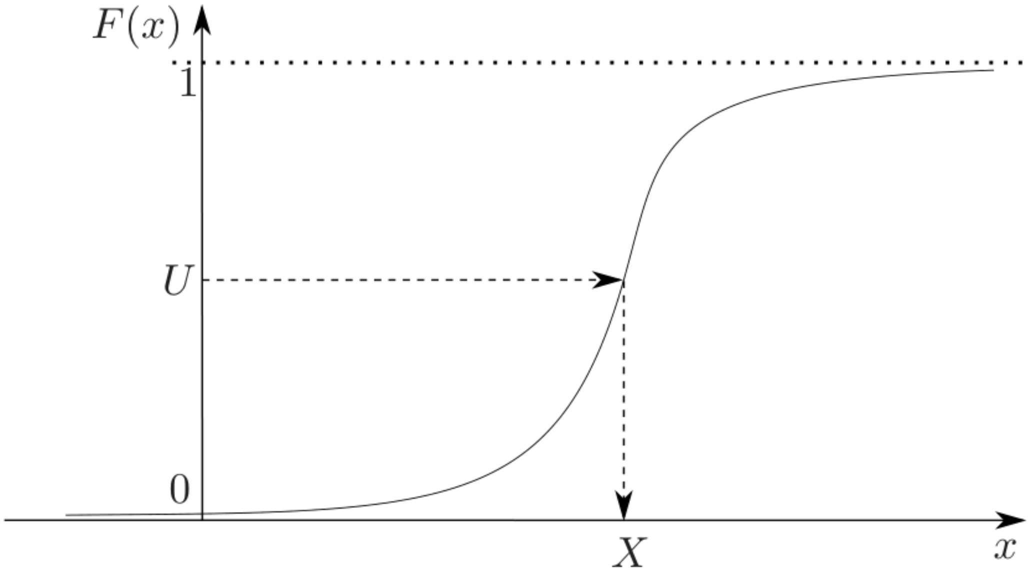
$$P(X \leq x) = P(F^{-1}(U) \leq x) = P(U \leq F(x)) = F(x).$$

This sequence of equalities demonstrates that $P(X \leq x)$ equals $F(x)$, indicating that X is distributed according to the CDF F. In other words, $X \sim F$.

This result forms the basis of the **inverse transform sampling method**, which is summarized as follows: to generate a random variable X with a given CDF F, first draw a random sample U from the uniform distribution $U(0,1)$, and then compute $X = F^{-1}(U)$.

This method can be used to generate random variables from virtually any distribution, provided that the inverse CDF can be calculated or approximated.

It is illustrated in the following figure, which visually represents the transformation from the uniform random variable U to the random variable X with the desired distribution F. The figure typically shows the CDF $F(x)$ and its inverse $F^{-1}(u)$, alongside the uniform distribution on $[0,1]$ for U.



In the figure, you would typically see a graph of the CDF $F(x)$ and the inverse function $F^{-1}(u)$. It may also show how a vertical line drawn at some value u in the uniform distribution intersects the inverse function $F^{-1}(u)$, and then how this maps to a value in the distribution of X. This visualizes the process of transforming a uniformly distributed random number into a random number from the distribution with CDF F.

Inverse transform for discrete random variables

Consider a X discrete random variable that can assume the k values $x_1, x_2, \ldots, x_k$ with probabilities. $p_1 \geq p_2 \geq \cdots \geq p_k$.

The CDF of X is

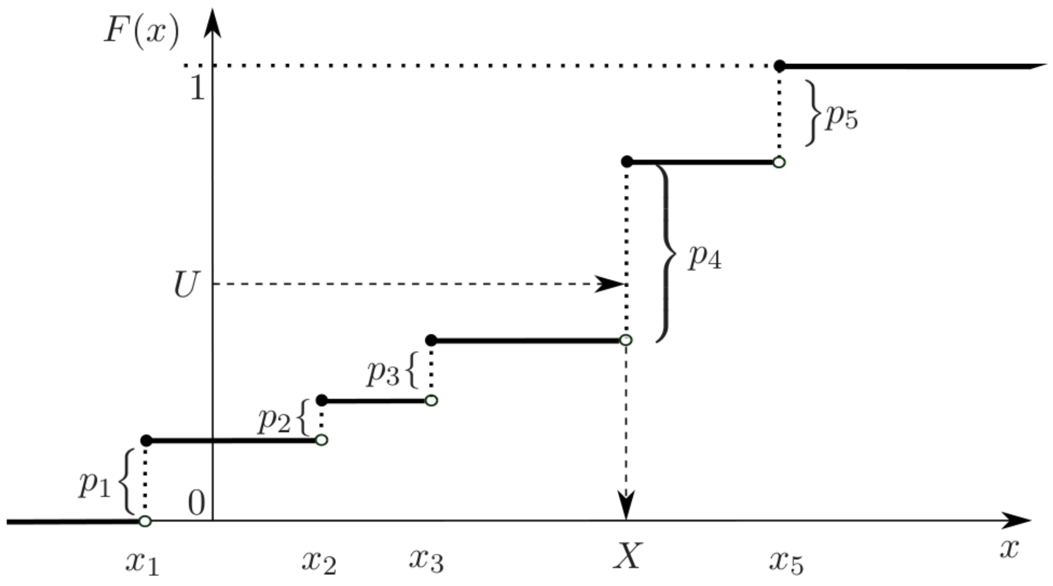
$$F(x) = \sum_{i:x_i \leq x} p_i$$

The inverse transform algorithm to obtain random values from X can be described as:

Algorithm

1. Generate a random number U in $(0,1)$.

2. Set $X = x_i$ if $F(x_{i-1}) < U \le F(x_i)$ and stop.



The values in support of X have been ordered in terms of their probabilities to gain efficiency.

Example. Inverse transform for discrete random variables

We want to simulate a rv X with probability mass function

$$
\begin{aligned}
P(X = 0) &= 0.05 \\
P(X = 1) &= 0.1 \\
P(X = 2) &= 0.45 \\
P(X = 3) &= 0.4.
\end{aligned}
$$

The PMF of X specifies the probabilities for each possible value of X. The CDF is the cumulative sum of the probabilities:

$$F(X) = P(X \le x).$$

The values are:

- $P(X \le 0) = 0.05$,
- $P(X \le 1) = 0.05 + 0.10 = 0.15$,
- $P(X \le 2) = 0.15 + 0.45 = 0.60$,

– $P(X \leq 3) = 0.60 + 0.40 = 1$.

This CDF partitions the interval $[0, 1)$ into subintervals:

– $[0, 0.05)$ corresponds to $X = 0$,

– $[0.05, 0.15)$ corresponds to $X = 1$,

– $[0.15, 0.60)$ corresponds to $X = 2$,

– $[0.60, 1)$ corresponds to $X = 3$.

A uniform random variable U in $(0, 1)$ is used to generate a value of X based on the subinterval in which U falls. The algorithm assigns X values according to these intervals:

– If $U < 0.05$, $X = 0$,

– If $0.05 \leq U < 0.15$, $X = 1$,

– If $0.15 \leq U < 0.60$, $X = 2$,

– If $0.60 \leq U < 1$, $X = 3$.

Which algorithm is more efficient?

Algorithm 1

1. Generate a random number U in $(0, 1)$.
2. If $U < 0.05$ set $X = 0$ and stop.
3. If $U < 0.15$ set $X = 1$ and stop.
4. If $U < 0.6$ set $X = 2$ and stop.
5. Set $X = 3$ and stop.

Algorithm 2

1. Generate a random number U in $(0, 1)$.

2. If $U < 0.45$ set $X = 2$ and stop.

3. If $U < 0.85$ set $X = 3$ and stop.

4. If $U < 0.95$ set $X = 1$ and stop.

5. Set $X = 0$ and stop.

Algorithm 1

```
sim.disc1 = function(MC) {
   sim.disc = rep(3, MC)
   sim.u = runif(MC)
   for(i in 1:MC){
      if (sim.u[i]<0.05) sim.disc[i] = 0
      else if (sim.u[i]<0.15) sim.disc[i] = 1
      else if (sim.u[i]<0.6) sim.disc[i] = 2
}
  return(sim.disc)
}
```

Algorithm 2

```
sim.disc2 = function(MC) {
   sim.disc = rep(0, MC)
   sim.u = runif(MC)
   for(i in 1:MC){
      if (sim.u[i]<0.45) sim.disc[i] = 2
      else if (sim.u[i]<0.85) sim.disc[i] = 3
      else if (sim.u[i]<0.95) sim.disc[i] = 1
}
   return(sim.disc)
}
```

Other alternative

The use of vectorized operations using the command `which` avoids the need for explicit loops.

```r
sim.disc3 = function(MC) {
   # The last value 3 is given by default
   sim.disc = rep(3, MC)

   sim.u = runif(MC)

   # Identify indices where the random number is less than 0.05
   zero = which(sim.u < 0.05)
   sim.disc[zero] = 0  # Set corresponding values in 'sim.disc' to 0

   # Identify indices where the random number is between 0.05 and 0.15
   one = which(sim.u > 0.05 & sim.u < 0.15)
   sim.disc[one] = 1  # Set corresponding values in 'sim.disc' to 1

   # Identify indices where the random number is between 0.15 and 0.6
   two = which(sim.u > 0.15 & sim.u < 0.6)
   sim.disc[two] = 2  # Set corresponding values in 'sim.disc' to 2

   return(sim.disc)
}
```

Alternative with Rcpp

- `#include <Rcpp.h>` This directive includes the Rcpp header file, which is necessary for Rcpp functionality.
- `using namespace Rcpp;` This line tells the compiler to use the Rcpp namespace, allowing us to use the Rcpp functions and classes without prefixing them with `Rcpp::`.
- `NumericVector`: The Rcpp equivalent of R's numeric vector. We use this for both the output `sim_disc` and the uniformly distributed random numbers `sim_u`.
- `runif(MC)` This Rcpp function generates a vector of uniformly distributed random numbers, similar to R's `runif` function.
- The statements `for` loop and `if` are similar in structure to R but follow the C++ syntax.
- The function is exported to R with the `// [[Rcpp::export]]` comment, which allows it to be called from R after compiling the C++ code.

```r
if (!require(Rcpp)) install.packages("Rcpp")
library(Rcpp)
```

```r
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
IntegerVector sim_disc3(int MC) {
    IntegerVector sim_disc(MC, 3);
    NumericVector sim_u = runif(MC);

    for(int i = 0; i < MC; ++i) {
        if(sim_u[i] < 0.05) {
            sim_disc[i] = 0;
        } else if(sim_u[i] > 0.05 && sim_u[i] < 0.15) {
            sim_disc[i] = 1;
        } else if(sim_u[i] > 0.15 && sim_u[i] < 0.6) {
            sim_disc[i] = 2;
        }
    }
    return sim_disc;
}
'
)

im_disc3(100)
```

```r
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector sim_disc2(int MC) {
    NumericVector sim_disc(MC);
    NumericVector sim_u = runif(MC);

    for(int i = 0; i < MC; i++) {
        if (sim_u[i] < 0.45) sim_disc[i] = 2;
        else if (sim_u[i] < 0.85) sim_disc[i] = 3;
        else if (sim_u[i] < 0.95) sim_disc[i] = 1;
        // No need for an else statement as sim_disc is initialized with 0s
    }

    return sim_disc;
}
'
)

sim_disc2(100)
```

```r
if (!require(microbenchmark)) install.packages("microbenchmark")
library(microbenchmark)

microbenchmark(sim.disc1(1000), sim.disc2(1000), sim.disc3(1000))
```

```
Unit: microseconds
            expr   min     lq    mean median     uq   max neval cld
 sim.disc1(1000) 117.7 121.45 132.815 122.15 123.35 387.1   100   a
 sim.disc2(1000) 101.8 104.95 108.894 106.00 107.00 161.7   100   b
 sim.disc3(1000)  57.7  62.05  92.691  63.10  84.35 495.2   100   c
```

```
microbenchmark(sim_disc2(1000), sim_disc3(1000))
```

```
Unit: microseconds
            expr  min   lq   mean median    uq  max neval cld
 sim_disc2(1000) 12.1 14.2 18.576  14.50 15.95 75.7   100   a
 sim_disc3(1000) 11.6 12.7 14.638  13.05 13.55 57.0   100   b
```

Goodness of fit

We can use the chi-square goodness-of-fit test to check that any of the algorithms that we have written actually samples from the desired distribution.

```
MC = 1000
set.seed(1)

sim.disc = sim.disc3(MC)
chisq.test(x=table(sim.disc), p=c(0.05,0.1,0.45,0.4))
```

```
    Chi-squared test for given probabilities

data:  table(sim.disc)
X-squared = 1.5389, df = 3, p-value = 0.6733
```

Discrete uniform random variables

Consider X discrete rv with $P(X = r) = \dfrac{1}{k}$ for $r \in \{1, 2, \dots, k\}$.

$$F_X(x) = \begin{cases} 0 & \text{if} \quad x < 1 \\ \frac{r}{k} & \text{if} \quad r \le x < r+1, \quad i \in \{1, 2, \dots, k-1\} \\ 1 & \text{if} \quad x \ge k \end{cases}$$

$$F_X^{-1}(u) = x \quad \text{if} \quad \frac{r-1}{k} < u \le \frac{r}{k}$$

or equivalently (*ceiling*)

$$F_X^{-1}(u) = \lfloor x \rfloor + 1$$

The symbol $\lfloor x \rfloor$ represents the floor function, which returns the greatest integer less than or equal to x. When the equation includes $\lfloor x \rfloor + 1$, it represents the ceiling function, which returns the smallest integer greater than or equal to x.

Then

Algorithm

1. Generate a random number U in $(0,1)$.

2. Return $X = \lfloor U \cdot k \rfloor + 1$ (or equiv. $X = \lceil U \cdot k \rceil$).

```r
MC = 1000
set.seed(2)
k = 5

sim.dunif = ceiling(runif(MC)*k)
chisq.test(x=table(sim.dunif), p=rep(1,k)/k)
```

```
   Chi-squared test for given probabilities

data:  table(sim.dunif)
X-squared = 5.71, df = 4, p-value = 0.2219
```

Example: Inverse transform for a geometric random variable

Consider $X \sim G(p)$.

The PMF is given as:

$$P(X = r) = p(1-p)^r, \quad r \in \{0, 1, 2, \dots\}.$$

The geometric random variable X describes the number of failures before the first success in a sequence of Bernoulli trials with success probability p.

- Case 1: For $x < 0$, $P(X \le x) = 0$, because X can only take non-negative integer values.

- Case 2: For $r \le x < r+1$, where r is an integer.

  The CDF $F_X(r)$ is the cumulative probability up to r, i.e.,

$$F_X(r) = P(X \le r) = \sum_{k=0}^{r} P(X = k).$$

  Substituting the PMF:

$$F_X(r) = \sum_{k=0}^{r} p(1-p)^k.$$

  The constant p can be factored out of the summation:

$$F_X(r) = p \sum_{k=0}^{r} (1-p)^k.$$

The term $(1-p)^k$ represents a geometric series. The sum of the first $n+1$ terms of a geometric series with ratio $(1-p)$ is:

$$\sum_{k=0}^{r}(1-p)^k = \frac{1-(1-p)^{r+1}}{1-(1-p)}.$$

Simplify the denominator:

$$1-(1-p) = p.$$

Thus:

$$\sum_{k=0}^{r}(1-p)^k = \frac{1-(1-p)^{r+1}}{p}.$$

Substitute this result back into the formula for $F_X(r)$:

$$F_X(r) = p \cdot \frac{1-(1-p)^{r+1}}{p}.$$

The p cancels out, this summation results in:

$$F_X(r) = 1-(1-p)^{r+1}.$$

The CDF is:

$$F_X(x) = \begin{cases} 0 & \text{if } x < 0, \\ 1-(1-p)^{r+1} & \text{if } r \le x < r+1, \quad r \in \{0,1,2,\ldots\}. \end{cases}$$

The inverse CDF, $F_X^{-1}(u)$, maps a uniform random variable $U \sim U(0,1)$ to a value of X such that $F_X(X) \le u < F_X(X+1)$.

Given the CDF:

$$F_X(r) = 1-(1-p)^{r+1},$$

the inverse can be found by solving for r in terms of u:

$$u = 1-(1-p)^{r+1}.$$

Rearranging:

$$(1-p)^{r+1} = 1-u.$$

Taking the logarithm:

$$r+1 = \frac{\log(1-u)}{\log(1-p)}.$$

Therefore:

$$r = \left\lfloor \frac{\log(1-u)}{\log(1-p)} \right\rfloor.$$

Then the inverse CDF is:

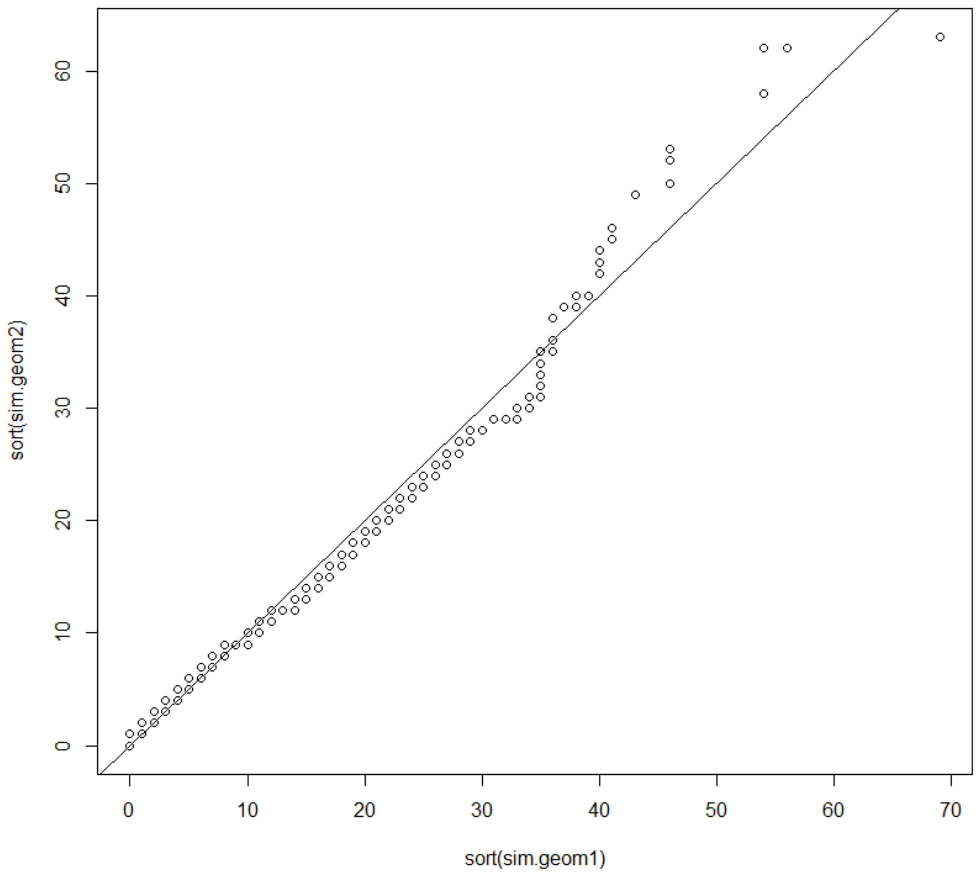
$$F_X^{-1}(u) = \left\lfloor \frac{\log(1-u)}{\log(1-p)} \right\rfloor.$$

Algorithm

1. Generate a random number U in $(0,1)$.

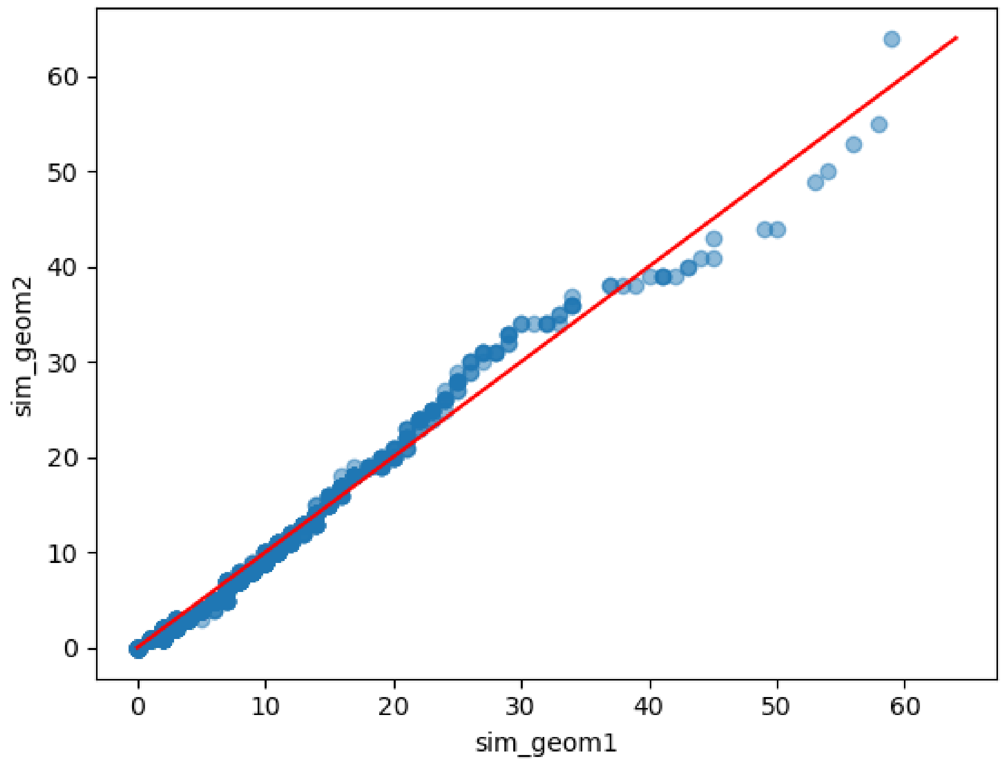
2. Set $X = \lfloor \log(U)/\log(1-p) \rfloor$ and stop.

```
MC = 1000
set.seed(2)
p = 0.1

sim.geom1 = floor(log(runif(MC))/log(1-p))
sim.geom2 = rgeom(MC,prob=p)

plot(sort(sim.geom1),sort(sim.geom2))
abline(0,1)
```



In Python

```
import numpy as np
import matplotlib.pyplot as plt

# Set parameters
MC = 1000
p = 0.1
np.random.seed(2)

# Simulate geometric distributions
sim_geom1 = np.floor(np.log(np.random.uniform(size=MC)) / np.log(1 - p)).astype(
                                                int)
sim_geom2 = np.random.geometric(p, MC) - 1
# NumPy's geometric starts at 1

# Plotting
plt.scatter(np.sort(sim_geom1), np.sort(sim_geom2), alpha=0.5)
plt.plot([0, max(sim_geom1.max(), sim_geom2.max())], [0, max(sim_geom1.max(),
                                                sim_geom2.max())], 'r')
# abline equivalent
plt.xlabel('sim_geom1')
plt.ylabel('sim_geom2')
plt.show()
```



Inverse transform for continuous random variables

Consider X continuous rv with CDF F and quantile function F^{-1}. The inverse transform algorithm to obtain values from X at random is

Algorithm

1. Generate a random number U in $(0, 1)$.
2. Set $X = F^{-1}(U)$ and stop.

Example: exponential distribution

Let $X \sim \exp(\lambda)$.

Its CDF is $F(x) = 1 - e^{-\lambda x}$ for $x > 0$.

$$\begin{aligned}
u &= F(x) \Leftrightarrow \\
u &= 1 - e^{-\lambda x} \Leftrightarrow \\
1 - u &= e^{-\lambda x} \Leftrightarrow \\
-\log(1-u) &= \lambda x \Leftrightarrow \\
\\
x &= \frac{-\log(1-u)}{\lambda}.
\end{aligned}$$

Algorithm

1. Generate a random number U in $(0, 1)$.
2. Set $X = -\log(1-U)/\lambda$ (or $X = -\log(U)/\lambda$) and stop.

```r
MC = 1000
set.seed(1)

sim.exp = -log(runif(MC))
ks.test(sim.exp, "pexp", rate=1)
```

```
	Asymptotic one-sample Kolmogorov-Smirnov test

data:  sim.exp
D = 0.024366, p-value = 0.5928
alternative hypothesis: two-sided
```

Aceptance-Rejection

The Acceptance-Rejection method, also known as the Rejection Sampling method, is a technique used in random variate generation, especially in simulation and Monte Carlo methods. It is used when direct sampling from the desired distribution is difficult, but sampling from a simpler distribution is feasible.

The method involves generating random samples from a proposal distribution and accepting or rejecting them based on a certain criterion so that the accepted samples follow the target distribution.

The key is to ensure that the proposal distribution, scaled by a constant c, **bounds** the target distribution of above.

The Method

1. **Choose a Proposal Distribution**: Select a distribution of proposals $g(x)$ that is easy to sample. This distribution should cover the support of the target distribution $f(x)$.

2. **Scaling the Proposal Distribution**: Find a constant c such that $c \cdot g(x)$ is always greater than or equal to $f(x)$ for all x. Mathematically, $c \cdot g(x) \geq f(x)$ for all x.

3. **Sampling and Acceptance Criterion**:

   - Generate a sample X from the proposal distribution $g(x)$.
   - Generate a uniform random variable U in the interval [0, 1].
   - Accept the sample X if $U \leq \frac{f(X)}{c \cdot g(X)}$; otherwise, reject it.

4. **Repeat**: Continue this process until the desired number of samples from the target distribution is obtained.

Proof:

We will prove that, conditional on acceptance, the random variable X follows the target distribution $f(x)$. To do this, we analyze the probability $P(X \leq x \mid A)$, where A represents the acceptance event.

Notations

- X: a random variable with probability density function (pdf) $g(x)$, denoted as $X \sim g(x)$.
- U: a random variable uniformly distributed on $[0, 1]$, i.e., $U \sim \text{Uniform}[0, 1]$. Hence,

$$P(U \leq u) = u, \quad \text{for } 0 \leq u \leq 1.$$

- A: the acceptance event, defined as

$$A = \Big\{U \leq \frac{f(X)}{c\, g(X)}\Big\},$$

  where $c > 0$ ensures $f(x) \leq c\, g(x)$ for all x, so $\frac{f(x)}{c\, g(x)} \leq 1$.

- X and U are independent.

1. Joint distribution of (X, U)

Since X and U are independent, the joint pdf is the product of their marginals:

$$f_{X,U}(x, u) = g(x) \cdot 1 = g(x), \quad \text{for } u \in [0, 1].$$

Outside $u \in [0, 1]$, the joint pdf is zero.

2. Conditional probability of acceptance

Given $X = x$, the acceptance event A depends only on U. Since U is uniform on $[0, 1]$:

$$P(A \mid X = x) = P\Big(U \leq \frac{f(x)}{c\, g(x)}\Big) = \frac{f(x)}{c\, g(x)}, \quad \text{if } \frac{f(x)}{c\, g(x)} \leq 1.$$

3. Unconditional probability of acceptance

By the law of total probability:

$$P(A) = \int_{-\infty}^{\infty} P(A \mid X = x)\, g(x)\, dx.$$

Substituting $P(A \mid X = x) = \frac{f(x)}{c\,g(x)}$:

$$P(A) = \int_{-\infty}^{\infty} \frac{f(x)}{c\,g(x)}\, g(x)\, dx = \int_{-\infty}^{\infty} \frac{f(x)}{c}\, dx = \frac{1}{c}\int_{-\infty}^{\infty} f(x)\, dx.$$

Since $f(x)$ is a pdf, $\int_{-\infty}^{\infty} f(x)\, dx = 1$, so $P(A) = \frac{1}{c}$.

4. Distribution of accepted samples

To show that the accepted samples follow $f(x)$, we compute $P(X \le x \mid A)$. By definition:

$$P(X \le x \mid A) = \frac{P(X \le x, A)}{P(A)}.$$

First, compute $P(X \le x, A)$:

$$P(X \le x, A) = \int_{-\infty}^{x} P(X = t, A)\, dt$$

where we integrate over all possible values of $t \le x$.

The joint probability $P(X = t, A)$ can be decomposed using the definition of conditional probability:

$$P(X = t, A) = P(A \mid X = t)\, P(X = t).$$

In this case, $P(X = t)$ is represented by the pdf $g(t)$ (since $X \sim g(x)$) and $P(A \mid X = t)$ is the probability of acceptance given $X = t$. Therefore:

$$P(X = t, A) = P(A \mid X = t)\, g(t).$$

To find the total joint probability of the events $X \le x$ and A, sum (integrate) the joint probabilities over all values of t from $-\infty$ to x:

$$P(X \le x, A) = \int_{-\infty}^{x} P(A \mid X = t)\, g(t)\, dt = \int_{-\infty}^{x} \frac{f(t)}{c\,g(t)}\, g(t)\, dt = \int_{-\infty}^{x} \frac{f(t)}{c}\, dt.$$

Then, substitute into the conditional probability:

$$P(X \le x \mid A) = \frac{\int_{-\infty}^{x} \frac{f(t)}{c}\, dt}{\int_{-\infty}^{\infty} \frac{f(t)}{c}\, dt} = \frac{\int_{-\infty}^{x} f(t)\, dt}{\int_{-\infty}^{\infty} f(t)\, dt}.$$

Since $\int_{-\infty}^{\infty} f(t)\, dt = 1$:

$$P(X \le x \mid A) = \int_{-\infty}^{x} f(t)\, dt,$$

which is the cumulative distribution function of $f(x)$. Therefore, the pdf of X given acceptance is $f(x)$.

Sometimes, to understand the result more directly, when a sample X is accepted, its density is proportional to the target density $f(x)$, scaled by the constant c:

$$p_{\text{accepted}}(x) = \frac{f(x)}{c}.$$

Here, the factor $\frac{1}{c}$ arises from the scaling in the Acceptance-Rejection method.

For $p_{\text{accepted}}(x)$ to be a valid probability density function (pdf), its integral over the full support must equal 1. Lets check this:

$$\int_{-\infty}^{\infty} p_{\text{accepted}}(x)\, dx = \int_{-\infty}^{\infty} \frac{f(x)}{c}\, dx.$$

Since $\int_{-\infty}^{\infty} f(x)\, dx = 1$ (because $f(x)$ is a pdf), this simplifies to:

$$\int_{-\infty}^{\infty} p_{\text{accepted}}(x)\, dx = \frac{1}{c} \cdot 1 = \frac{1}{c}.$$

The total probability of accepting a sample, $P(A)$, equals the integral of $p_{\text{accepted}}(x) = \frac{1}{c}$ over all x: Thus, only a fraction $\frac{1}{c}$ of the drawn samples are accepted on average.

When conditioning on the event A (acceptance), the density of the accepted samples must be normalized by $P(A)$, ensuring it integrates to 1. The conditional density becomes:

$$\frac{p_{\text{accepted}}(x)}{P(A)} = \frac{\frac{f(x)}{c}}{\frac{1}{c}} = f(x).$$

This normalization cancels out the $\frac{1}{c}$ factor, leaving us with $f(x)$, the target density.

The Acceptance-Rejection method provides a convenient procedure to generate samples from a complicated target distribution $f(x)$ by comparing it with a simpler proposal distribution $g(x)$ scaled by a constant c. Once a sample X is drawn from g, it is accepted with probability $\frac{f(X)}{c\,g(X)}$. By analyzing the joint distribution of (X, U) and conditional probabilities, we see that *the distribution of X given that it has been accepted is precisely* $f(x)$. This establishes the correctness of the method.

5. Summary

1. Draw $X \sim g(x)$ and $U \sim \text{Uniform}[0, 1]$, independently.

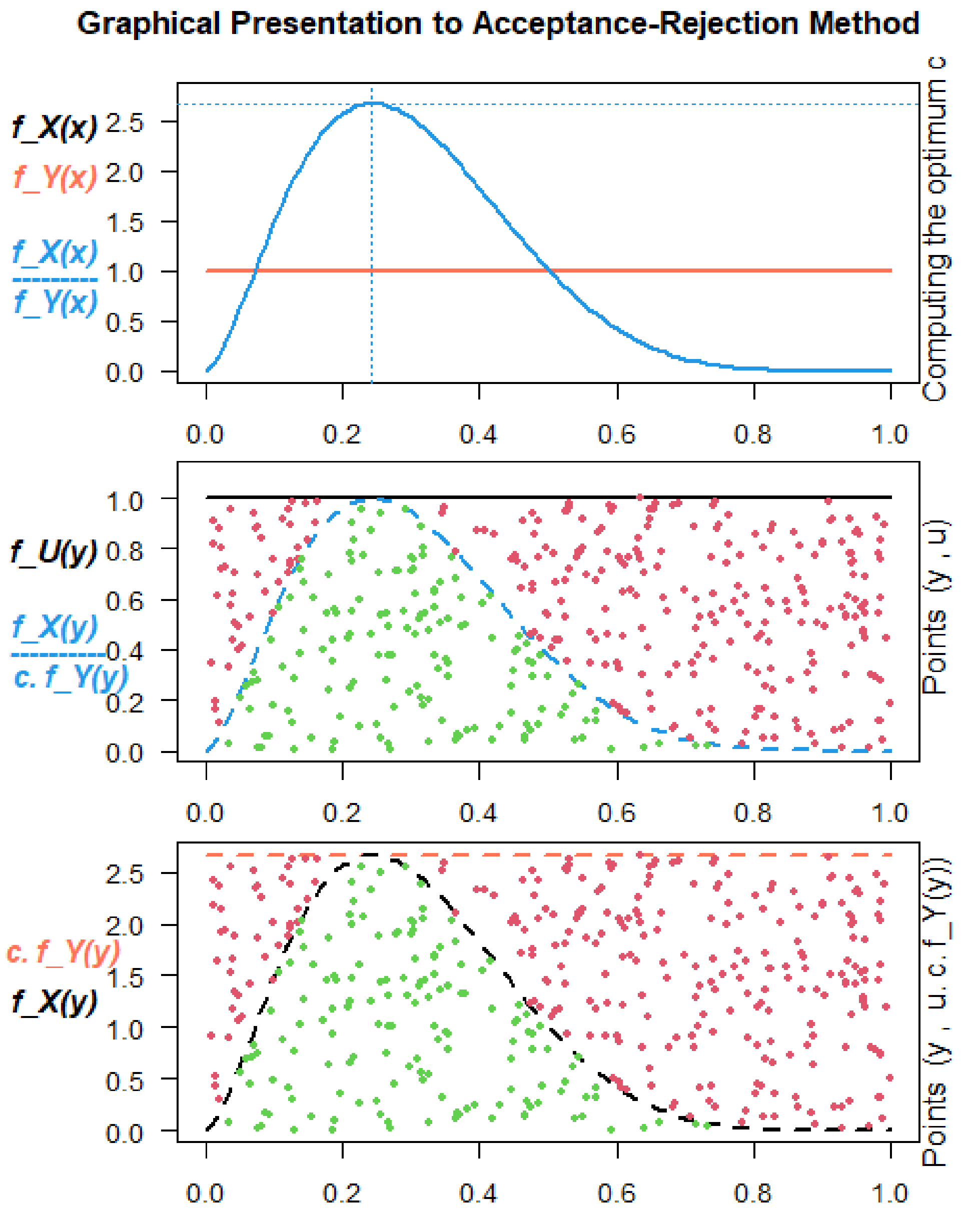
2. Accept X if $U \le \frac{f(X)}{c\,g(X)}$.

3. The unconditional acceptance probability is $P(A) = \frac{1}{c}$.

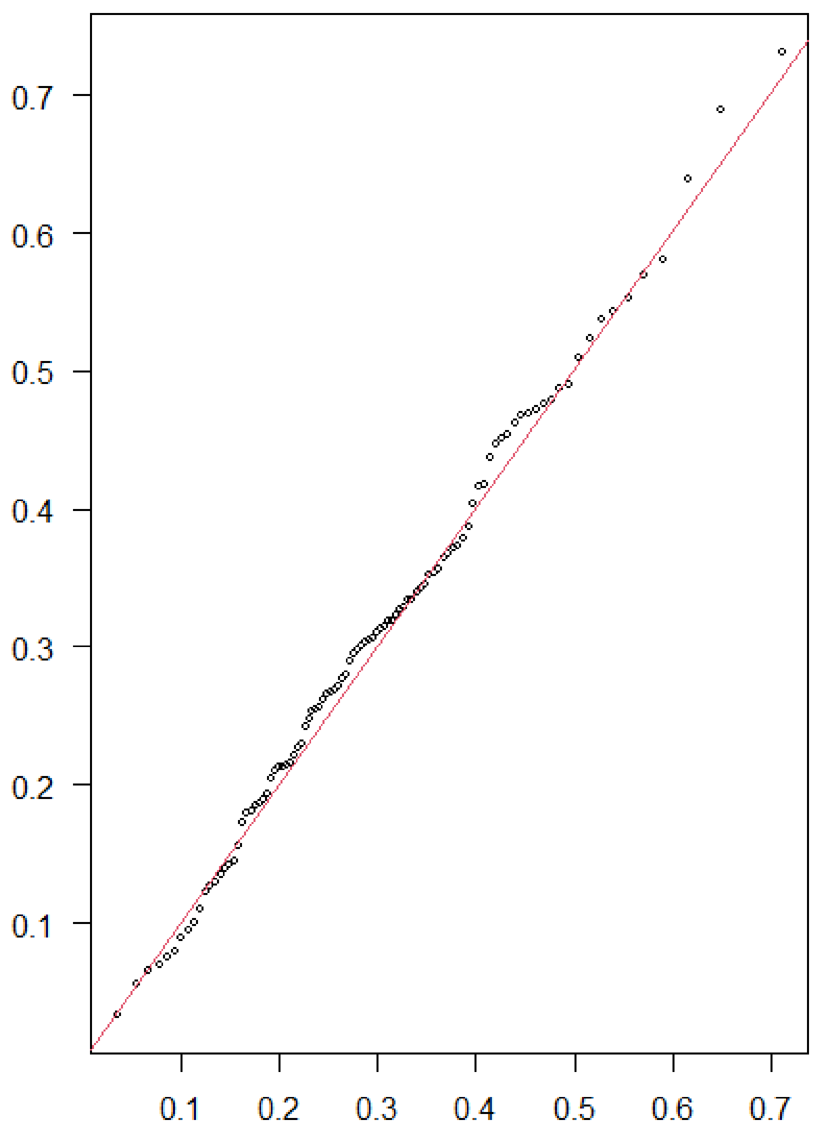
4. The accepted samples follow the target distribution $f(x)$.

```
library(AR)

data = AR.Sim(n = 150,
f_X = function(y){dbeta(y,2.7,6.3)},
Y.dist = "unif", Y.dist.par = c(0,1),
Rej.Num = TRUE, Rej.Rate = TRUE, Acc.Rate = TRUE)
```

```
Optimal c = 2.67
The numbers of Rejections = 224
Ratio of Rejections = 0.599
Ratio of Acceptance = 0.401
```



```
# QQ-plot
q = qbeta(ppoints(100), 2.7, 6.3)
qqplot(q, data, cex=0.6, xlab="Quantiles of Beta(2.7,6.3)",
ylab = "Empirical Quantiles of simulated data")
abline(0, 1, col=2)
```



Example: $|Z|$ with $Z \sim N(0,1)$

The density function of $|Z|$ with $Z \sim N(0,1)$ is

$$f(x) = \frac{2}{\sqrt{2\pi}} \exp\left[-\frac{x^2}{2}\right],$$

for $x > 0$, while the density function of an Exponential random variable with $\lambda = 1$ is $g(x) = e^{-x}$ for $x > 0$.

Then,

$$\frac{f(x)}{g(x)} = \frac{\frac{2}{\sqrt{2\pi}} \exp\left[-\frac{x^2}{2}\right]}{e^{-x}} = \sqrt{\frac{2}{\pi}} \exp\left[x - \frac{x^2}{2}\right] \le \sqrt{\frac{2e}{\pi}}$$

Set

$$c = \sqrt{\frac{2e}{\pi}}$$

and

$$\frac{f(x)}{cg(x)} = \exp\left[x - \frac{x^2}{2} - \frac{1}{2}\right] = \exp\left[\frac{(x-1)^2}{2}\right]$$

Algorithm

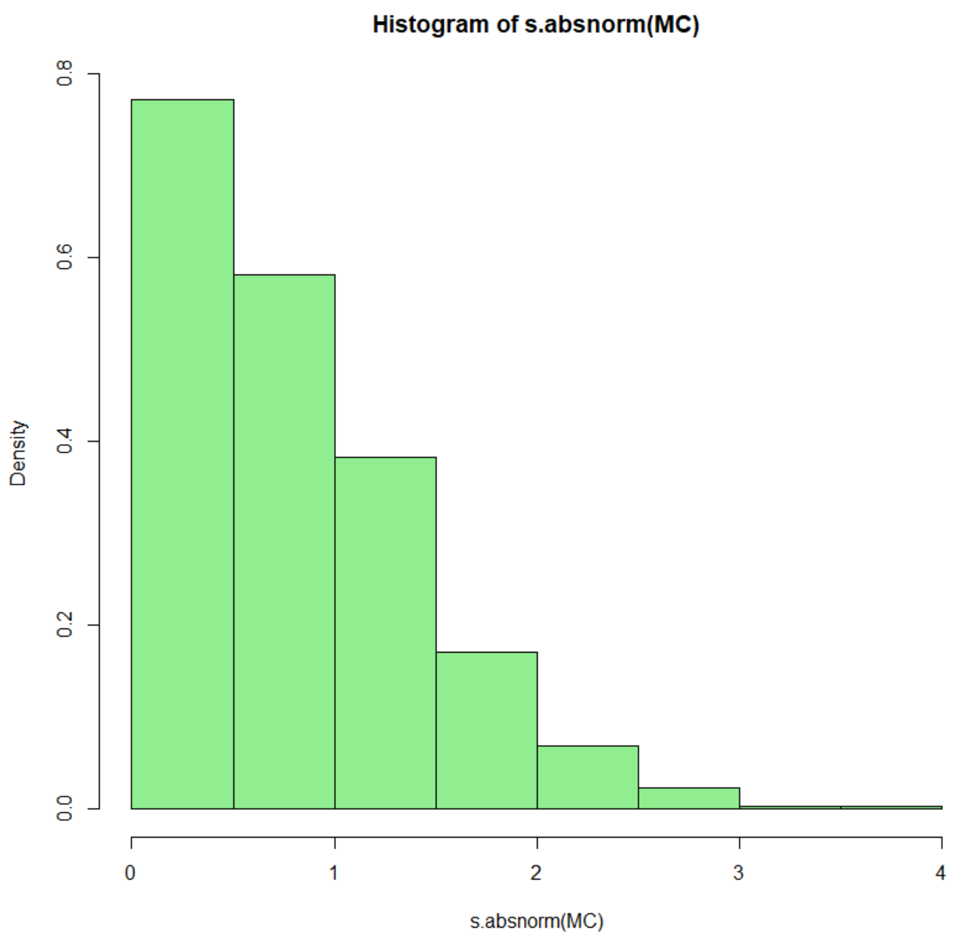
1. Generate Y with density (mass) g.
2. Generate a random number U in $(0, 1)$.
3. If $U < \exp\{-(Y-1)^2/2\}$ set $X = Y$ and stop.

   Otherwise, return to Step 1.

```
s.absnorm = function(n) {
   y = vector(length=n)
   for(i in 1:n){
      y[i] = -log(runif(1))
      while (runif(1) > exp(-(y[i]-1)^2/2)) y[i] = -log(runif(1))
   }
return(y)
}

MC = 1000
set.seed(1)

hist(s.absnorm(MC), probability=T, col="lightgreen")
```

Comments about the `s.absnorm` function:

- `y[i] = -log(runif(1))`: Generate a random number using the exponential distribution. This is done by taking the negative logarithm of a uniformly distributed random number. The exponential distribution is used here as the proposal distribution for the rejection sampling.
- `while (runif(1) > exp(-(y[i]-1)^2/2))`: This is the rejection step. The algorithm repeatedly samples the exponential distribution until it finds a value that satisfies the condition. The condition is based on the density function of the half-normal distribution. Essentially, this step ensures that the sampled values follow the target half-normal distribution. Once a value passes the rejection condition, it is stored in `y[i]`.



Program in Rcpp:

```
library(Rcpp)

sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector s_absnorm(int n) {
  NumericVector y(n);
  for(int i = 0; i < n; ++i) {
    y[i] = -log(R::runif(0, 1));
    while (R::runif(0, 1) > exp(-pow(y[i] - 1, 2) / 2)) {
      y[i] = -log(R::runif(0, 1));
    }
  }
  return y;
}
'
)

s_absnorm(100)
```

Rejection for truncated models

A zero trucated Poisson random variable X with parameter λ has pmf

$$p(r) = P(X = r) = \frac{e^{-\lambda}}{1 - e^{-\lambda}} \frac{\lambda^r}{r!}, \quad r \in \{1, 2, \ldots\}$$

Set q equal to the pmf of a Pois(λ) random variable and $M = (1 - e^{-\lambda})^{-1}$, then

$$\frac{p(r)}{Mq(r)} = 1$$

(and accepted) if $r \geq 1$, and

$$\frac{p(0)}{Mq(0)} = 0$$

(and rejected).

Algorithm

1. Generate Y a Poisson rv.
2. If $Y \geq 1$ set $X = Y$ and stop.

   Otherwise, return to Step 1.

The Box-Muller Method for Simulating Normal Distribution

The Box-Muller method is the standard procedure used to generate pairs of independent, standard, normally distributed random variates from a source of uniformly distributed random numbers.

1. **Start with Uniform Random Numbers**: Generate two independent random numbers, U_1 and U_2, from a uniform distribution in $(0,1)$.

2. **Compute Radius and Angle**: Transform U_1 and U_2 into polar coordinates with the following steps:

   - Compute $R^2 = -2\ln(U_1)$. This represents the squared radius in the polar coordinates.
   - Compute $\theta = 2\pi U_2$. This represents the angle and ensures that the points are uniformly distributed around the circle.

3. **Transform to Normal Variables**: Use the radius and angle to generate two normal distributed random variables through the following transformations:

$$\begin{aligned} Z_0 &= R\cos(\theta) = \sqrt{-2\ln(U_1)}\cos(2\pi U_2), \\ Z_1 &= R\sin(\theta) = \sqrt{-2\ln(U_1)}\sin(2\pi U_2). \end{aligned}$$

4. **Resulting Normal Variables**: The variables Z_0 and Z_1 are independent and normally distributed with mean 0 and variance 1, that is, they follow the standard normal distribution $N(0,1)$.

5. Repeat the process starting from step 1 with new pairs of uniform random numbers.

The Box-Muller method efficiently transforms uniformly distributed random numbers into normally distributed numbers, making it invaluable in computational simulations.

Proof Given U_1 and U_2, we define new variables $R^2 = -2\ln(U_1)$ and $\Theta = 2\pi U_2$. These are polar coordinates, which are then transformed into Cartesian coordinates Z_0 and Z_1.

Jacobian of the Transformation The Jacobian determinant is crucial in converting the probability density function from one set of variables to another. For the Box-Muller transformation, we calculate the determinant of the Jacobian matrix as follows:

$$J = \begin{vmatrix} \frac{\partial Z_0}{\partial U_1} & \frac{\partial Z_0}{\partial U_2} \\ \frac{\partial Z_1}{\partial U_1} & \frac{\partial Z_1}{\partial U_2} \end{vmatrix} = \frac{\partial Z_0}{\partial U_1}\frac{\partial Z_1}{\partial U_2} - \frac{\partial Z_0}{\partial U_2}\frac{\partial Z_1}{\partial U_1} = \frac{1}{R}.$$

Upon substitution and simplification, this leads to an expression that relates the change in area in the (U_1, U_2) space to the (Z_0, Z_1) space.

Derivation of the Probability Density Function The joint pdf of U_1 and U_2 is uniform, given by $f_{U_1,U_2}(u_1, u_2) = 1$ for $0 < u_1, u_2 < 1$. Through the transformation, we search for the joint PDF of Z_0 and Z_1. The transformation of variables in probability theory provides that:

$$\begin{aligned} f_{Z_0,Z_1}(z_0, z_1) &= f_{U_1,U_2}(u_1, u_2)|J| \\ &= \frac{1}{2\pi}e^{-\frac{z_0^2+z_1^2}{2}}, \end{aligned}$$

indicating that Z_0 and Z_1 have a joint distribution that is the product of two independent standard normal distributions.

Composition approach

If we need to simulate a random variable X with a mixture distribution either finite (or countable)

$$F(x) = \sum_{i \in I} p_i F_i$$

or continuous

$$F(x) = \int_A \omega(a) F_a(x) da$$

we can simulate from F_i (F_a) and a random variable with pmf $(p_i)_{i \in I}$ or density function ω.

Then:

Algorithm

1. Generate Y with pmf $(p_i)_{i \in I}$ or density function ω.
2. Generate X with CDF F_Y and stop.

Composition Approach for Simulating Mixture Distributions

When we need to simulate a random variable X that follows a mixture distribution, we can encounter two main types: a finite (or countable) mixture distribution and a continuous mixture distribution. The mixture distribution combines several distributions, each weighted according to its contribution to the overall mixture.

Finite or Countable Mixture Distribution

For a finite or countable mixture distribution, the cumulative distribution function (CDF) is given by:

$$F(x) = \sum_{i \in I} p_i F_i(x)$$

Here, $F(x)$ is the overall CDF of the mixture distribution, $F_i(x)$ represents the CDF of the distribution of components i in the mixture and p_i is the probability mass function (pmf) that weights the contribution of each distribution of components. The index set I represents the set of all the distributions of components in the mixture.

Continuous Mixture Distribution

In the case of a continuous mixture distribution, the CDF is defined as:

$$F(x) = \int_A \omega(a) F_a(x) \, da$$

In this equation, $\omega(a)$ is a density function on a set A that determines the weight of each distribution of components $F_a(x)$ in the mixture.

Algorithm for Simulation

To simulate a random variable X from a mixture distribution, whether finite/countable or continuous, the following algorithm can be used:

1. First, generate a random variable Y. For a finite or countable mixture distribution, Y should be generated with a probability mass function (pmf) $(p_i)_{i \in I}$. For a continuous mixture distribution, generate Y with a density function ω. This step effectively selects which component distribution F_Y will be used to simulate X.

2. Next, generate the random variable X using the cumulative distribution function (CDF) F_Y of the selected component distribution. This step simulates a value from the chosen component distribution.

Following this algorithm, we can accurately simulate values from a mixture distribution, which is essential in various applications such as statistical modeling, risk assessment, and machine learning.

Example: Generating a Laplace Random Variable

The Laplace distribution, also known as the double exponential distribution, is a continuous probability distribution characterized by its location parameter $m \in \mathbb{R}$ and a positive rate parameter $\lambda > 0$. The probability density function (pdf) of the Laplace distribution is given by:

$$f(x) = \frac{\lambda}{2} \exp\left[-\lambda|x - m|\right],$$

where x is a real number. This distribution is symmetrical around its mean, which is the location parameter m. The rate parameter λ controls the spread of the distribution, with higher values of λ leading to a steeper peak and thinner tails.

The Laplace distribution can be conceptualized as a mixture of two exponential distributions: one with parameter λ and the other with parameter $-\lambda$. Both distributions are weighted equally and shifted by m units. This perspective helps to understand the symmetry and shape of the Laplace distribution.

Algorithm for Generating a Laplace Random Variable To generate a random variable that follows the Laplace distribution, we use the following algorithm, which involves generating two independent uniform random numbers and then transforming them appropriately.

1. Generate two independent random numbers U_1 and U_2, each uniformly distributed in the interval $(0, 1)$.

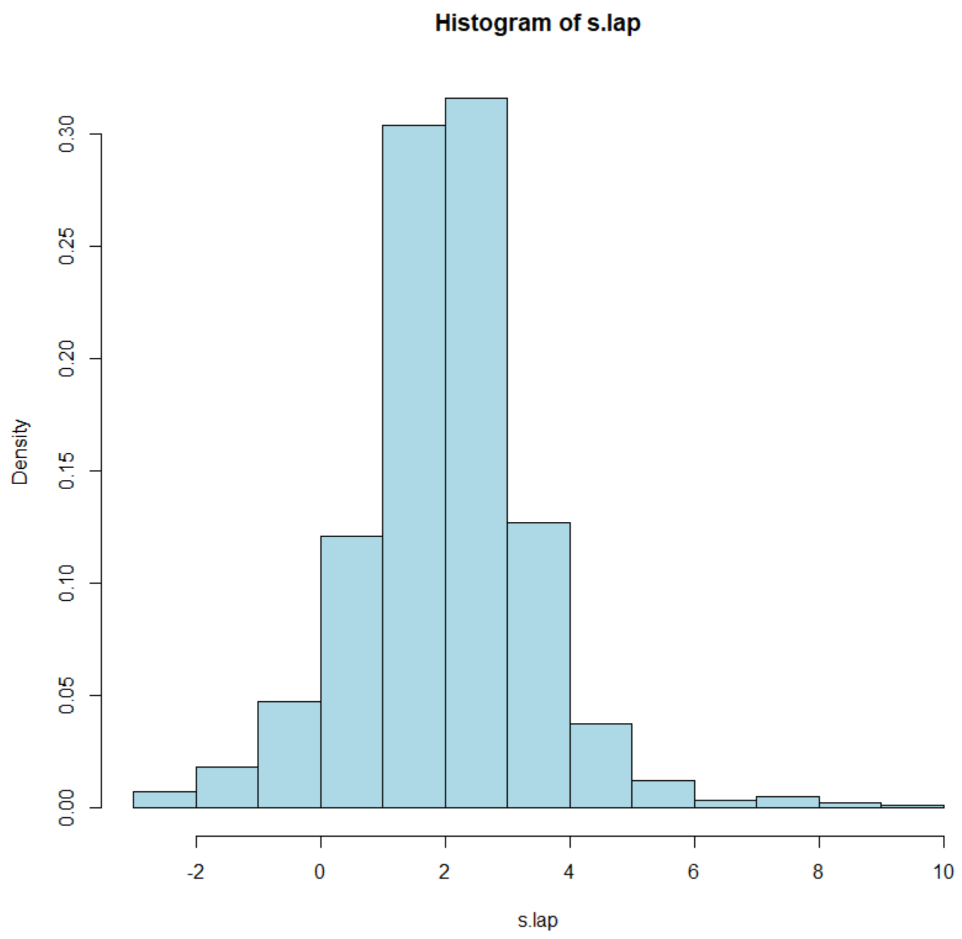
2. Return $X = m + \dfrac{\operatorname{sign}(U_1 - 0.5)\log(U_2)}{\lambda}$.

In this algorithm, the term $\operatorname{sign}(U_1 - 0.5)$ determines the sign of the output (positive or negative) based on whether U_1 is greater or less than 0.5. This transformation is crucial because it creates a symmetric split around 0.

The function **sign()** then assigns a $+1$ or -1 based on whether $U_1 - 0.5$ is positive or negative, respectively. This means that there is a 50% chance of getting a positive sign and a 50% chance of getting a negative sign, reflecting the symmetric nature of the Laplace distribution.

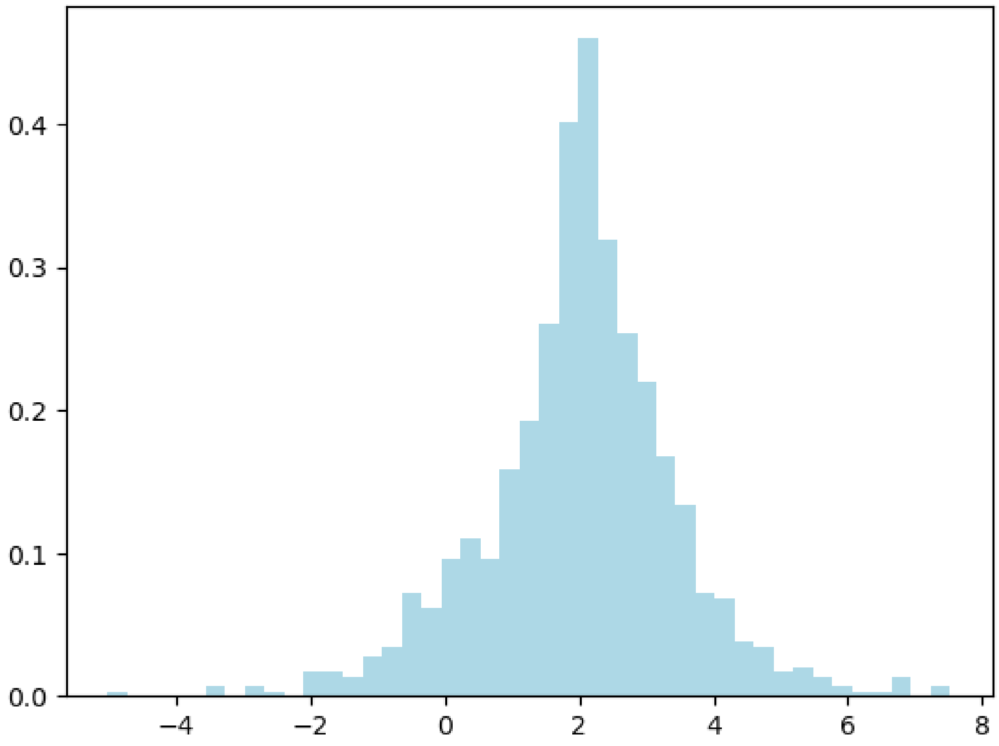
$\log(U_2)$: The logarithm of a uniformly distributed random variable follows an exponential distribution. This step is crucial to achieving the exponential behavior in the tails of the Laplace distribution.

```
MC = 1000
set.seed(2)

m = 2
lmbd = 1

s.lap = m + sign(runif(MC)-0.5)*log(runif(MC))/lmbd

hist(s.lap, prob=T, col="lightblue")
```



In Python

```
import numpy as np
import matplotlib.pyplot as plt

# Setting the seed for reproducibility
np.random.seed(2)

# Parameters
MC = 1000
m = 2
lmbd = 1

# Generating samples from a Laplace distribution
s_lap = m + np.sign(np.random.uniform(0, 1, MC) - 0.5) * \
np.log(np.random.uniform(0, 1, MC)) / lmbd

# Plotting the histogram
plt.hist(s_lap, bins='auto', density=True, color='lightblue')
plt.show()
```



Example: Normal random variable (rejection + composition)

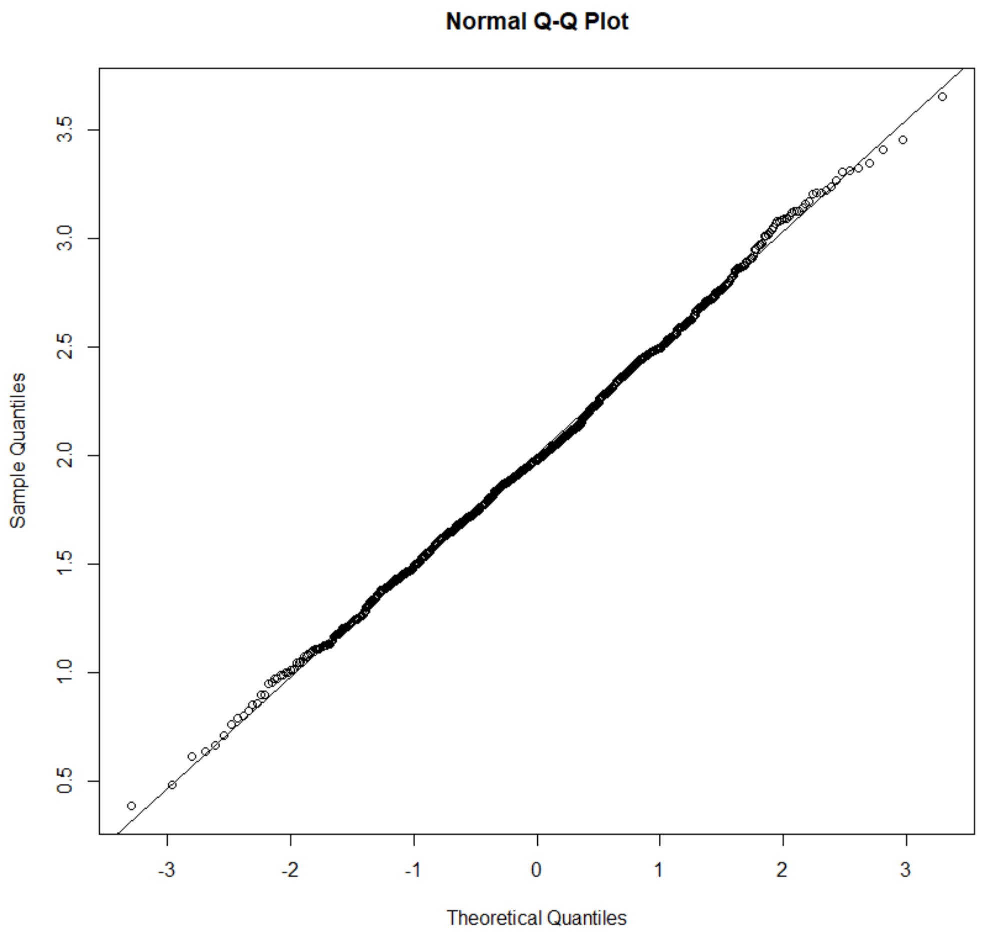
The algorithm generates random numbers from a normal distribution by first generating random numbers from a half-normal distribution (or the absolute value of a normal distribution), flipping their sign randomly (to simulate the symmetric property of the normal distribution), and then scaling and shifting these values to match the desired mean and standard deviation.

The `s.absnorm` function is implemented using a rejection sampling method to generate random variables from a half normal distribution.

```
s.absnorm = function(n) {
   y = vector(length=n)
   for(i in 1:n){
      y[i] = -log(runif(1))
      while (runif(1) > exp(-(y[i]-1)^2/2)) y[i] = -log(runif(1))
   }
return(y)
}

MC = 1000
set.seed(1)

mu = 2
sigma = 0.5

s.norm = sigma*(sign(runif(MC)-0.5)*s.absnorm(MC)) + mu
shapiro.test(s.norm)$p.value
```

```
[1] 0.6076152
```

```
qqnorm(s.norm)
qqline(s.norm)
```


Normal Q-Q Plot
Sample Quantiles
Theoretical Quantiles
0.5
1.0
1.5
2.0
2.5
3.0
3.5
-3
-2
-1
0
1
2
3

In Python

```python
import numpy as np
from scipy import stats

def s_absnorm(n):
    y = np.zeros(n)
    for i in range(n):
        y[i] = -np.log(np.random.uniform())
        while np.random.uniform() > np.exp(-(y[i] - 1)**2 / 2):
            y[i] = -np.log(np.random.uniform())
    return y

def generate_normal_samples(mu, sigma, n_samples):
    s_norm = sigma * (np.sign(np.random.uniform(size=n_samples) - 0.5) *
                                             s_absnorm(n_samples)) + mu

    return s_norm

# Set parameters
mu = 2
sigma = 0.5
n_samples = 1000

# Set random seed for reproducibility
np.random.seed(1)

# Generate samples
samples = generate_normal_samples(mu, sigma, n_samples)

# Output some samples
print(samples[:10])  # Print first 10 samples for illustration

# Shapiro-Wilk test for normality
shapiro_test = stats.shapiro(samples)
p_value = shapiro_test.pvalue

print("P-value:", p_value)
```

```
[1.4392795  2.14270385 1.62193418 1.80002237 1.86787982 1.17854505
 1.66534427 1.58074655 1.605236   2.06787083]
P-value: 0.13808122277259827
```

The *Alias Method* for generating discrete random variables

The Alias Method for generating discrete random variables is often attributed to Walker (1974) and it is a classical approach to generate an integer $i \in \{1, 2, \ldots, n\}$ with prescribed probabilities $p_1, p_2, \ldots, p_n$ in $O(1)$ time after an $O(n)$ preprocessing step.

We want to simulate a discrete random variable X taking values in $\{1, 2, \ldots, n\}$ such that

$$P(X = i) \;=\; p_i,$$

where $p_1 + p_2 + \cdots + p_n = 1$.

After an $O(n)$ time preprocessing of the probabilities p_i, we want to be able to generate a sample from this distribution in $O(1)$ time for each draw.

The Alias Method converts the problem of sampling from an arbitrary discrete distribution into the following two-step process:

1. Choose an index i uniformly from $\{1, 2, \ldots, n\}$. That is, pick i with probability $1/n$.
2. Use one additional Bernoulli trial (coin flip) to decide whether to accept i as the generated outcome or to replace it ("alias" it) by some other index j.

We build a table of size n, where each entry i has:

- A number q_i (sometimes called `prob[i]` in codes), which we will interpret as the probability threshold for deciding whether the outcome is i or some "alias".
- An alias a_i (sometimes called `alias[i]`), which is another index in $\{1, 2, \ldots, n\}$.

Then, when we pick i uniformly, we flip a (biased) coin that comes up heads with probability q_i. If it comes up heads, we output i. If it comes up tails, we output a_i.

Then, the sampling algorithm is described as:

$$\mathsf{AliasSampler}(\{q_i, a_i\}_{i=1}^n) : \begin{cases} \text{1. Choose } i \text{ uniformly in } \{1, \ldots, n\}. \\ \\ \text{2. Draw a Bernoulli}\big(q_i\big). \\ \quad \text{if the Bernoulli is heads, return } i, \\ \quad \text{else return } a_i. \end{cases}$$

Construction of the Alias Table

The core of the method is how to construct the entries $\{q_i, a_i\}$

1. We want that

$$P(\text{output} = i) = p_i.$$

2. In the final sampling scheme, each of the n indices i is chosen with probability $\frac{1}{n}$ at step 1, and then we refine the choice using the Bernoulli draw with probability q_i.

Intuitively, each row i in the alias table tries to represent a “slice” of size p_i within a total “pie” of size 1, but forced into a uniform “width” of $\frac{1}{n}$.

Some rows will not use up their entire slice $\frac{1}{n}$ (these rows correspond to $p_i < \frac{1}{n}$), and some rows will exceed $\frac{1}{n}$ (these correspond to $p_i > \frac{1}{n}$). We handle the “unused portion” (for those that are smaller) by aliasing them to those that are bigger.

Construction Algorithm

1. Initialize two lists (or stacks):

$$L = \emptyset \quad \text{(the list of “small” indices)}, \quad G = \emptyset \quad \text{(the list of “large” indices)}.$$

2. Scale probabilities in terms of np_i. Define

$$r_i = n\,p_i.$$

3. Partition indices into two groups:

$$L \;=\; \{i : r_i < 1\}, \quad G \;=\; \{i : r_i > 1\}.$$

   (If $r_i = 1$, we can put i in either list or treat it separately.)

4. While both L and G are nonempty:

   - Pop an index i from L (small).
   - Pop an index j from G (large).
   - Set

$$q_i = r_i \quad \bigl(\text{recall that } r_i = np_i\bigr),$$

     and alias $a_i = j$.

- Decrease r_j by $(1 - r_i)$ (i.e., $r_j := r_j - (1 - r_i)$) to account for giving "some leftover probability" from the row j to fill up the row i.
- If after this subtraction $r_j < 1$, move j into L; if $r_j > 1$, keep j in G. (If it becomes exactly 1, it is "done").

5. For any index still in L or G after the loop:

   Set $q_i = 1$ and $a_i = i$. (Or equivalently, if $r_i = 1$, we simply define $q_i = 1$.)

This procedure ensures that:

- Each index i ultimately has a number q_i satisfying $0 < q_i \leq 1$.
- Each index i has exactly one alias a_i.

Complexity: The above construction can be done in $O(n)$ time because each index is inserted into and removed from exactly one of the lists L or G at most once.

Example Distribution

For a concrete illustration, suppose you have four outcomes A, B, C, D with probabilities:

$$p_A = 0.1, \quad p_B = 0.4, \quad p_C = 0.2, \quad p_D = 0.3.$$

Then $n = 4$. For the Alias Method, we first multiply each p_i by $n = 4$:

$$q_A = 4 \times p_A = 0.4, \quad q_B = 4 \times p_B = 1.6, \quad q_C = 4 \times p_C = 0.8, \quad q_D = 4 \times p_D = 1.2.$$

Separating "Small" and "Large" Probabilities

We separate the outcomes based on whether $q_i < 1$ or $q_i \geq 1$:

$$\text{Small list: } \{A\ (0.4),\ C\ (0.8)\},$$
$$\text{Large list: } \{B\ (1.6),\ D\ (1.2)\}.$$

In ASCII form:

```
Small list      Large list
----------      ----------
    A (0.4)         B (1.6)
    C (0.8)         D (1.2)
```

Building the Alias Tables

We create two arrays of length 4, Prob[1..4] and Alias[1..4]. We fill them by repeatedly pairing one "small" element with one "large" element. Below is a step-by-step illustration:

Step 1: Pair A with B

$$q_A = 0.4, \quad q_B = 1.6.$$

- Set Prob$[A] = 0.4$.
- Set Alias$[A] = B$.
- Update $q_B := q_B - (1 - 0.4) = 1.6 - 0.6 = 1.0$.

Hence, A is "used up," and B's new "leftover" is 1.0.

Step 2: Pair C with D

Moving on (since $B = 1.0$ can be considered "borderline" or effectively used up), we now pair:

$$q_C = 0.8, \quad q_D = 1.2.$$

- Set Prob$[C] = 0.8$.
- Set Alias$[C] = D$.
- Update $q_D := q_D - (1 - 0.8) = 1.2 - 0.2 = 1.0$.

So C is used up, and D also lands at 1.0.

Finalize Slots for B and D

When a leftover is exactly 1.0, we can set that slot's probability to 1.0 and alias to itself. Thus:

$$\text{Prob}[B] = 1.0, \quad \text{Alias}[B] = B,$$

$$\text{Prob}[D] = 1.0, \quad \text{Alias}[D] = D.$$

Final Alias Table

If we label $A = 1, B = 2, C = 3, D = 4$, the final tables might look like:

Index i	Outcome	Prob[i]	Alias[i]
1	A	0.4	B
2	B	1.0	B
3	C	0.8	D
4	D	1.0	D

An ASCII version:

```
Index:      1     2     3     4

           (A)   (B)   (C)   (D)

Prob:      0.4   1.0   0.8   1.0

Alias:      B     B     D     D
```

Sampling Procedure

To sample from the constructed Alias Table:

1. Choose i uniformly at random from $\{1, 2, 3, 4\}$.
2. Flip a biased coin with probability Prob[i]:

$$\text{With probability Prob}[i],\ \text{return } i, \quad \text{else return Alias}[i].$$

Here is a flow diagram:

```
Sample():
  1) i <- UniformPick({1,2,3,4})


   +-----------------------------------------------------------------+
   | With probability Prob[i], return i. Otherwise, return Alias[i]. |
   +-----------------------------------------------------------------+
```

Intuitive Bin Visualization

A popular way to visualize the final structure is to imagine each column (index) as a bin of total height 1. "Prob[i]" is the top portion, and "(1 - Prob[i])" is the bottom (alias) portion. In our example:

```
Height
   1.0  +-----+    +-----+    +-----+    +-----+
        | B   |    | B   |    | D   |    | D   |
        |0.6|       |0.0|       |0.2|       |0.0|
        +-----+    +-----+    +-----+    +-----+
        | A   |    | B   |    | C   |    | D   |
   0.0  -----------------------------------------
          Col1       Col2       Col3       Col4
          i=1         i=2        i=3        i=4
```

where:

- Column 1 has Prob[1] = 0.4 for A on top, and alias B on bottom.
- Column 2 is all B (prob 1.0).
- Column 3 has top C with prob 0.8, and bottom D with leftover 0.2.
- Column 4 is all D (prob 1.0).

To sample:

1. Pick a column i with probability $\frac{1}{n}$.
2. With probability Prob[i], pick the top outcome; otherwise pick the alias outcome.

Why Does It Work? (Proof)

Proof

Here is a proof showing why this procedure produces the correct distribution p_i for each i.

Let X be the final output of the sampling algorithm.

Decomposition by the Law of Total Probability

1. First, let I be the uniformly chosen index from $\{1, 2, \ldots, n\}$ in Step 1 of the algorithm. Then for any k with $1 \le k \le n$,
$$P(I = k) = \frac{1}{n}.$$

2. Condition on $I = k$. After picking $I = k$, the algorithm does exactly one coin flip with success probability q_k. We define two possible outcomes of the coin flip:

   - **Heads (with probability q_k):** The algorithm returns k.
   - **Tails (with probability $1 - q_k$):** The algorithm returns a_k.

3. Therefore, for a specific outcome i, we can write
$$P(X = i) = \sum_{k=1}^{n} P\big(I = k\big) \cdot P\big(X = i \mid I = k\big).$$
Since $P(I = k) = 1/n$, this becomes
$$P(X = i) = \frac{1}{n} \sum_{k=1}^{n} P\big(X = i \mid I = k\big).$$

Computing $P(X = i \mid I = k)$

We next examine $P(X = i \mid I = k)$ in detail:

1. **Case 1: $k = i$.** If the chosen index k equals i, then the outcome X will be i exactly when the coin flip is heads. The probability of heads is q_i. Hence
$$P\big(X = i \mid I = i\big) = q_i.$$

2. **Case 2: $k \neq i$.** If the chosen index k differs from i, then the algorithm outputs i if and only if $a_k = i$ *and* the coin flip is tails (probability $1 - q_k$). In other words, if $a_k \neq i$, then $P(X = i \mid I = k) = 0$. If $a_k = i$, we get $P(X = i \mid I = k) = 1 - q_k$. Symbolically,
$$P\big(X = i \mid I = k\big) = \mathbf{1}\big(a_k = i\big)\,(1 - q_k),$$
where $\mathbf{1}(\cdot)$ denotes the indicator function (1 if the condition is true, 0 otherwise).

Putting these together:
$$P\big(X = i \mid I = k\big) = \mathbf{1}(k = i)\, q_i + \mathbf{1}(a_k = i)\,(1 - q_k).$$

Summation Over All k

Recall that $P(I = k) = 1/n$. Thus,

$$P(X = i) = \frac{1}{n} \sum_{k=1}^{n} P\big(X = i \mid I = k\big).$$

Plug in the expression for $P(X = i \mid I = k)$:

$$P(X = i) = \frac{1}{n} \sum_{k=1}^{n} \Big[\mathbf{1}(k = i)\, q_i \;+\; \mathbf{1}(a_k = i)\,(1 - q_k)\Big].$$

We can break this sum into two parts:

1. The term $\mathbf{1}(k = i)\, q_i$ is nonzero only when $k = i$. Hence it contributes exactly q_i when $k = i$, and 0 otherwise. As we sum over $k = 1$ to n, exactly one k satisfies $k = i$, so that portion yields q_i.

2. The term $\mathbf{1}(a_k = i)\,(1-q_k)$ is nonzero only when the alias a_k equals i. In that case, it contributes $1 - q_k$. Summing over all k from 1 to n thus gives us

$$\sum_{k=1}^{n} \mathbf{1}(a_k = i)\,(1 - q_k) = \sum_{k:\, a_k = i} (1 - q_k).$$

Therefore,

$$\sum_{k=1}^{n} \Big[\mathbf{1}(k = i)\, q_i + \mathbf{1}(a_k = i)\,(1 - q_k)\Big] = q_i + \sum_{k:\, a_k = i} (1 - q_k).$$

Thus,

$$P(X = i) = \frac{1}{n}\Big[\, q_i \;+\; \sum_{k:\, a_k = i} (1 - q_k)\Big].$$

For the final distribution to match p_i, we require

$$P(X = i) = p_i, \quad \text{for each } i = 1, 2, \ldots, n.$$

Using the derived expression, this translates to:

$$\frac{1}{n}\Big(q_i + \sum_{k:\, a_k = i} (1 - q_k)\Big) \;=\; p_i,$$

i.e.,

$$q_i + \sum_{k:\, a_k = i} (1 - q_k) \;=\; n\, p_i.$$

Hence, the condition for correctness is:

$$q_i + \sum_{k:\, a_k = i} (1 - q_k) = n\, p_i, \quad \forall i. \tag{1}$$

The construction of the alias table is precisely designed to satisfy the conditions: when we set $q_i = r_i = n\,p_i$ for each row i (if it is a "small" row) and then use an alias j to fill the remainder $(1 - q_i)$, we effectively split $n\,p_j$ among its own row j plus possibly other rows i that need leftover space.

Each time we "pair" a small row i with a large row j, we enforce the identity $q_i = n\,p_i$ and then reduce r_j by $1 - r_i$, ensuring the total allocations remain consistent with the original probabilities.

The alias table construction ensures that for every outcome i, exactly $n\,p_i$ total probability mass is distributed among i's own cell (q_i) and the cells that alias to i $(1 - q_k)$. Consequently, the condition (1) is met.

Thus, by the law of total probability and the above step-by-step summation, we have

$$P(X = i) = p_i$$

for all i. Hence the algorithm correctly samples from the desired discrete distribution.

Extensions: The method is often used in event simulation, Monte Carlo methods, and anywhere one needs to sample from a discrete distribution many times. It can be extended to handle dynamic updates in more sophisticated variants (though that typically costs more than $O(1)$ time per update).

Summary

We want to generate a discrete random variable X taking values in $\{1, 2, \ldots, n\}$ with

$$P(X = i) = p_i, \quad \sum_{i=1}^{n} p_i = 1.$$

Construction

1. Define $r_i = np_i$ for $i = 1, \ldots, n$.

2. Let $L = \{i : r_i < 1\}$ and $G = \{i : r_i > 1\}$.

3. While both L and G are not empty:

    (a) Pop i from L and j from G.

(b) Set $q_i = r_i$ and $a_i = j$.

(c) Update $r_j := r_j - (1 - r_i)$.

(d) If $r_j < 1$, move j to L (otherwise keep it in G).

4. For any i remaining, set $q_i = 1$ and $a_i = i$.

Sampling

To sample from $\{1, \ldots, n\}$:

$$\begin{cases} I \leftarrow \text{Uniform}\{1, \ldots, n\}, \\ \text{draw Bernoulli}(q_I), \\ \text{if Heads, return } I, \text{ else return } a_I. \end{cases}$$

Correctness

By construction, we have

$$P(X = i) \; = \; \sum_{k=1}^{n} \frac{1}{n} \Big[\mathbf{1}(k = i) q_i \; + \; \mathbf{1}(a_k = i)(1 - q_k) \Big] \; = \; p_i.$$

Thus the Alias Method generates the desired distribution in $O(1)$ time per sample after $O(n)$ preprocessing.

Intuitive ideas

Imagine that you want to sample from the following discrete distribution of letters $\{A, T, C, G\}$. Each letter has a probability (height) h_i. The total probability is 1, so the sum of all heights is 1. Figure 1 shows a simple histogram representation of the distribution:

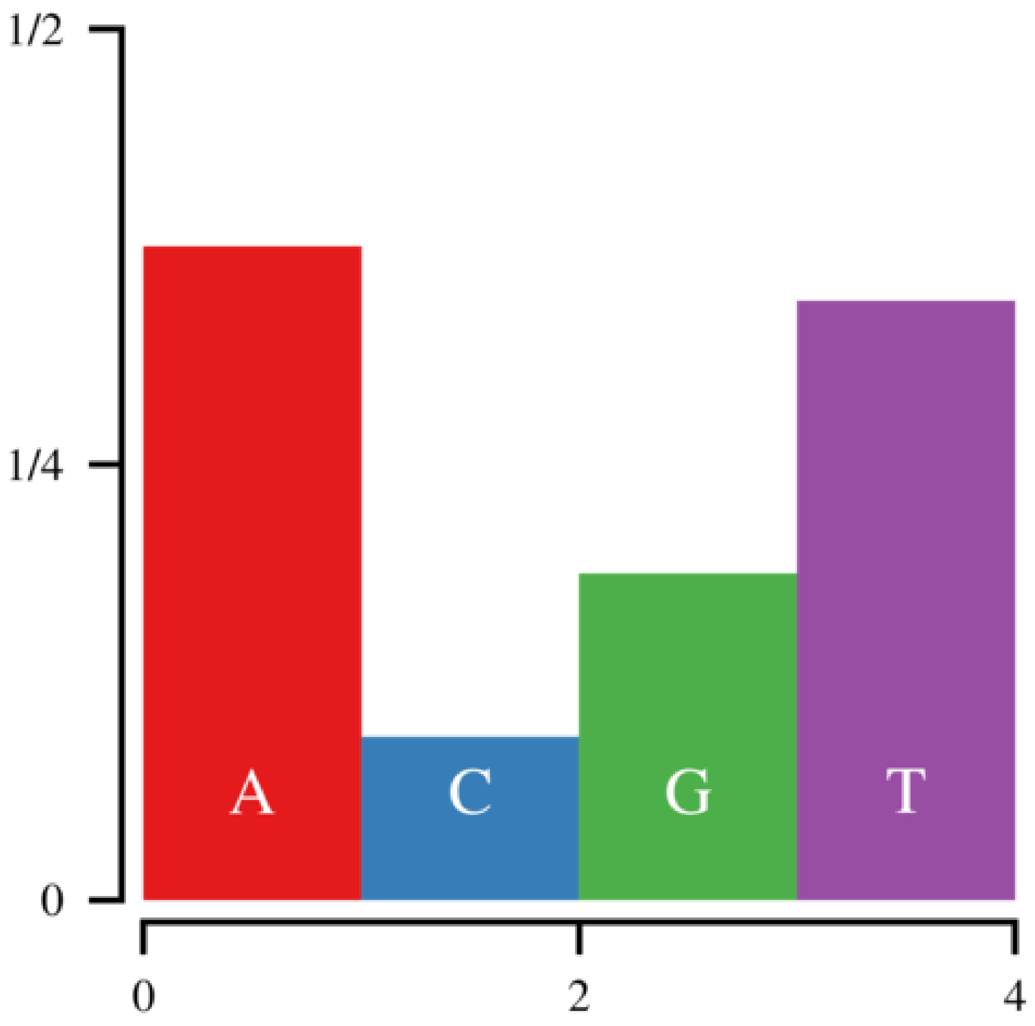


Figure 1: Histogram of probabilities for letters A, C, G, and T. The heights of the bars are h_0, h_1, h_2, h_3, which sum to 1.

Sampling by Walking Through the Histogram

A straightforward way to sample from this histogram is:

1. Draw a uniform random number $u \in (0, 1)$.
2. Move from left to right across the bars until the cumulative height exceeds u.
3. The letter corresponding to that bar is returned as the sample.

Though this works, it can be slow if there are many bars, because each sample may require several comparisons.

Sampling with a Bounding Box (Rejection Sampling)

Another way is to enclose the bars in a bounding rectangle, as in Figure 2:

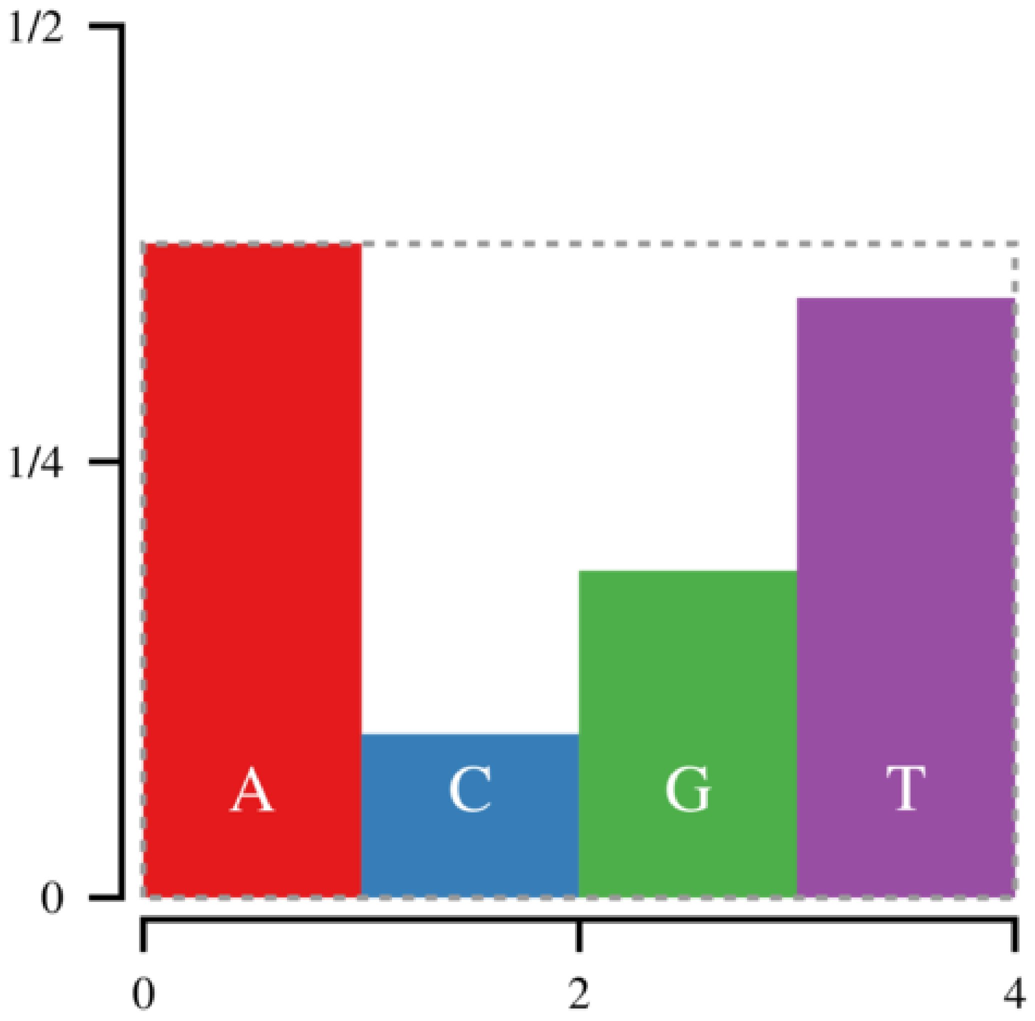


Figure 2: A bounding box (width = 4, height = 3/8) containing the bars. Note the unused "white space" above some bars.

The idea is:

1. Draw (x, y) uniformly from the bounding box.

$$x \sim \text{Uniform}(0, 4), \quad y \sim \text{Uniform}\big(0, \tfrac{3}{8}\big).$$

2. Identify which bar is at position x.

3. If y is under that bars top (i.e. below its height), accept that letter. Otherwise, reject and draw again.

This can be more efficient than walking through the histogram because we only compare y to one bars height each time. However, the empty space above the shorter bars leads to a certain rejection rate, so on average you might generate more than one point per valid sample.

Constructing a Filled Box: Alias Idea

Notice in Figure 2 that the amount of "white space" within the rectangle equals the "excess space" sticking out of the top of the shorter bars. This hints that we can *cut* the histogram into blocks and rearrange them to fill the bounding box exactly, leaving no empty space. Figure 3 shows a schematic of what the filled box looks like:

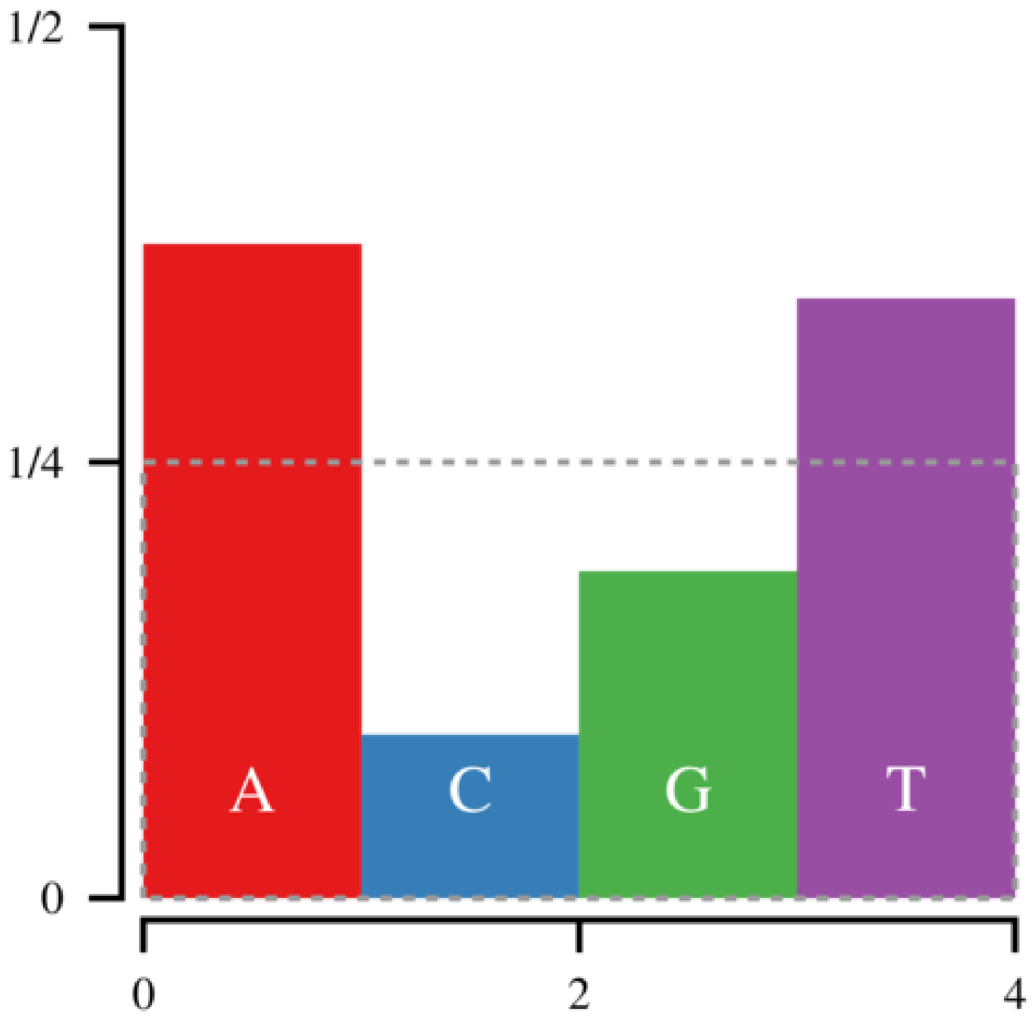


Figure 3: A rearranged histogram so that the bounding box has no empty space. Each bar now has a "normal" region (lower) and an "alias" region (upper).

By carefully slicing and moving pieces of the tall bars into the empty space above the shorter bars, we create:

(1) A uniform strip in width for each outcome, (2) Two sections in height (normal/alias) that perfectly fill

Now, sampling is simpler:

1. Generate a uniform random number $U_1 \in (0, 1)$. Multiply by n (the number of outcomes) to pick which bar (strip) you land in.
2. Generate another uniform random number $U_2 \in (0, 1)$. If U_2 is below the "split" between normal and alias, return the primary (normal) letter; otherwise return the alternate (alias) letter.

No rejections occur because the box is completely filled. We only need:

$$\text{(1) One table lookup (which strip),} \quad \text{(2) One comparison (which region).}$$

Creating the Alias Structure (Large/Small Lists)

The main puzzle is how to slice/move these bars so everything fits. One standard algorithm uses two lists:

$$\text{Large} = \{\text{bars above the average height}\}, \quad \text{Small} = \{\text{bars below the average height}\}.$$

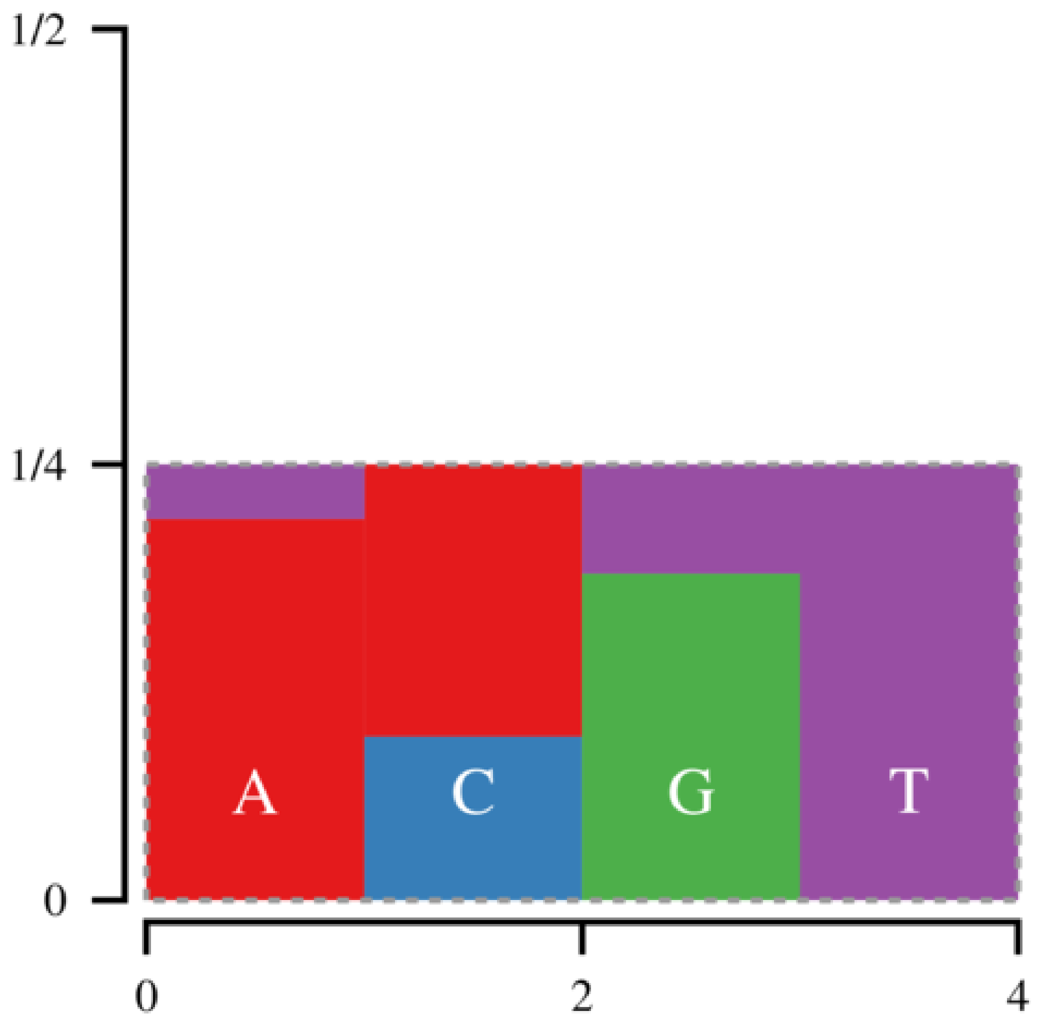
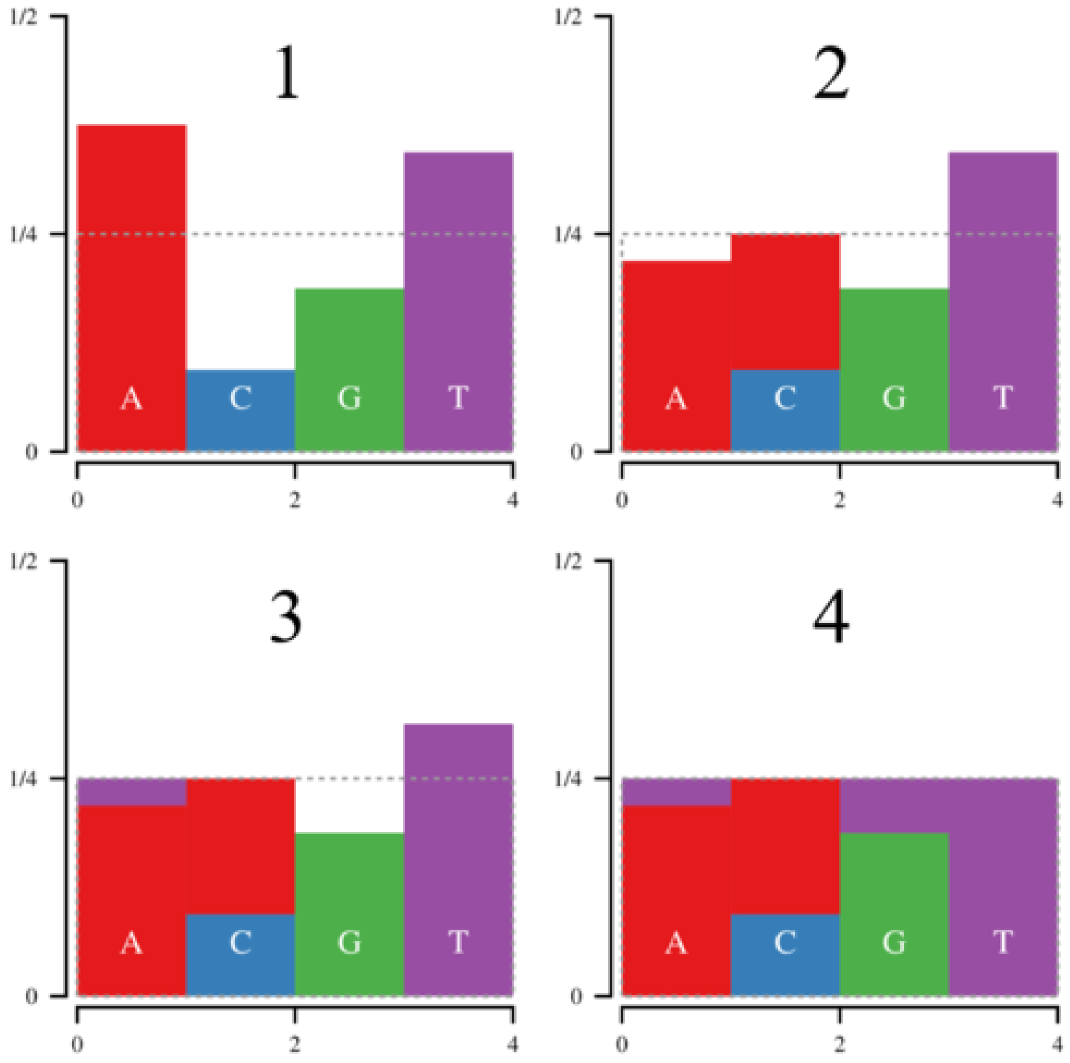


Figure 4: Conceptual view of slicing and moving blocks from the large bars to fill the small bars so that all strips end up at the average height.

We repeatedly pair one bar from Large with one bar from Small. We "trim" the large bar so that it exactly fills the gap in the small bar, making the small bar reach the average. If there is any leftover from the large bar, it remains (potentially still large or possibly small now). This process ensures that by the end, every bar has the same *effective* height (the average). Along the way, we record which letters become "aliases" for each other.

The process looks a bit like this:


1
2
3
4
1/2
1/4
0
A
C
G
T
0
2
4

Example: Generating a random variable from a custom discrete distribution

Suppose that we have a discrete distribution with four outcomes (for example, A, B, C, D) with probabilities $0.1, 0.2, 0.4, 0.3$ respectively.

Step 1: Define the probability distribution

```r
# Probabilities for each outcome
probabilities = c(0.1, 0.2, 0.4, 0.3)

# The outcomes
outcomes = c("A", "B", "C", "D")
```

Step 2: Preprocess to create Alias and Probability Tables

```r
# Create alias and probability tables

# - "n": the number of outcomes.
# - "average": the average probability (since we aim to make each slot
#    in the table have an equal chance, which is 1/n).
# - "alias" and "probability": two vectors to store the alias table and
#    adjusted probabilities, respectively.
# - "small" and "large": lists to store indices of probabilities below
#    and above the average

n = length(probabilities)
average = 1 / n
alias = numeric(n)
probability = numeric(n)

small = list()
large = list()

for (i in 1:n) {
# It loops through each probability and classifies it as either small
# or large based on whether it is below or above the average,
# adding the index to the respective list.

  if (probabilities[i] < average) {
    small = c(small, i)
  } else {
    large = c(large, i)
  }
}
```

```r
while (length(small) > 0 && length(large) > 0) {
# Create Alias and Probability Tables:
# This block iterates until all probabilities are allocated to either
# the "small" or "large" list.
# For each pair of small (l) and large (g) probabilities, it adjusts
# the probabilities so that each bucket (an element in the "probability" array)
# has the same total probability.
# It sets up the alias table by pointing from the smaller probability to
# the larger, adjusting the remaining probability of "g" accordingly, and
# reallocating "g" to the appropriate list based on its new probability.

  l = small[[1]]
  g = large[[1]]
  small = small[-1]
  large = large[-1]
  probability[l] = probabilities[l] * n
  alias[l] = g
  probabilities[g] = probabilities[g] - (average - probabilities[l])
  if (probabilities[g] < average) {
    small = c(small, g)
  } else {
    large = c(large, g)
  }
}

# Ensures all remaining probabilities (which will all be large at
# this point) are set to 1, as they don't need an alias.
for (g in large) { probability[g] = 1 }

generate_random_variable = function() {
# Function to generate a random variable using the Alias Method
# It randomly selects an index i (simulating the choice of a bucket)
# and then uses a uniform random number to decide whether to take
# the value associated with i or its alias.
# This decision is based on the adjusted probability for i

  i = sample(n, 1)
  if (runif(1) < probability[i]) {
    return(outcomes[i])
  } else {
    return(outcomes[alias[i]])
  }
}
```

Step 4: Generate random variables

```r
# Generate 10 random variables
set.seed(123) # For reproducibility
random_variables = replicate(10, generate_random_variable())
print(random_variables)
```

```
[1] "C" "C" "C" "D" "C" "D" "B" "D" "D" "C"
```

This iterative process ensures that each outcome can be sampled with a simple two-step procedure: first, randomly select a bucket and then use a uniform random number to decide whether to take the outcome directly associated with that bucket or to switch to its alias.

This system allows the Alias method to efficiently sample from complex discrete distributions using a setup that guarantees a sampling time of $O(1)$ after pre-processing of $O(n)$.

In `R`, when applying `sample()`, if `replace` is `TRUE`, Walker's alias method is used when there are more than 250 reasonably probable values.

Other example

```
# Let us create a function that builds the alias table
build_alias_table = function(prob) {
  n = length(prob)

  # The alias method needs two arrays:
  alias = integer(n)
  prob_adj = numeric(n)

  # We first scale our probabilities so that the average is 1
  # then we separate them into two groups: small and large
  scaled_prob = prob * n

  # Queues (or lists) to hold the indices of small and large
  small = vector()
  large = vector()

  for (i in seq_len(n)) {
    if (scaled_prob[i] < 1.0) {
      small = c(small, i)
    } else {
      large = c(large, i)
    }
  }

  # Fill the alias and prob_adj tables
  while (length(small) > 0 && length(large) > 0) {
    s = small[length(small)]  # get the last small
    small = small[-length(small)]
    l = large[length(large)]  # get the last large
    large = large[-length(large)]

    prob_adj[s] = scaled_prob[s]
    alias[s] = l
    scaled_prob[l] = (scaled_prob[l] + scaled_prob[s]) - 1.0

    if (scaled_prob[l] < 1.0) {
      small = c(small, l)
    } else {
      large = c(large, l)
    }
  }

  # For those still in large or small, set prob_adj to 1
  # because they won't need any aliasing
  while (length(large) > 0) {
    l = large[length(large)]
    large = large[-length(large)]
    prob_adj[l] = 1
  }
  while (length(small) > 0) {
    s = small[length(small)]
    small = small[-length(small)]
    prob_adj[s] = 1
  }

  # Return a list that has the probability adjustments and alias table
  list(prob_adj = prob_adj, alias = alias)
}

# Let's say we have 3 flavors:
flavors = c('strawberry', 'chocolate', 'vanilla')
# Their probabilities:
p = c(0.2, 0.5, 0.3)

# Build the alias table:
alias_info = build_alias_table(p)

# Now let's create a function to sample from it
sample_alias = function(alias_info) {
  n = length(alias_info$prob_adj)
  # Step 1: pick a slot from 1..n uniformly
  i = sample.int(n, size = 1)
  # Step 2: flip a tiny 'coin' to see if we stay or go to the alias
  if (runif(1) < alias_info$prob_adj[i]) {
    return(i)
  } else {
    return(alias_info$alias[i])
  }
}

# Generate 10 random flavors
set.seed(123)  # just for reproducibility
```

```r
picks = sapply(1:10, function(x) {
  idx = sample_alias(alias_info)
  flavors[idx]
})
picks
# This should give you 10 flavors chosen
# according to p = (0.2, 0.5, 0.3).
```

```
[1] "vanilla"    "vanilla"    "vanilla"    "chocolate"  "vanilla"    "chocolate"  "strawberry"
    "vanilla"    "strawberry" "vanilla"
```

In Python

Step 1: Define the Probability Distribution

```python
import random

# Probabilities for each outcome
probabilities = [0.1, 0.2, 0.4, 0.3]

# The outcomes
outcomes = ["A", "B", "C", "D"]
```

Step 2: Preprocess to Create Alias and Probability Lists

```python
n = len(probabilities)
average = 1.0 / n
alias = [0] * n
probability = [0] * n

small = []
large = []

for i in range(n):
    if probabilities[i] < average:
        small.append(i)
    else:
        large.append(i)

while small and large:
    l = small.pop()
    g = large.pop()
    probability[l] = probabilities[l] * n
    alias[l] = g
    probabilities[g] = probabilities[g] - (average - probabilities[l])
    if probabilities[g] < average:
        small.append(g)
    else:
        large.append(g)
```

```
for g in large:
    probability[g] = 1
```

Step 3: Function to Generate a Random Variable

```
def generate_random_variable():
    i = random.randint(0, n - 1)
    if random.random() < probability[i]:
        return outcomes[i]
    else:
        return outcomes[alias[i]]
```

Step 4: Generate Random Variables

```
# Generate 10 random variables
random.seed(123)  # For reproducibility
random_variables = [generate_random_variable() for _ in range(10)]
print(random_variables)
```

```
['A', 'D', 'A', 'C', 'A', 'C', 'B', 'C', 'D', 'C']
```