Generating non-Uniform Distributions using Monte Carlo Methods

Ahmer Nadeem Khan

02 April 2025

1 Abstract

We have briefly discussed the expansive theory and methodology concerning the generation of pseudo-random (and low discrepancy) sequences using (quasi) Monte Carlo methods. These methods are focused on generating uniform numbers (that is, samples from the Uniform distribution). We now discuss the generation of non-Uniform sequences, using the uniform sequences as a foundation. Obviously we are routinely interested in all sorts of distributions - of particular importance is the Normal distribution, with vast applications in mathematical and computational finance. (Homework problems are marked in red).

2 Objectives

In this assignment, we first describe well-known methods to generate numbers from non-uniform distributions, including

- Inverse Transformation Method (a general method), which is used whenever the inverse transformation of the cdf is available and inexpensive to implement. We focus on the Normal distribution
- Acceptance-Rejection Method, which is also a general method, but works under certain constraints
- We then describe how to generate Brownian Motion (Wiener Process) recursively
- We implement selected algorithms and methods for simple examples to demonstrate correctness of our analysis and implementations.

3 Non-Uniform PRNGs

Inverse Transformation Method

The initial most useful theorem to us is the following:

Theorem 3.1 (Inverse transformation method)**.** *Let $U_1, U_2, \ldots, U_n$ be i.i.d. random variables from the uniform distribution $U(0,1)$. Let F be a distribution function and $X_i = F^{-1}(U_i)$. Then $X_1, X_2, \ldots, X_n$ are i.i.d. random variables from the distribution F.*

Proof. Simply observe that

$$P\{X_i \leq x\} = P\{F^{-1}(U_i) \leq x\} = P\{U_i \leq F(x)\} = F(x). \qquad \square$$

We can apply this theorem to produce random numbers from a discrete, countable (non-uniform) distribution. Consider the probability density function of a discrete r.v. X

$$P(X = x_i) = p_i, \quad i = 1, 2, \ldots$$

where $\sum p_i = 1$. To generate a pseudo-random number from X using the inverse transformation method, first generate a number u from $U(0,1)$ and set

$$X = \begin{cases} x_1 & \text{if } u < p_1 \\ x_2 & \text{if } p_1 \leq u < p_1 + p_2 \\ \vdots & \\ x_i & \text{if } p_1 + \cdots + p_{i-1} \leq u < p_1 + \cdots + p_i \\ \vdots & \end{cases}$$

where we have split the interval into a series of sub-intervals of Lebesgue/uniform measure of $\lambda(I_i) = p_i$. Then exploiting the uniformity of u (the sample from the Uniform distribution), we can simulate the discrete, non-uniform (or indeed a uniform one) distribution by assigning probabilities if $u \in I_i$, with the obvious definition. Note that this construction is not unique, but this is the simplest, and we have relied on almost sure properties of uniform sequences. With this construction, we also get that if F is the distribution function for X, the above expression (if the inequalities are left closed, right open) is the *generalized inverse function*:

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

This, and other methods, apply directly to QMC sequences due to the definition of discrepancy; in fact, we may reproduce slightly modified foundational results for QMC sequences.

Definition 1. *A sequence $(x_n) \subset [0,1)$ is said to be* uniformly distributed mod 1 with respect to a Borel probability measure *μ on $[0,1)$ if for every interval $(a,b) \subset [0,1)$, we have:*

$$\lim_{N \to \infty} \frac{1}{N} \sum_{n=1}^{N} \mathbf{1}_{[a,b)}(x_n) = \mu((a,b))$$

where $\mathbf{1}_{[a,b)}$ is the indicator function of the interval $[a,b)$.

The corresponding result, which we state without proof, is:

Theorem 3.2 (Chelson '76, Aistleitner & Dick '10). *Let f be measurable, and of bounded variation on $[0,1]^s$ in the sense of Hardy and Krause. Let μ be a finite measure on $[0,1]^s$. Then for any $N > 0$,*

$$\left| \frac{1}{N} \sum_{n=1}^{N} f(x_n) - \int_{[0,1]^s} f(x)\, d\mu \right| \leq V(f) D_N^*(x_n; \mu)$$

which has the form we expect.

(Exercise 3.6.10) If we know how to generate from a set of distributions, we can also generate from a random sample with distribution the product of the original set. Suppose we know how to generate a random variable from any of the distributions $F_1, \dots, F_n$. Then we want to generate from the distribution

$$F(x) = \prod_{i=1}^{n} F_i(x).$$

First generate samples from the set of computable distributions independently, i.e. for $1 \leq i \leq n$, generate

$$X_i \sim F_i \text{ independently}$$

Then set

$$X = \max\{X_1, X_2, \dots, X_n\}$$

Then note that

$$P(\max(X_1, \dots, X_n) \leq x) = \prod_{i=1}^{n} P(X_i \leq x) = \prod_{i=1}^{n} F_i(x) = F(x)$$

from independence and the definition of cdf. This implies that the variable X has the required distribution F. Therefore, to sample from the distribution F, we generate n independent samples from each given distribution F_i (we assume this is possible), and compute the maximum of these samples to obtain a single sample from F.

Normal Distribution

We say X has normal distribution with parameters μ and σ^2, and write $X \sim \mathcal{N}(\mu, \sigma^2)$, if its density function is

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}, \quad -\infty < x < \infty.$$

Here $\mu = \mathbb{E}[X]$ is any real constant and $\sigma = \sqrt{\operatorname{Var}(X)}$ is any positive real constant. To generate $X \sim \mathcal{N}(\mu, \sigma^2)$ using the inverse transformation method, we need to invert its distribution function

$$F(x) = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{(y-\mu)^2}{2\sigma^2}}\, dy.$$

This can be done fairly efficiently using numerical techniques. One such technique that is popular in the financial math literature is the Beasley-Springer-Moro algorithm. The algorithm outputs an approximation to Φ^{-1}, where Φ is the cdf for $\mathcal{N}(0,1)$. The algorithm is fairly cumbersome to state here, and there are several other well-studied methods to generate the Normal distribution which we do not discuss here (e.g. the Box-Muller Method).

(Exercise 3.6.12) We are also interested in generating from the normal distribution conditional on a connected interval in the sample space, i.e. we want to generate the standard Normal Random Variable Z conditional on the event $a < Z < b$ where $a < b$ and $a, b \in \mathbb{R}$. Consider the algorithm

1. Generate u from $U(0,1)$
2. Compute $u' = \phi(a) + (\phi(b) - \phi(a))u$
3. Set $X = \phi^{-1}(u')$.

Note that steps 1,2 are the steps of the Inverse Transformation method, and step 2 is a positive linear scale into the interval $(\phi(a), \phi(b))$ of u, i.e

$$u' = \phi(a) + (\phi(b) - \phi(a))u$$

may be re-written as

$$u' = u\,\phi(b) + (1-u)\,\phi(a)$$

that is, we map the interval $(0,1)$ linearly and positively into the interval $(\phi(a), \phi(b))$. Clearly, $u' \sim U(\phi(a), \phi(b))$. The conditional cumulative distribution function of $Z \sim \mathcal{N}(0,1)$ given $a < Z < b$ is:

$$F_{Z|a<Z<b}(x) = \frac{\phi(x) - \phi(a)}{\phi(b) - \phi(a)}, \quad \text{for } a < x < b$$

which can be deduced very easily from the definition of the ϕ, and the fact that this is a continuous distribution (we divide by the integral on (a,b) to get a scaled cdf for the conditional). Now, u' has the pdf

$$P(u') = \frac{1}{\phi(b) - \phi(a)}$$

on the interval $(\phi(a), \phi(b))$ and 0 elsewhere. Then, integrating gives us the cdf which is simply

$$F(\phi(x)) = P(u' \leq \phi(x)) = \frac{\phi(x) - \phi(b)}{\phi(b) - \phi(a)} + 1 = \frac{\phi(x) - \phi(a)}{\phi(b) - \phi(a)}$$

on the interval $(\phi(a), \phi(b))$ and 0 on the interval $(-\infty, \phi(a)]$ and 1 on the interval $[\phi(b), \infty)$. The above equation then becomes

$$P(\phi^{-1}(u') \leq x) = \frac{x - \phi(a)}{\phi(b) - \phi(a)}$$

which is

$$P(X \leq x) = F_X = \frac{x - \phi(a)}{\phi(b) - \phi(a)}$$

i.e.

$$F_X = F_{Z|a<Z<b}$$

which is what we require (since densities are determined by distributions).

Acceptance-Rejection Method

The Acceptance-Rejection Sampling Method was first formulated by John Von Neumann in 1950, and allows us to generate pmf's (pdf's) using a known pmf (pdf). In particular, we want to generate samples from the variable X which has density f, and we know we can sample from the variable Y with density g. The idea is to sample values $y \in Y$, and accept or reject the value a sample from X on certain criterion. This first requires us that X and Y have the same sample space (or at least the same support). Once the requirements are met, the algorithm proceeds as follows:

1. Generate a value y from Y with pmf g
2. Generate u from $U(0, 1)$
3. If $u < \frac{f(y)}{cg(y)}$, set $X = y$ and stop. Else, return to step 1.

where $c \geq \sup \frac{f}{g}$. Note that the third step forces acceptance with probability $\frac{f}{cg}$ where we need to divide by c simply so this number is a probability in $[0, 1]$. Then, since Y has pmf g, y will appear as a sample of X with probability $\frac{f}{g} \times g = f$, which is what we require. This can be proven formally, but we state the relevant theorem here without proof.

Theorem 3.3. *The acceptance-rejection algorithm generates a random variable X with pmf f. In addition, the number of iterations of the algorithm needed to obtain a value from X is a geometric random variable with mean c.*

Note that since $f(i)/g(i) \leq c$ for all i, we have $f(i) \leq cg(i)$ and thus $\sum f(i) = 1 \leq c \sum g(i) = c$. So c is at least one. The average number of iterations needed in acceptance-rejection is c. Therefore the closer c is to 1, the more efficient is the acceptance-rejection method. Intuitively, if c is closer to 1, then that means the two pmf's $f(i)$ and $g(i)$ are "alike", or, $f(i) \approx g(i)$. This means when constructing an algorithm we want to want g to be as similar to f as possible. We primarily use acceptance-rejection when the inverse transformation of F is unavailable, impossible, or expensive to numerically compute or approximate.

Brownian Motion

Brownian Motion is the ubiquitous stochastic process in financial simulations, and is used to simulate stock price paths used in pricing derivatives, simulating interest rate models, or simply simulating the stock prices themselves. The definition of Brownian motion is as follows:

Definition 2. *A stochastic process $\{X(t); 0 \le t \le T\}$ is called a **Brownian motion with parameters** μ **and** σ on $[0, T]$ if*

1. *$X(0) = 0$.*
2. *The process has independent increments: for any $t_1 < t_2 < \ldots < t_n$, the increments*
$$X(t_2) - X(t_1),\, X(t_3) - X(t_2),\, \ldots,\, X(t_n) - X(t_{n-1})$$
are independent random variables.
3. *For every $0 \le s < t \le T$, the increment satisfies*
$$X(t) - X(s) \sim \mathcal{N}(\mu(t-s),\, \sigma^2(t-s)).$$

*The process is denoted by $BM(\mu, \sigma^2)$. The parameter μ is called the drift, and σ^2 the variance of the Brownian motion. The special case $\mu = 0$, $\sigma^2 = 1$ is called the **standard Brownian motion**, and denoted by $W(t)$.*

Note that if $X(t) \sim BM(\mu, \sigma^2)$, then $\frac{X(t)-\mu t}{\sigma\sqrt{t}}$ is a standard Brownian motion. We can then write

$$W(t) = \frac{X(t) - \mu t}{\sigma\sqrt{t}} \quad \Longrightarrow \quad X(t) = \mu t + \sigma\sqrt{t}W(t).$$

The path of a Brownian motion is continuous with probability 1, i.e., for a fixed w, $W(\cdot, w)$ is continuous. The path is nowhere differentiable and has infinite total variation with probability 1. The distribution of $W(t) - W(s)$ depends only on the time increment $t - s$. So, for example, $W(t) - W(s) \sim W(t-s)$.

We can use the independence of the increments to first discretize the time variable, and then recursively generate the path values. This is the random walk construction (BM is the limit of this discrete random walk). From the definition of a Brownian motion, the increments

$$X(t_1) - X(0),\; X(t_2) - X(t_1),\; \ldots,\; X(t_n) - X(t_{n-1})$$

are independent, and $X(t_i) - X(t_{i-1}) \sim \mathcal{N}(\mu(t_i - t_{i-1}),\; \sigma^2(t_i - t_{i-1}))$, $i = 1, \ldots, n$. Observe that the distribution of

$$\mu(t_i - t_{i-1}) + \sigma\sqrt{t_i - t_{i-1}}Z$$

is also $\mathcal{N}(\mu(t_i - t_{i-1}),\; \sigma^2(t_i - t_{i-1}))$, where Z is a standard normal random variable. Then we can write

$$\begin{aligned} X(t_1) - X(0) &= \mu(t_1 - t_0) + \sigma\sqrt{t_1 - t_0}Z_1 \\ X(t_2) - X(t_1) &= \mu(t_2 - t_1) + \sigma\sqrt{t_2 - t_1}Z_2 \\ &\vdots \\ X(t_n) - X(t_{n-1}) &= \mu(t_n - t_{n-1}) + \sigma\sqrt{t_n - t_{n-1}}Z_n \end{aligned}$$

where $Z_1, \dots, Z_n$ are independent standard normals.

Finally, to simulate the Brownian motion path $X(t_1), \dots, X(t_n)$, we generate independent numbers $z_1, \dots, z_n$ from the standard normal random variable Z, using the methods we have discussed before, and substitute them in the equations above to solve for $X(t_1), X(t_2), \dots, X(t_n)$ recursively. It turns out, we can in fact simulate the random walk in any order we want, not necessarily forward in time. The most common is the Brownian bridge construction, that simulates the values in the middle of each previous interval - the main idea is still the independence. We use the following lemma to our advantage.

Lemma 3.4. *The conditional distribution of $W(s)$ given that $W(t_1) = x_1$ and $W(t_2) = x_2$, where $t_1 < s < t_2$, is a normal distribution with*

$$\mathbb{E}[W(s) \mid W(t_1) = x_1 \text{ and } W(t_2) = x_2] = \frac{x_1(t_2 - s) + x_2(s - t_1)}{t_2 - t_1}$$

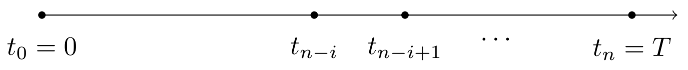
$$Var(W(s) \mid W(t_1) = x_1 \text{ and } W(t_2) = x_2) = \frac{(t_2 - s)(s - t_1)}{t_2 - t_1}.$$

Note the Lagrange interpolating form of the expectation - and the variance has the form of the newton divided difference error correction term! Then the standard normal variable Z acts as the divided difference coefficient as we generate the path recursively.

(Exercise 3.6.16) Let $0 = t_0 < t_1 < \cdots < t_n = T$. We want to simulate $W(t)$ backwards in time, i.e., given $W(0) = 0$, we want to simulate the Brownian motion in this order: $W(t_n), W(t_{n-1}), \dots, W(t_1)$. Given $W(0) = 0$, the equation for $W(T)$ is the same as the Brownian bridge construction, and follows directly from the independence of increments and the definition of Brownian motion:

$$W(T) = \sqrt{T}\,Z_1$$

We can construct the other equations in general now. Note the general situation is when we need $W(t_{n-i})$ from our history, and due to independence and Lemma 3.4, we only need to know $W(0) = 0$ and $W(t_{n-i+1})$.



Applying Lemma 3.4, we get $x_1 = 0, t_1 = 0, x_2 = W(t_{n-i+1}), t_2 = t_{n-i+1}$ and $s = t_{n-i}$ which gives us the equations:

$$\mu_i = \frac{W(t_{n-i+1})(t_{n-i})}{t_{n-i+1}}$$
$$\sigma_i^2 = \frac{t_{n-i+1} - t_{n-i}}{t_{n-i+1}}$$

then

$$W(t_{n-1}) = W(t_{n-i+1})\frac{(t_{n-i})}{t_{n-i+1}} + \sqrt{\frac{t_{n-i+1} - t_{n-i}}{t_{n-i+1}}}\, Z_i$$

gives us the recursive equation we require to generate the backwards Brownian motion.

4 Implementation and Results

We describe the implementation of selected methods, and then present basic results.

First, we want to test the **Inverse Transformation Method** for a simple, finite case. (Exercise 3.6.1) Let P be the measure on $\Omega = \{1, 2, 3, 4, 5\}$ s.t.

$$P(X = 1) = 0.2$$
$$P(X = 2) = 0.2$$
$$P(X = 3) = 0.4$$
$$P(X = 4) = 0.1$$
$$P(X = 5) = 0.1$$

Note that the sum is 1. Then, if we generate u from $U(0, 1)$, we can generate samples from the distribution of P above by setting

$$X = \begin{cases} 1 & \text{if } u < 0.2 \\ 2 & \text{if } 0.2 \leq u < 0.4 \\ 3 & \text{if } 0.4 \leq u < 0.8 \\ 4 & \text{if } 0.8 \leq u < 0.9 \\ 5 & \text{if } 0.9 \leq u \end{cases}$$

This follows directly from the description of the method in the beginning of the previous section.

In our test, we sample 1000 values from $U(0, 1)$ and apply the inverse transformation given by X. We then plot the values in a histogram:

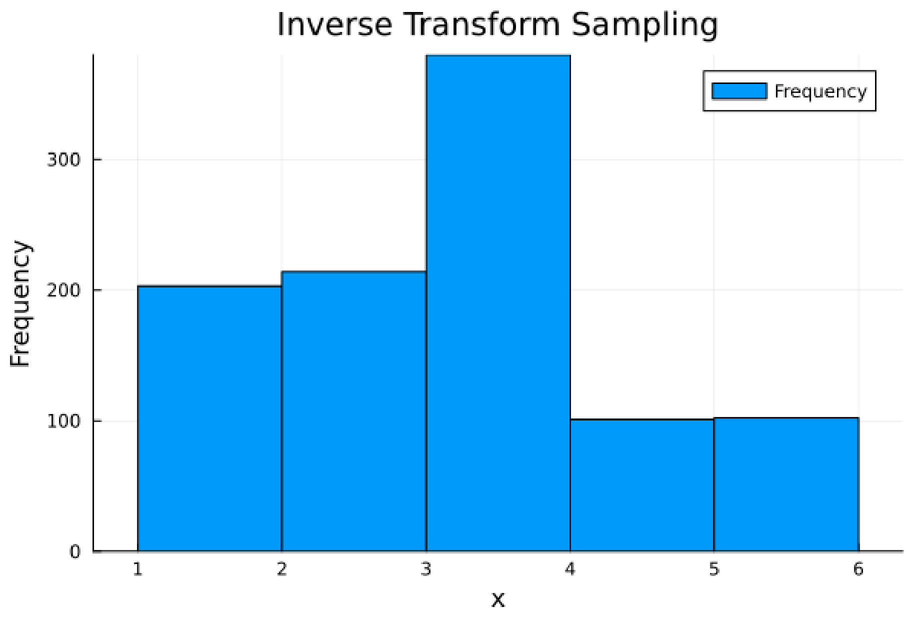


Figure 1: Sample PMF

We can visually observe the correctness of the method. The true PMF should have heights $200, 200, 400, 100, 100$, and this is very well-approximated. Multiple independent runs of the experiment confirm the validity, at least visually.

We can implement this as a running sum of probability values, and check when a a uniform pseudo-random number is larger than the running sum, then return the index of the iteration since our space is exactly the natural numbers till n. However, since we want to generate a large number of samples, it makes sense to pre-compute the cdf using the pdf values (which can be implemented easily in Julia using the function cumsum. Then for each independent generation, we only need to traverse the cdf array and compare sizes as long as u is larger than the cdf value. See code appendix for details.

Next, we want to implement the **Acceptance-Rejection Method** (Exercise 3.6.5), and demonstrate the algorithm using a simple example (which we would not use in practice). Consider the problem of generating a sample from the discrete, finite random variable X with pmf given by

$$f(X = 1) = 0.3$$
$$f(X = 2) = 0.2$$
$$f(X = 3) = 0.35$$
$$f(X = 4) = 0.15$$

For any discrete probability mass function defined on the set $\{1, 2, \ldots, n\}$, the most obvious candidate for the pmf g of the variable Y is the discrete uniform distribution which assigns the probabilities

$$g(Y = i) = \frac{1}{n}$$

for $i = 1, \ldots, n$. This has the same space as (X, f), and it is easy to compute using inverse transformations; in fact, the inverse function reduces to an elementary form. It is also in a general sense alike to an arbitrary discrete distribution. The formula for the inverse function, i.e. to generate $y \in Y$ reduces to

$$y = \text{Floor}(4u) + 1$$

for some u from the uniform distribution, since $n = 4$, and

$$g(Y = i) = \frac{1}{4}.$$

Note that since the value of g is identically $\frac{1}{4}$, our computations become even simpler, as we will see. Also note that since we are assuming that we generate y with an inverse transformation, we first need an independent u from the one in the AR-algorithm to generate y. We then need to compute

$$c^* = \sup \frac{f}{g} = \max_{1 \leq i \leq 4} \frac{f(i)}{1/4} = 4 \max_{1 \leq i \leq 4} f(i) = 4(0.35) = 1.4$$

and we can choose $c = 1.4$. We finally get the simplified algorithm

1. Generate u_1 from $U(0, 1)$ and set $y = \text{Floor}(4u_1) + 1$
2. Generate u_2 from $U(0, 1)$
3. If
   $$u_2 \leq \frac{f(y)}{1.4 \cdot \frac{1}{4}} = \frac{f(y)}{0.35},$$
   or if,
   $$0.35 \times u_2 \leq f(y)$$
   set $X = y$. Otherwise return to step 1.

I chose the multiplication in the last step because it is usually better optimized than division. We again generate 1000 samples from the AR algorithm and plot them on a histogram to visually substantiate the algorithm:

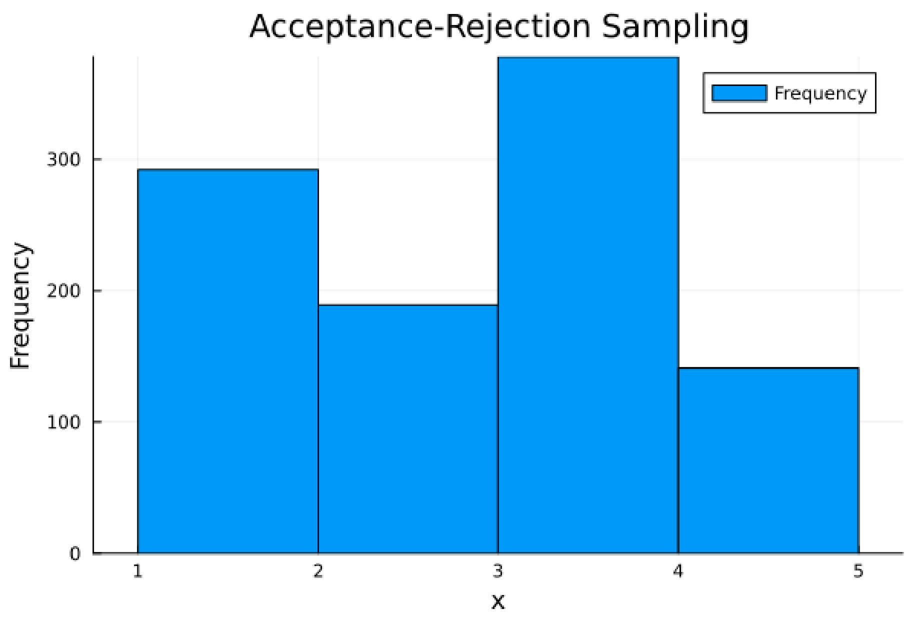


Figure 2: Sample PMF

The true PMF has heights $300, 200, 350, 150$ - we have a very accurate sample again, albeit for a very simple example. The implementation is obvious from the pseudo-code above - I wrote one function to generate numbers from the uniform discrete distribution g and another function to run until a value is accepted (no exception handling was done to safeguard against runtime errors for this simple example). See code appendix for details.

It also makes sense to calculate the time complexity of the Acceptance-Rejection algorithm - since it is probabilistic, we can compute the average or expected complexity $\mathbb{E}[T]$, where T is the number of Floating Point Operations i.e. FLOPS $(\pm, \times, \div)$ and comparisons (checking if a number is less than another) that need to be computed to simulate a value from X(we ignore any operations in initialization and assignments $a := b$, and assume that generating a uniform random number is worth 3 FLOPS). We count the total operations step wise:

- $+3$ operations for generating u_1, $+1$ for a multiplication and $+1$ for an addition in calculating y. We can ignore Floor computation since this probably has a low-level C or math intrinsic implementation in Julia.
- $+3$ for generating u_2.
- $+1$ for a multiplication, and then $+1$ for a comparison (not FLOP)
- Algorithm now terminates with some probability.

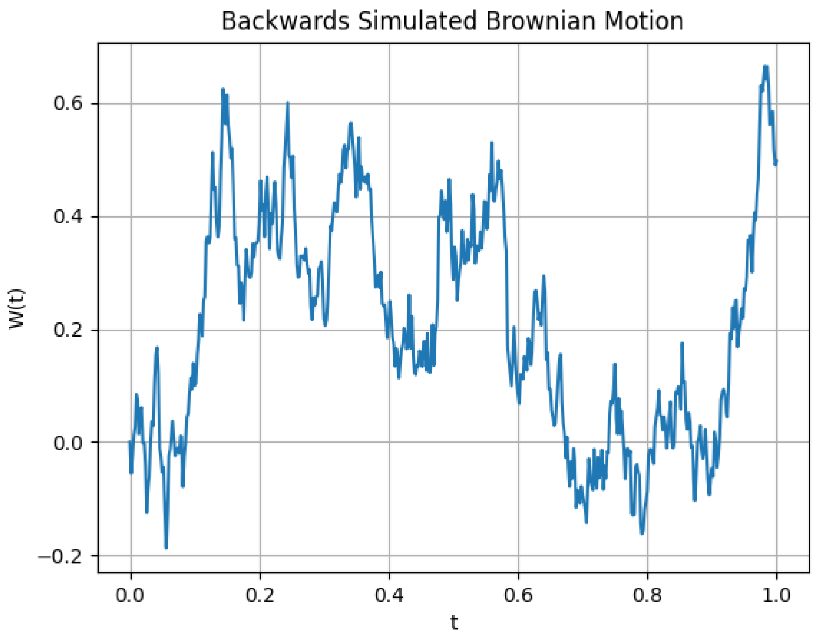
- We get 10 operations in total in an acceptance state.

By Theorem 3.3, the average number of iterations to have an acceptance state is $c = 1.4$. Then, if we have 10 operations per step, we get

$$\mathbb{E}[T] = \mathbb{E}[\text{iterations} \mid \text{acceptance}] \times (\text{computations per iteration}) = 1.4 \times 10 = 14$$

so we get an expected number of 14 computations for the acceptance-rejection algorithm. I ran the code thrice in Julia for this example to get the count of loops, and then multiplied by 10 and then averaged to get an estimate of the average time complexity, and got the values $14.1, 14.01, 14.06$, which corroborates our result. The code is available in the appendix.

We also implement our Backwards Brownian Motion algorithm in python just to visually ascertain correctness of our recursive equations:



Code Appendix

Inverse Transformation

```
using Plots

# Parameters
pvals = [0.2, 0.2, 0.4, 0.1, 0.1]
xvals = Float64.(1:5)

# Pre-compute cdf to avoid recomputing the aggregate sum per sample
cdf = cumsum(pvals)

# Inverse transform
function inv_transform_cdf(cdf::Vector{Float64}, x::Vector{Float64})
    u = rand()  # Xoshiro256++ (Default)
    # Default rand() function used to be MT19937
    # Now Xoshiro256++ is used in the newer versions of Julia

    for i in 1:length(cdf)
        if u < cdf[i]
            return x[i]
        end
    end

    return x[end]
end

# Generate samples
samples = [inv_transform_cdf(cdf, xvals) for _ in 1:1000]

# Only 5 bins | fast!
hist = histogram(samples;
    bins = 5,
    normalize = false,
    label = "Frequency",
    xlabel = "x",
    ylabel = "Frequency",
    title = "Inverse Transform Sampling"
)

# Save to PNG
# savefig(hist, "histogram.png")

display(hist)
readline()
```

Acceptance-Rejection

```
using Plots
using Statistics

# Inverse transform
function discrete(n::Int64)
    u = rand()
    return floor(Int, n * u + 1)
end

function ar(n::Int64, pvals::Vector{Float64}, c::Float64)
    run = true
    counter = 0
    while run
        counter += 1
        y = discrete(n)
        u = rand()
        if (c*u <= pvals[y] )
            run = false
            return y, counter
        end
    end
end


n=4
pvals = [0.3, 0.2, 0.35, 0.15]
xvals = Float64.(1:n)

# Using pmf g(i) = 1/4
c = 0.35;

results = [ar(n, pvals, c) for _ in 1:1000]

samples = [r[1] for r in results]
attempts = [r[2] for r in results]

println("Time Complexity: ", mean(10 * attempts))

# Only 5 bins | fast!
hist = histogram(samples;
    bins = 4,
    normalize = false,
    label = "Frequency",
    xlabel = "x",
    ylabel = "Frequency",
    title = "Acceptance-Rejection Sampling"
)

# Save to PNG
# savefig(hist, "histogram2.png")
display(hist)
readline()
```

Backward Brownian Motion (Python)

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

def backwards_brownian_bridge(n, T=1.0, seed=None):
    if seed is not None:
        np.random.seed(seed)

    t = np.linspace(0, T, n+1)  # t_0 to t_n
    W = np.zeros(n+1)          # W[0] = W(t_0) = 0
    Z = np.random.randn(n)     # Standard normals Z_1 to Z_n

    # Step 1: Generate W(t_n) ~ N(0, T)
    W[-1] = np.sqrt(T) * Z[0]

    # Step 2: Recursive simulation backward using the lemma
    for i in range(n-1, 0, -1):  # simulate W(t_i) backward
        t_ni = t[i]
        t_nip1 = t[i+1]
        t_0 = 0

        mu = W[i+1] * (t_ni - t_0) / (t_nip1 - t_0)
        sigma2 = (t_nip1 - t_ni) * (t_ni - t_0) / (t_nip1 - t_0)
        W[i] = mu + np.sqrt(sigma2) * Z[n - i]

    return t, W

# Run and plot
t, W = backwards_brownian_bridge(n=500, T=1.0, seed=42)

plt.plot(t, W)
plt.title("Backwards Simulated Brownian Motion")
plt.xlabel("t")
plt.ylabel("W(t)")
plt.grid(True)
plt.show()
```