Simulating random vectors, stochastic processes, and simulation of discrete events

Outline

1. Multivariate distributions
2. Multivariate normal distribution
3. Poisson processes
4. Gaussian processes
5. Discrete event simulation: queues, inventories, collective risk, and repair problem

Multivariate distributions

Generate random vector with known conditional cdfs

To simulate a random vector whose components have specified marginal and conditional distributions, we utilize their cumulative distribution functions (cdfs). Consider a random vector (X_1, X_2, X_3). The joint cdf of this vector can be expressed as:

$$F_{X_1,X_2,X_3}(x_1,x_2,x_3) = F_{X_1}(x_1) \cdot F_{X_2|X_1}(x_2|x_1) \cdot F_{X_3|X_1,X_2}(x_3|x_1,x_2)$$

where $F_{X_1}(x_1)$ is the marginal cdf of X_1, and $F_{X_2|X_1}(x_2|x_1)$, $F_{X_3|X_1,X_2}(x_3|x_1,x_2)$ are the conditional cdfs of X_2 and X_3 given X_1 and (X_1, X_2), respectively.

Algorithm To generate a realization of this vector, follow these steps:

1. **Generate** X_1: Sample of the distribution with cdf F_{X_1}.
2. **Generate** X_2: Given the value of X_1, sample X_2 from the conditional distribution with cdf $F_{X_2|X_1}(x_2|x_1)$.
3. **Generate** X_3: Given the values of X_1 and X_2, sample X_3 from the conditional distribution with cdf $F_{X_3|X_1,X_2}(x_3|x_1,x_2)$.
4. **Construct the Vector**: Combine these values into the vector $\mathbf{X} = (X_1, X_2, X_3)^t$.

Multinomial distribution

In the context of a multinomial distribution, the marginal and conditional distributions of each component are binomial distributions. This property allows us to simulate a multinomial vector effectively.

Consider a random vector $\mathbf{X} = (X_1, X_2, X_3, X_4)^t$ that follows a multinomial distribution $M(n, (p_1, p_2, p_3, 1 - p_1 - p_2 - p_3))$.

Algorithm The steps to simulate this Multinomial distribution are as follows:

1. **Generate** X_1: Sample X_1 from a binomial distribution with parameters n (number of trials) and p_1 (probability of success), denoted $X_1 \sim Bin(n,\ p_1)$.
2. **Generate** X_2: Sample X_2 from a binomial distribution with updated parameters $(n - X_1)$ and $p_2/(1 - p_1)$, denoted $X_2 \sim Bin(n - X_1,\ p_2/(1 - p_1))$.
3. **Generate** X_3: Similarly, sample X_3 from a binomial distribution with parameters $(n - X_1 - X_2)$ and $p_3/(1 - p_1 - p_2)$, denoted $X_3 \sim Bin(n - X_1 - X_2,\ p_3/(1 - p_1 - p_2))$.
4. **Determine** X_4: Compute X_4 as the remaining trials, that is, $X_4 = n - X_1 - X_2 - X_3$.
5. **Construct the Vector**: Combine these values into the vector $\mathbf{X} = (X_1, X_2, X_3, X_4)^t$.

Alternatively, use `rmultinom(n, size, prob)`.

Random permutations

A permutation π of $S = \{1, 2, \ldots, n\}$ is any of the $n!$ sortings of S.

That is $\pi(i) \in S$ for $i \in S$ and $\pi(i) \neq \pi(j)$ for $i, j \in S$ with $i \neq j$.

We can use the conditional cdfs argument to generate a random permutation from $S = \{1, 2, 3, 4, 5\}$ as follows.

Algorithm

1. Set $\pi(1)$ random number from S.
2. Set $\pi(2)$ random number from $S \backslash \{\pi(1)\}$
3. Set $\pi(3)$ random number from $S \backslash \{\pi(1), \pi(2)\}$
4. Set $\pi(4)$ random number from $S \backslash \{\pi(1), \pi(2), \pi(3)\}$
5. Set $\pi(5) \in S \backslash \{\pi(1), \pi(2), \pi(3), \pi(4)\}$ and stop.

If A and B are two sets, $A \backslash B$ is the set formed by the elements of A that are not in B.

```
k = 20
perm = (1:k)

for(i in 1:(k-1)){
   interchange = ceiling(runif(1)*(k-i+1))
   aux = perm[i]
   perm[i] = perm[interchange + (i-1)]
   perm[interchange + (i-1)] = aux
}

perm
```

```
[1]  3 13 16  7  4 15  9  8 18 11 12  2 19  6 10  5  1 17 20 14
```

Alternatively

```
rank(runif(10))
sample(10, x=1:10, replace=F)
```

```
[1]  1 10  9  5  2  3  7  8  4  6
[1] 10  9  4  6  7  1  8  2  3  5
```

In Rcpp:

```
library(Rcpp)

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

// [[Rcpp::export]]
IntegerVector permutation(int k) {
    // Create a vector of integers from 1 to k
    IntegerVector perm = seq_len(k);

    for(int i = 0; i < k - 1; i++) {
        // Adjust for C++ 0-indexing
        int interchange = std::ceil(R::runif(0, k - i)) - 1;
        int aux = perm[i];
        perm[i] = perm[interchange + i];
        perm[interchange + i] = aux;
    }

    return perm;
}
'
)

permutation(20)
```

```
[1] 14 13 15 20 12  6 11  5 18  4 19  2  8  7 10  3 17 16  9  1
```

Multivariate normal distribution

Bivariate Normal Generation (Box-Mullers Polar Method)

The Box-Muller transformation is an important technique in statistical simulation. It allows for the generation of pairs of independent, standard, normally distributed (zero mean, unit variance) random variables given a source of uniformly distributed random numbers. This method is particularly useful in the context of simulating scenarios in which a normal distribution is assumed.

Algorithm The transformation algorithm below can be used to simulate a bivariate standard normal random variable. The steps are as follows:

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

2. Set $(X_1, X_2) = \sqrt{-2\log U_1}(\cos(2\pi U_2), \sin(2\pi U_2))$

   Stop.

Here, (X_1, X_2) are the generated bivariate normal random variables.

- **Representation as a Random Point:** If (X_1, X_2) represents a random point in the plane, then X_1 and X_2 are independent standard normal random variables. This means that each has a mean of 0 and a variance of 1.

- **Polar Coordinates:** The polar coordinates of the point (X_1, X_2) are given by the radius $\sqrt{X_1^2 + X_2^2}$ and the angle $\arctan 2(X_2, X_1)$.

  Here, $\arctan 2(\cdot, \cdot)$ is the two-argument arc-tangent function, which returns the angle whose tangent is the quotient of its two arguments.

- **Distance to Origin:** The distance of the point (X_1, X_2) to the origin, which is $\sqrt{X_1^2 + X_2^2}$, follows a square root of a chi-square distribution with 2 degrees of freedom. This is equivalent to the square root of an exponential random variable with rate parameter $\lambda = \frac{1}{2}$.

- **Random Angle Distribution:** The angle $\arctan 2(X_2, X_1)$ represents a uniformly distributed random angle in the interval $(0, 2\pi)$. This implies that the angle is equally likely to be any value between the 0 and 2π radians.

Bivariate normal distribution

The bivariate normal distribution is a two-dimensional extension of the normal distribution. Let $(Z_1, Z_2) \sim N_2(0, \mathbf{I}_2)$ be a standard bivariate normal random variable with mean vector zero and identity covariance matrix. We can transform this standard normal rv to another bivariate normal rv (X_1, X_2) using the following linear transformations:

$$
\begin{aligned}
X_1 &= \mu_1 + \sigma_1 Z_1 \\
X_2 &= \mu_2 + \sigma_2 \left(Z_1 \rho + Z_2 \sqrt{1 - \rho^2} \right)
\end{aligned}
$$

Here, μ_1 and μ_2 are the means of X_1 and X_2 respectively, σ_1 and σ_2 are the standard deviations and ρ is the correlation coefficient between X_1 and X_2.

Properties:

- **Expectation and Variance of X_1:**
  - The expectation of X_1:
  $$E[X_1] = E[\mu_1 + \sigma_1 Z_1] = \mu_1$$
  This follows because $E[Z_1] = 0$ and $E[\mu_1 + \sigma_1 Z_1] = \mu_1 + \sigma_1 E[Z_1] = \mu_1$.
  - The variance of X_1:
  $$Var[X_1] = Var[\mu_1 + \sigma_1 Z_1] = \sigma_1^2$$
  Since $Var[a + bX] = b^2 Var[X]$ for constants a and b, and $Var[Z_1] = 1$ (standard normal distribution).
- **Expectation and Variance of X_2:**
  - The expectation of X_2:
  $$E[X_2] = E\left[\mu_2 + \sigma_2\left(Z_1\rho + Z_2\sqrt{1-\rho^2}\right)\right] = \mu_2$$
  This result comes from the linearity of expectation and the fact that both Z_1 and Z_2 have zero mean.
  - The variance of X_2:
  $$Var[X_2] = Var\left[\mu_2 + \sigma_2\left(Z_1\rho + Z_2\sqrt{1-\rho^2}\right)\right] = \sigma_2^2$$
  This is derived from the properties of variance, considering the independence of Z_1 and Z_2, and the fact that their variances are 1.
- **Covariance between X_1 and X_2:**
$$Cov(X_1, X_2) = E\left[X_1 X_2\right] - E[X_1] \cdot E[X_2] = \rho\sigma_1\sigma_2$$
This is calculated using the definition of covariance and the properties of the standard normal distribution.
- **Joint Distribution:** The joint distribution of (X_1, X_2) is a bivariate normal distribution. This is a consequence of the linear transformation of standard normal variables.

Algorithm for simulating a Bivariate Normal Distribution

1. Generate two independent standard normal random variables Z_1 and Z_2. Standard normal random variables have a mean of 0 and a variance of 1.

2. Transform Z_1 and Z_2 to obtain X_1 and X_2 using the given linear transformations:

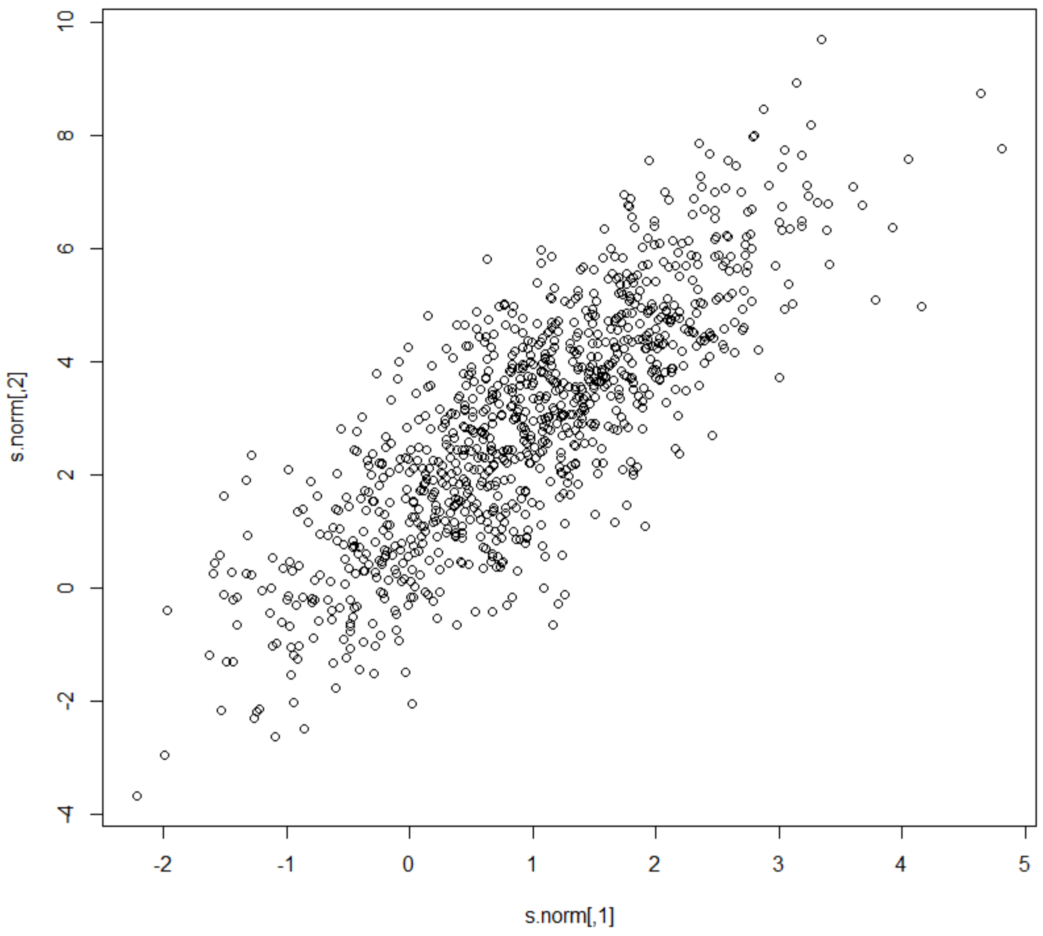
$$X_1 = \mu_1 + \sigma_1 Z_1$$
$$X_2 = \mu_2 + \sigma_2 \left(Z_1 \rho + Z_2 \sqrt{1 - \rho^2} \right)$$

3. The pair (X_1, X_2) generated through this process follows a bivariate normal distribution with the specified parameters.

Note: It is important to understand that the properties of the bivariate normal distribution, such as means, variances, and covariance, are derived from the properties of the standard normal distribution and the linear transformations applied.

The correlation coefficient ρ plays a crucial role in determining the degree of linear relationship between X_1 and X_2. When $\rho = 0$, X_1 and X_2 are independent; when $|\rho| = 1$, they are perfectly linearly related.

```r
mu1 = 1
mu2 = 3
sigma1 = 1
sigma2 = 2
rho =0.8
MC = 1000
set.seed(1)

s.bnorm = matrix(rnorm(2*MC), nrow=2)

m.trans = matrix(c(sigma1, 0, sigma2*rho, sigma2*sqrt(1-rho^2)),
ncol=2, byrow=T)

s.norm = t(m.trans%*%s.bnorm+c(mu1,mu2))

plot(s.norm)
```


s.norm[,2]
s.norm[,1]
-4
-2
0
2
4
6
8
10
-2
-1
0
1
2
3
4
5

Multivariate Normal Distribution

Consider a multivariate normal distribution in a d-dimensional space. This distribution is characterized by a mean vector d of dimensions, denoted as μ, and a covariance matrix $d \times d$, denoted as Σ. The covariance matrix Σ is a symmetric positive definite matrix that describes the variance-covariance structure of the distribution.

Algorithm for Sampling from a Multivariate Normal Distribution

To generate random samples from a d-variate normal distribution, we can use the following algorithm:

1. Determine a matrix A such that $AA^t = \Sigma$. This can be achieved using Cholesky decomposition or other matrix decomposition methods such as eigenvalue decomposition.
2. Generate d independent standard normal random variables, denoted as $Z_1, Z_2, \ldots, Z_d$.

   These are univariate normal variables with mean 0 and variance 1.
3. Construct the random vector X by multiplying the matrix A with the vector of standard normal rvs $(Z_1, \ldots, Z_d)^t$, and then adding the mean vector μ.

   Formally, $X = A(Z_1, \ldots, Z_d)^t + \mu$. The resulting vector X is a sample of the multivariate normal distribution.

Cholesky Decomposition Cholesky decomposition is a specific type of matrix decomposition that is particularly useful for symmetric, positive-definite matrices such as the covariance matrix Σ.

This decomposition represents Σ as the product of a lower triangular matrix L and its transpose, that is, $\Sigma = LL^t$.

In practice, when using `R`, the command `chol(sigma)` provides the *upper triangular* matrix L^t. To obtain the lower triangular matrix L, we simply take the transpose of L^t. This matrix L can then be used in the algorithm above to generate samples from the multivariate normal distribution.

The matrix computed in a bivariate example (where $d = 2$) using this method will be equivalent to the one obtained using Cholesky decomposition, ensuring the consistency of the approach to generate multivariate normal samples.

```
m.trans
```

```
     [,1] [,2]
[1,]  1.0  0.0
[2,]  1.6  1.2
```

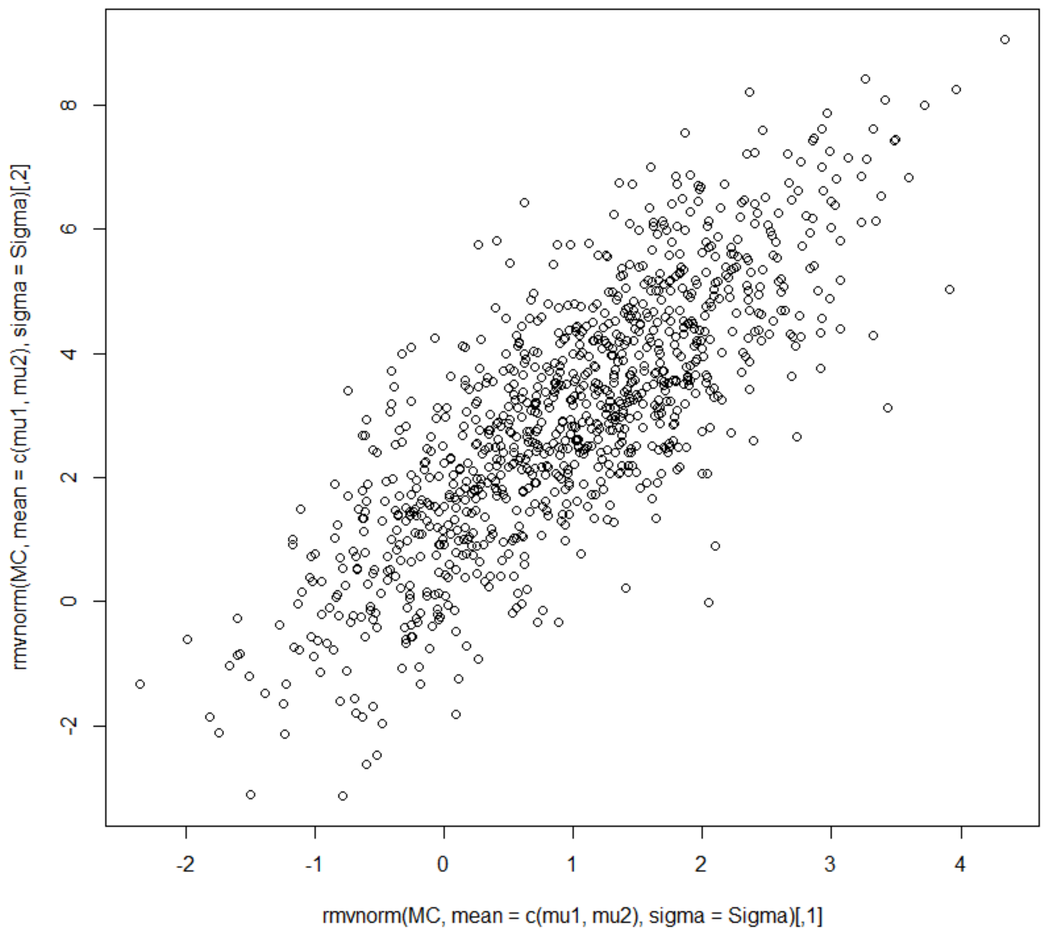
```
Sigma = matrix(c(sigma1^2, rho*sigma1*sigma2,
rho*sigma1*sigma2, sigma2^2), ncol=2, byrow=T)

# Use C = t(chol(Sigma))
t(chol(Sigma))
```

```
     [,1] [,2]
[1,]  1.0  0.0
[2,]  1.6  1.2
```

Multivariate normal with `mvnorm`

Alternatively, we can use `rmvnorm(n, mean, sigma)` from package `mvtnorm`.

```
library(mvtnorm)

mu1 = 1
mu2 = 3
sigma1 = 1
sigma2 = 2
rho = 0.8

Sigma = matrix(c(sigma1^2, rho*sigma1*sigma2,
rho*sigma1*sigma2, sigma2^2), ncol=2, byrow=T)

MC = 1000
set.seed(1)

plot(rmvnorm(MC, mean=c(mu1,mu2), sigma=Sigma))
```



rmvnorm(MC, mean = c(mu1, mu2), sigma = Sigma)[,2]
rmvnorm(MC, mean = c(mu1, mu2), sigma = Sigma)[,1]
-2
-1
0
1
2
3
4
-2
0
2
4
6
8

Poisson Processes

A **Poisson process** is a specific type of **counting process**, which is a fundamental concept in the field of stochastic processes. A counting process $\{N(t) : t \geq 0\}$ is characterized by the following properties:

- $N(t) \geq 0$ for all $t \geq 0$. This means that the process counts events, and hence the count is always non-negative.
- $N(t)$ is an integer for all $t \geq 0$. As it counts occurrences, the values it takes must be whole numbers.
- If $t_2 \geq t_1$, then $N(t_2) \geq N(t_1)$. This property implies that the count can only stay the same or increase as time progresses; it never decreases, reflecting the nature of counting events over time.

Furthermore, for any $t_2 > t_1$, the increment $N(t_2) - N(t_1)$ represents the number of events that occur in the time interval $[t_1, t_2]$. This increment is a measure of the events that have occurred in a specific period.

A counting process can be characterized in two ways:

1. By the sequence of random variables $\{T_1, T_2, \ldots\}$, where T_i represents the time at which the i-th event occurs. These are the instances when the count increases.
2. By the sequence of interarrival times $\{W_1, W_2, \ldots\}$, where $W_i = T_i - T_{i-1}$ represents the time between the $(i-1)$-th and i-th events. This perspective focuses on the duration between consecutive events.

In the specific case of a **Poisson process**, these properties are further specialized:

- The process has **independent increments**: The number of events that occur in disjoint time intervals is independent of each other.
- The process has **stationary increments**: The probability distribution of the number of events in any time interval depends only on the length of the interval, not on its position in time.

- The probability of a single event occurring in a small interval of length Δt is approximately $\lambda\Delta t$, where λ is the constant rate of the process.
- The probability of two or more events occurring in this small interval is negligible (that is, it tends to zero faster than Δt as Δt tends to zero).

These characteristics make the Poisson process a suitable model for various real-world phenomena that involve counting the occurrences of events over time, especially when these events happen independently of each other and at a constant average rate.

Homogeneous Poisson process

If the increments are independent and rv $N(t)$ follows a Poisson distribution, then $\{N(t) : t \geq 0\}$ is a Poisson process.

The following counting process is a (homogeneous) Poisson process with the rate (intensity) $\lambda > 0$ if

- $N(0) = 0$.
- $N(t_2) - N(t_1)$ is independent of $N(s_2) - N(s_1)$ for every $0 \leq t_1 < t_2 \leq s_1 < s_2$ (independent increments).
- $N(t_2) - N(t_1) \sim Pois(\lambda(t_2 - t_1))$ (stationary increments with a Poisson distribution).

We have $W_i \sim Exp(\lambda)$ for every i and independent, and

$$T_i = \sum_{r=1}^{i} W_r \sim Gamma(i, \lambda).$$

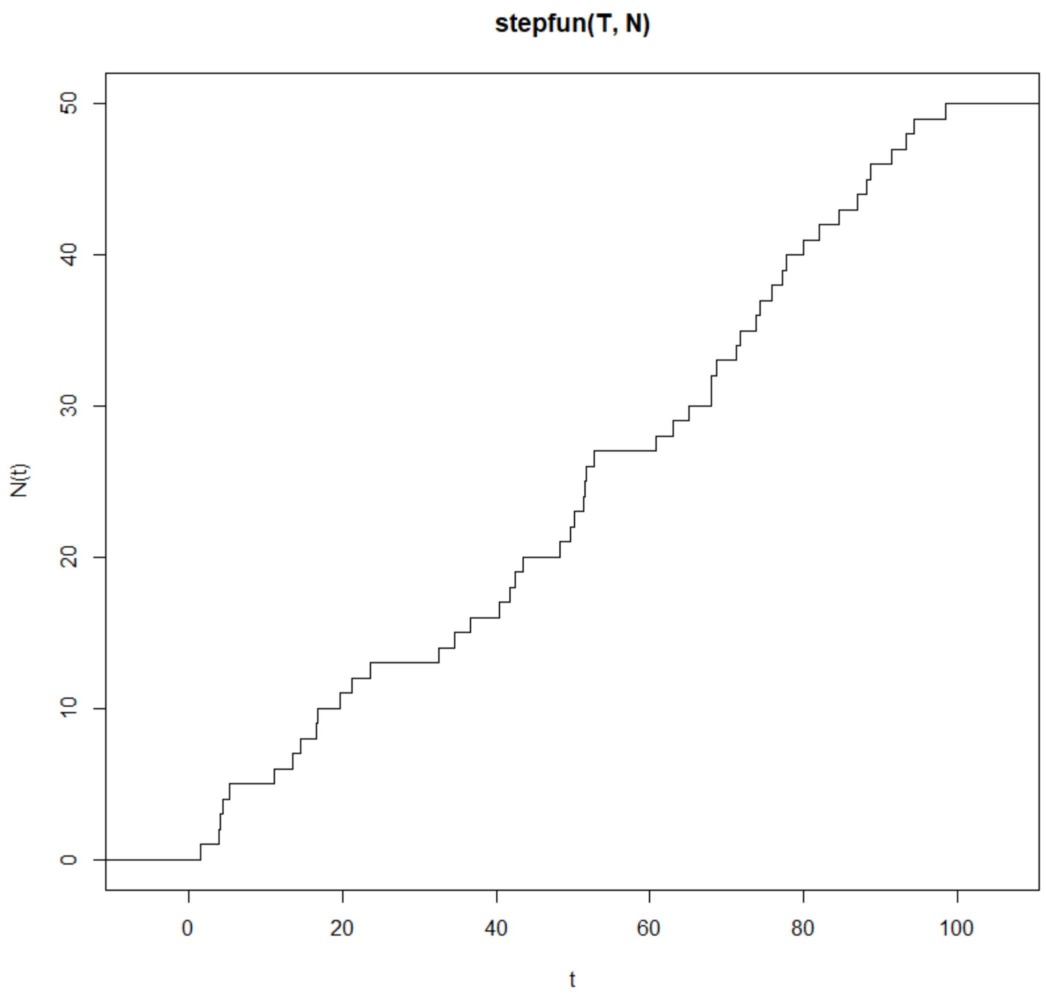
Simulation of homogeneous Poisson process until k-th event

Algorithm

1. Set $T_0 = 0$.
2. Generate $W_1, W_2, \ldots, W_k$ Exponential rvs with parameter λ.

3. For $1 \le i \le k$, set $T_i = T_{i-1} + W_i$ and stop.

```
k = 50
lmbd = 0.5
set.seed(1)

T = cumsum(rexp(k, rate=lmbd))
N = 0:k

plot(stepfun(T,N), do.points=F, xlab="t", ylab="N(t)")
```



Simulation of homogeneous Poisson process in $[0, t_l]$

For a homogeneous Poisson process with rate λ on an interval $[0, t_\ell]$, two facts hold:

1. The total number of points (events) $N(t_\ell)$ in $[0, t_\ell]$ is a Poisson random variable with parameter λt_ℓ.

2. Conditionally on $N(t_\ell) = k$, the (unordered) arrival times are distributed as k i.i.d. uniform random variables on $[0, t_\ell]$.

Using these properties, we can simulate the process as follows:

1. **Generate the total number of events $k = N(t_\ell)$** as a Poisson random variable with parameter λt_ℓ. This captures how many arrival times fall in $[0, t_\ell]$.

$$k \sim \text{Poisson}(\lambda t_\ell).$$

2. **Generate uniform random variables.** Once k is known, draw k independent uniform$(0, 1)$ random variables:

$$U_1, U_2, \ldots, U_k \sim \text{Uniform}(0, 1).$$

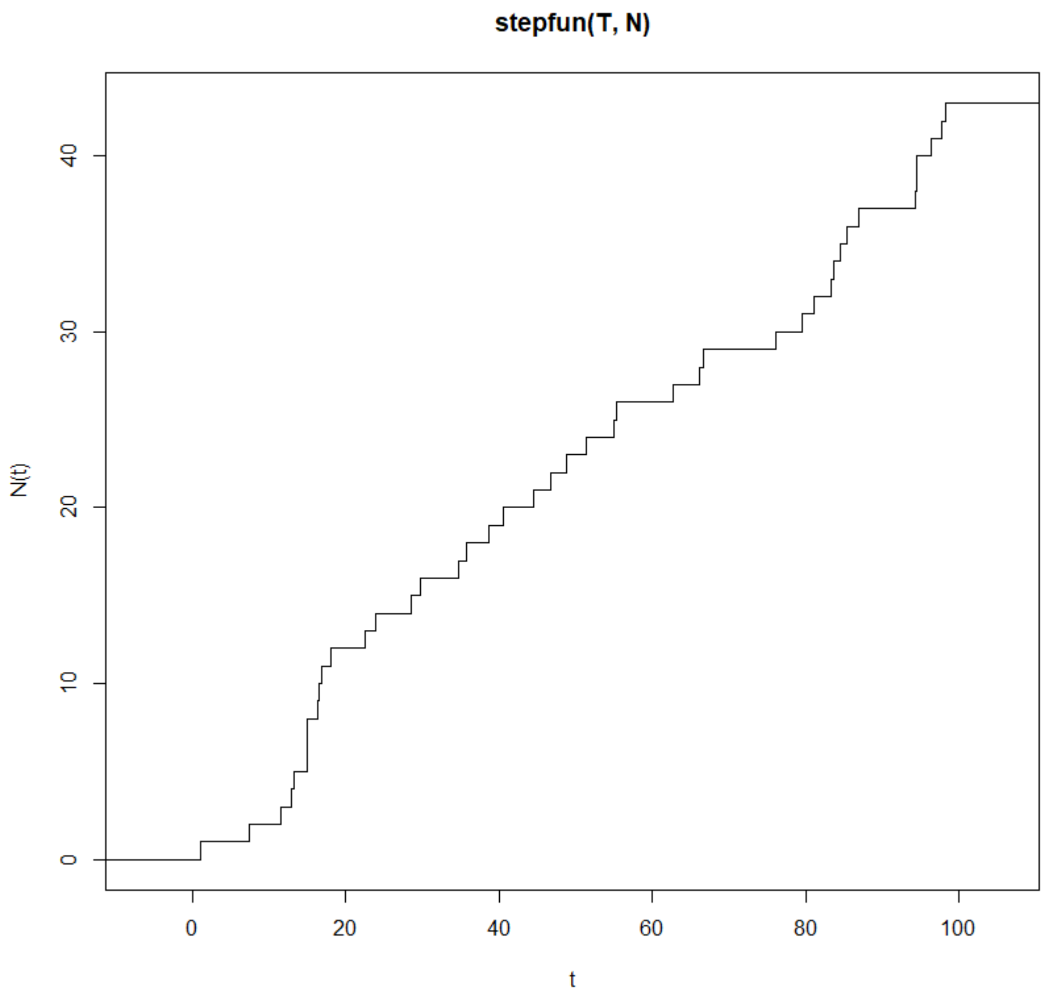
3. **Sort the uniform random variables.** Arrange the sample in ascending order:

$$U_{(1)} < U_{(2)} < \cdots < U_{(k)}.$$

4. **Map the sorted uniform variables into arrival times.** Scale each sorted uniform variable by t_ℓ to obtain the arrival (event) times in $[0, t_\ell]$:

$$T_i \;=\; t_\ell U_{(i)}, \quad \text{for } i = 1, 2, \ldots, k.$$

```
tl = 100      # t_\ell = 100
lmbd = 0.5    # lambda = 0.5
set.seed(2)

k = rpois(1, lambda = lmbd*tl)
u = runif(k)
T = tl*sort(u)
N = 0:k

plot(stepfun(T, N), do.points=FALSE, xlab="t", ylab="N(t)")
```


stepfun(T, N)
N(t)
0
10
20
30
40
t
0
20
40
60
80
100

Non-homogeneous Poisson process

A counting process is a non-homogeneous Poisson process with intensity function $\lambda : \mathbb{R}^+ \to \mathbb{R}^+$ if

- $N(0) = 0$.
- $N(t_2) - N(t_1)$ is independent of $N(s_2) - N(s_1)$ for every $0 \leq t_1 < t2 \leq s1 < s2$ (independent increments).
- $N(t_2) - N(t_1) \sim Pois(\int_{t_1}^{t_2} \lambda(s)ds)$ (nonstationary increments with a Poisson distribution).

We denote $M(t) = \displaystyle\int_0^t \lambda(s)ds$ and $\lambda_m \geq \lambda(s)$ for every $0 \leq s \leq t_l$, where λ_m, is a constant rate assumed to be greater than or equal to the maximum rate of the actual non-homogeneous process $\lambda(t)$ for all t.

Simulation of non-homogeneous Poisson process until k-th event

The distribution function of the rv time until next event from time t is

$$F_t(r) = P\left(W_i \leq r | T_{i-1} = t\right) = 1 - P\left(W_i > r | T_{i-1} = t\right) =$$

$$1 - P\left(N(t+r) - N(t) = 0\right) = 1 - \exp\left(-\int_t^{t+r} \lambda(s)ds\right)$$

We can use the inverse transform technique to simulate the first k events of a non-homogeneous Poisson process as:

Algorithm

1. Set $T_0 = 0$.
2. Generate $U_1, U_2, \ldots, U_k$ random numbers in $(0,1)$.
3. For $1 \leq i \leq k$, set $T_i = T_{i-1} + F_{T_{i-1}}^{-1}(U_i)$ and stop.

```r
# Function to calculate the integral of lambda(s) from t to t + r
lambda_integral = function(t, r, lambda) {
  integrate(lambda, lower = t, upper = t + r)$value
}

# Inverse of the cdf Ft
Ft_inverse = function(u, t, lambda) {
  optimize(function(r) abs(u - (1 - exp(-lambda_integral(t, r, lambda)))),
           interval = c(0, 100))$minimum
}

# Simulating the nonhomogeneous Poisson process
simulate_poisson_process = function(k, lambda) {
  T = numeric(k)  # Storing the times of the events
  T[1] = 0        # Setting T0 to 0

  for (i in 2:k) {
    U = runif(1)  # Generate a random number in (0, 1)
    T[i] = T[i - 1] + Ft_inverse(U, T[i - 1], lambda)
  }

  return(T)
}

# Example lambda function
lambda_function = function(s) {
  # Define the lambda function here
  return(0.5 + 0.5 * sin(s))
}

# Simulating the first k events
k = 5  # Number of events to simulate
event_times = simulate_poisson_process(k, lambda_function)

# Output the simulated event times
print(event_times)
```

```
[1] 0.0000000 0.5363732 0.6435212 1.1561227 1.3572236
```

Simulation of a Non-Homogeneous Poisson Process in $[0, T]$

The following steps outline the algorithm for simulating a non-homogeneous Poisson process over a fixed time interval $[0, T]$, given a time-dependent intensity function $\lambda(t)$:

1. **Compute the cumulative intensity function:** Compute the integral of the intensity function over the interval $[0, T]$,

$$M(T) = \int_0^T \lambda(s)\, ds.$$

This represents the expected number of events occurring in the interval.

2. **Generate the total number of events:** Sample a Poisson-distributed random variable N with mean $M(T)$. This variable represents the total number of events in the interval $[0, T]$.

3. **Generate uniform random numbers:** Generate N independent uniform random numbers U_i over the interval $[0, M(T)]$, for $i = 1, \dots, N$.

4. **Transform uniform random numbers into event times:** For each U_i, determine the corresponding event time T_i by solving the equation:

$$\int_0^{T_i} \lambda(s)\, ds = U_i.$$

   If an analytical inverse is unavailable, numerical methods such as root-finding techniques can be used.

5. **Sort the event times:** Arrange the event times T_i in ascending order to ensure chronological order.

```r
# Define the intensity function lambda(t)
lambda = function(t) { 5 + sin(t) }

# Define the time interval
T = 1

# Compute the cumulative intensity function M(T) over [0, T]
M_T = integrate(lambda, lower = 0, upper = T)$value

# Generate the total number of events N from a Poisson distribution
# with mean M(T)
N = rpois(1, M_T)

# Generate uniform random numbers over [0, M(T)]
U = runif(N, min = 0, max = M_T)

# Define the inverse of the cumulative intensity function M^-1(U)
# Perform numerical inversion

M_inv = function(u) {
  # Solve for t such that the integral of lambda from 0 to t equals u
  uniroot(function(t) integrate(lambda, 0, t)$value - u, lower=0, upper=T)$root
}

# Transform U_i to T_i using the inverse function M^-1(U_i)
```

```r
T_i = sapply(U, M_inv)

# Sort the event times T_i
T_i_sorted = sort(T_i)

# Print the sorted event times
print(T_i_sorted)
```

```
[1] 0.5052944 0.5604225 0.7016878 0.7654629 0.8202144 0.8560005
```

Simulation of non-homogeneous Poisson process until k-th event (thinning)

The previous algorithm requires an explicit expression for F_t^{-1} (for every t).

An alternative approach is the thinning technique, which consists of rejecting some of the simulated events.

Algorithm

1. Set $T_0 = 0$, $T^* = 0$, $i = 0$.
2. Generate $X \sim \text{Exp}(\lambda_m)$, an exponential random variable with rate λ_m.
3. Set $T^* = T^* + X$.
4. Generate a uniform random number $U \sim U(0,1)$.
5. If $U < \frac{\lambda(T^*)}{\lambda_m}$, then set $i = i + 1$ and $T_i = T^*$.
6. If $i = k$, stop.
7. Otherwise, go to Step 2.

Example in R

```r
# Define the rate function of the non-homogeneous Poisson process
lambda = function(t) {
  return(0.1 + 0.05 * sin(t))
}

# Maximum rate, lambda_m, for the homogeneous Poisson process
lambda_m = 0.15  # Should be >= max(lambda(t))
```

```
# Initialize variables
T0 = 0
T_star = 0
i = 0
k = 5  # Number of events to simulate
T_i = numeric(k)  # Vector to store event times

set.seed(123)

# Simulation loop
while(i < k) {
  X = rexp(1, rate = lambda_m)
  T_star = T_star + X

  U = runif(1)

  if(U < lambda(T_star) / lambda_m) {
    i = i + 1
    T_i[i] = T_star
  }
}

print(T_i)
```

```
[1] 14.48341 14.69393 15.37973 45.08581 50.17064
```

More about Poisson and counting processes

Superposition principle of Poisson processes. If $\{N_1(t)\}$ and $\{N_2(t)\}$ are two independent Poisson processes with respective intensity functions λ_1 and λ_2 then $\{N_1(t)+N_2(t)\}$ is a Poisson process with intensity function $\lambda_1 + \lambda_2$.

– **Simulation**: Generate the path of a Poisson process with an intensity function $\lambda_1 + \lambda_2$.

How do we simulate such a combined Poisson process?

We can outline a method to determine to which original process ($\{N_1(t)\}$ or $\{N_2(t)\}$) a particular event in the combined process $\{N_1(t) + N_2(t)\}$ belongs.

When an event occurs at time t in the combined process, you can assign it to $\{N_1(t)\}$ with a probability proportional to $\lambda_1(t)$ relative to the total intensity $\lambda_1(t) + \lambda_2(t)$.

So, the probability that an event at time t belongs to process $\{N_1(t)\}$ is given by:

$$\frac{\lambda_1(t)}{\lambda_1(t) + \lambda_2(t)}$$

This method allows for the distinction between events arising from the first and the second Poisson processes in the simulation of the combined process.

It effectively assigns each event to its originating process on the basis of the relative intensity of the two processes at the time the event occurs. This is an important step in simulations, where it is necessary not only to generate events according to the total intensity, but also to distinguish between the contributions of each underlying process.

Compound Poisson process. We can associate an independent rv X_i to each event occurring in a Poisson process $N(t)$ and set the compound Poisson process

$$S(t) = \sum_{k}^{N(t)} X_k.$$

– **Simulation**: Generate the path of the Poisson process $N(t)$, at each event, generate a rv with the distribution of X and update $S(t)$.

Mixed Poisson process. The intensity (rate) is not a function of time, but a structural rv instead, so $N(t) \sim P(\Lambda_t)$ with Λ rv.

– **Simulation**: Generate an observation of Λ and simulate the homogeneous Poisson process.

Doubly stochastic Cox process. The intensity function is itself a stochastic process.

– **Simulation**: Generate a path of the stochastic process $\Lambda(t)$ and simulate the non-homogeneous Poisson process.

Renewal process. The times between events are not exponentially distributed.

– **Simulation**: As the simulation of the homogeneous Poisson process up to the k th event, but not the time between two consecutive events is not exponentially distributed.

Gaussian Processes

A **Gaussian process** is a type of stochastic process, denoted as $X(t)$, where t represents time or another variable. The defining characteristic of a Gaussian process is that its marginal distributions are normal, or Gaussian. This property has several important implications.

- **Normal Distribution of Random Vectors:** For any finite collection of time points $t_1, t_2, \ldots, t_n$, the random vector $(X(t_1), X(t_2), \ldots, X(t_n))^T$ is distributed as a multivariate normal. This means that for any such collection of time points, the joint distribution of the process at these times is a multivariate normal distribution.

- **Mean Function:** A Gaussian process is completely characterized by its *mean function* and its *autocovariance function.* The mean function, denoted $\mu_{X(t)}$, is defined as the expected value of the process at time t:

  $$\mu_{X(t)} = E[X(t)],$$

  where $E[\cdot]$ denotes the expectation. This function gives the average behavior of the process at each time point.

- **Autocovariance Function:** The autocovariance function, denoted $C_X(t_1, t_2)$, measures the covariance between different time points in the process. It is defined as:

  $$C_X(t_1, t_2) = \mathrm{Cov}(X(t_1), X(t_2)),$$

  where $\mathrm{Cov}(\cdot, \cdot)$ represents covariance. This function describes how the values of the process at different times are related to each other.

In summary, Gaussian processes are a fundamental concept in probabilistic modeling and have applications in fields such as machine learning, statistics, and engineering. They provide a flexible framework for modeling unknown functions, time series, and spatial data.

Brownian motion

Brownian motion, also known as the Wiener process, is a fundamental concept in stochastic processes and finds applications in various fields such as physics, finance, and mathematics. A process $B(t)$ is defined as a Brownian motion if it satisfies the following properties:

- **Initial Condition:** The process starts at zero, that is, $B(0) = 0$. This is a standard assumption, setting the initial state of the process.

- **Independent Increments:** For any sequence of times $0 \leq t_1 < t_2 < \cdots < t_n$, the increments $(B(t_2) - B(t_1)), (B(t_3) - B(t_2)), \ldots, (B(t_n) - B(t_{n-1}))$ are independent of each other. This property means that the future evolution of the process does not depend on its past, making it a Markov process.

- **Normal Increments:** For any $s > 0$, the increment $B(t+s) - B(t)$ follows a normal distribution with mean 0 and standard deviation $\sqrt{s}$, denoted as $B(t+s) - B(t) \sim N(0, \sqrt{s})$. This indicates that the process has Gaussian increments.

- **Continuous Paths:** The function $B(t)$ is continuous in t. This means that the Brownian paths are smooth without jumps, a fundamental property for modeling continuous phenomena.

A **generalized Brownian motion** introduces drift and volatility parameters. Specifically, a Brownian motion with drift μ (representing the systematic, directional component) and volatility σ (representing the scale of random fluctuations) is defined by the following stochastic differential equation:

$$X(t) = \mu t + \sigma B(t),$$

where $X(t)$ represents the value of the process at time t. The distribution of $X(t)$ at any time t is normal with mean μt and standard deviation $\sigma\sqrt{t}$, denoted as $X(t) \sim N(\mu t, \sigma\sqrt{t})$.

Simulating a Brownian motion

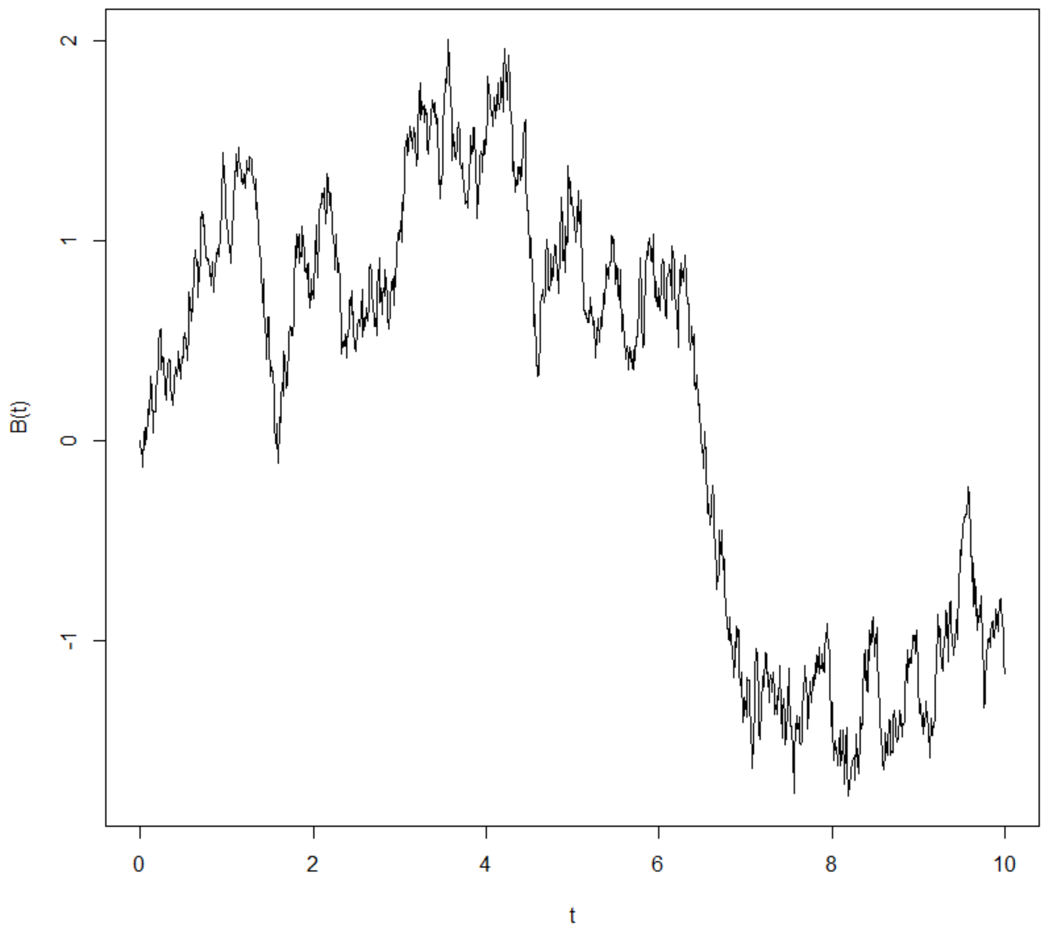
Algorithm for Simulation: To simulate a Brownian motion over a discrete set of times $t_1, t_2, \ldots, t_n$, one can follow these steps:

1. Initialize the process at zero: Set $B_0 = 0$.

2. For each time step i, generate a normally distributed random variable $X_i \sim N(0, \sqrt{t_i - t_{i-1}})$. This variable represents the increase of the process over the interval $[t_{i-1}, t_i]$.

3. Update the process: Set $B_{t_i} = B_{t_{i-1}} + X_i$. This step adds the increment to the previous value of the process, thereby simulating its path over time.

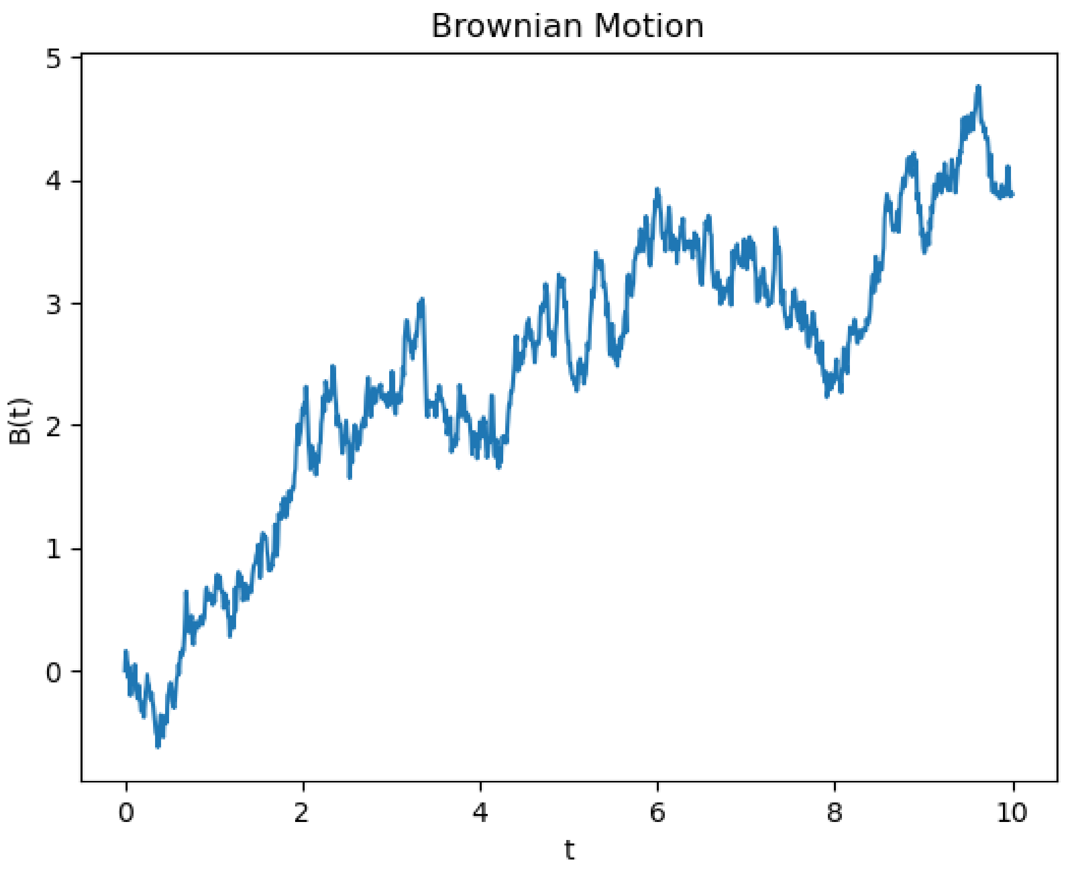
```r
n = 100
tl = 10
t = seq(0, tl, by=1/n)

set.seed(1)
B = append(0, cumsum(rnorm(n*tl, mean=0,sd=1/sqrt(n))))

plot(t, B, type="l", ylab="B(t)")
```



In Python

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

# Set parameters
n = 100
tl = 10

# Create time sequence
t = np.linspace(0, tl, num=n*tl+1)

# Set random seed
```

```
np.random.seed(1)

# Generate Brownian motion
B = np.append(0, np.cumsum(np.random.normal(0, 1/np.sqrt(n), n*tl)))

# Plot
plt.plot(t, B)
plt.xlabel('t')
plt.ylabel('B(t)')
plt.title('Brownian Motion')
plt.show()
```



Geometric Brownian Motion

Geometric Brownian motion (GBM) is a continuous-time stochastic process commonly used in financial mathematics to model the dynamics of stock prices. Mathematically, GBM is defined as follows:

$$S(t) = S(0)\exp\left[\left(\mu - \frac{\sigma^2}{2}\right)t + \sigma B(t)\right],$$

where:

- $S(t)$ represents the price of the stock at time t.

- $S(0)$ is the initial price of the stock at time $t = 0$.
- μ is the expected return rate of the stock.
- σ is the volatility of the stock, representing the standard deviation of the stock returns.
- $B(t)$ is a standard Brownian motion or Wiener process, which models the random component of stock price movements.

The key feature of GBM is that it models stock prices such that they remain positive and vary continuously over time. The exponential term in GBM ensures that the stock prices cannot become negative, which is a realistic feature for stock prices.

Distribution of Geometric Brownian Motion

An important characteristic of GBM is that its univariate marginals follow a log-normal distribution. This is due to the exponential function applied to a normally distributed variable. Specifically, if a random variable X is normally distributed, i.e. $X \sim N(\mu, \sigma^2)$, then e^X follows a log-normal distribution. This relationship can be written as:

$$\exp(X) \sim \text{LogNormal}(\mu, \sigma^2).$$

Expectation and Variance

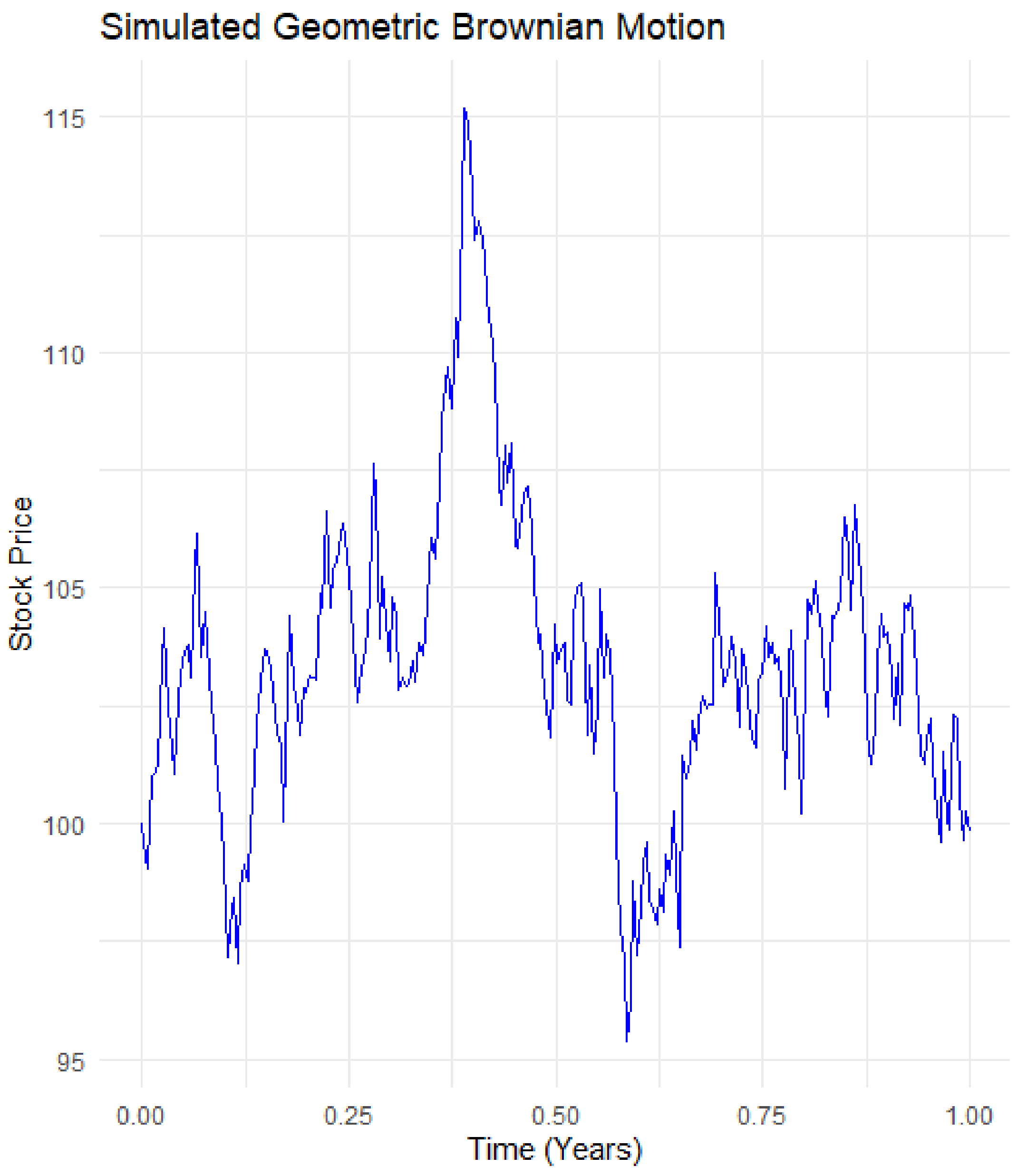
For a log-normally distributed variable, expectations and variance are given as follows. Let $X \sim N(\mu, \sigma^2)$, then for $Y = e^X$, we have:

$$\begin{aligned} \mathbb{E}[Y] &= \exp\left(\mu + \frac{\sigma^2}{2}\right), \\ \text{Var}[Y] &= \left(\exp(\sigma^2) - 1\right) \exp\left(2\mu + \sigma^2\right). \end{aligned}$$

Here, $\mathbb{E}[Y]$ and $\text{Var}[Y]$ denote the expectation and variance of Y, respectively. These formulas are derived from the properties of the log-normal distribution and are crucial in financial mathematics for the assessment of risk and the pricing of financial derivatives.

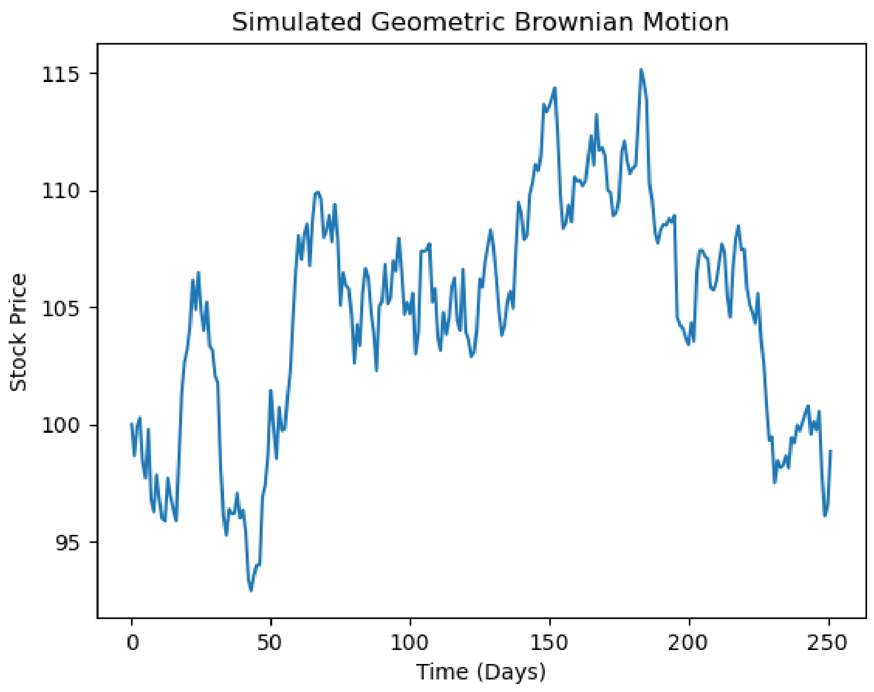
```r
library(ggplot2)

# Set parameters
S0 = 100     # Initial stock price
mu = 0.05    # Drift coefficient
```

```r
sigma = 0.2 # Volatility
T = 1       # Time horizon in years
n = 252     # Number of steps (trading days in a year)

# Time increment (dt)
dt = T / n

# Initialize the stock price vector
S = numeric(n)
S[1] = S0

# Simulate the stock price path
set.seed(123)
for (t in 2:n) {
  epsilon = rnorm(1)
  S[t] = S[t-1] * exp((mu - 0.5 * sigma^2) * dt + sigma * sqrt(dt) * epsilon)
}

# Create a data frame for plotting
time = seq(0, T, length.out = n)
stock_df = data.frame(time, Stock_Price = S)

# Plot
ggplot(stock_df, aes(x = time, y = Stock_Price)) +
  geom_line(color = "blue") +
  theme_minimal() +
  ggtitle("Simulated Geometric Brownian Motion") +
  xlab("Time (Years)") +
  ylab("Stock Price")
```


Simulated Geometric Brownian Motion
Stock Price
115
110
105
100
95
0.00
0.25
0.50
0.75
1.00
Time (Years)

In Python

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

S0 = 100     # Initial stock price
mu = 0.05    # Drift coefficient
sigma = 0.2 # Volatility
T = 1        # Time horizon in years
n = 252      # Number of steps (trading days in a year)

# Time increment
dt = T / n

# Simulating the stock price path
np.random.seed(123)
S = np.zeros(n)
S[0] = S0

for t in range(1, n):
    epsilon = np.random.randn()
    S[t] = S[t - 1] * np.exp((mu - 0.5 * sigma**2) * dt + sigma * np.sqrt(dt) *
                                             epsilon)

# Plotting
plt.plot(S)
plt.title("Simulated Geometric Brownian Motion")
plt.xlabel("Time (Days)")
plt.ylabel("Stock Price")
plt.show()
```



Discrete Event Simulation

Discrete event simulation models a system as a discrete sequence of events in time. This type of simulation is distinct from continuous simulation, where changes occur continuously over time. Discrete event simulation is particularly useful for systems where changes happen at distinct points in time and the system remains static between these points.

The fundamental concept in this simulation involves simulating the *times* at which the events occur and some random amounts associated with the *events* themselves. This approach allows for the analysis of complex systems by breaking them down into manageable discrete events.

Relevant Variables in Discrete Event Simulation

- **Time variable**, t: This represents the elapsed amount of (simulated) time. It is a crucial variable as it tracks the progression of the simulation.
- **Counter variables**: These are used to count the number of times certain events have occurred in time t. They are essential for understanding the frequency and occurrence of events within the simulation.
- **System state variable**, SS: This variable represents the state of the system at time t. It encapsulates all relevant information about the current state of the system and is updated whenever an event occurs.

Whenever an event occurs, these variables are updated to reflect the new state of the system. This update process is central to the operation of discrete event simulations.

Single and Multiple Server Queuing System

In a **single and multiple server queuing system**, customers or entities arrive at a service station and are serviced by one or more servers. The dynamics of this system can be complex, depending on the arrival process of customers, the service mechanism, and the discipline of the queue.

Model Description:

Customers arrive at a service station following a non-homogeneous Poisson process with intensity function $\lambda(t)$. This arrival process allows the rate of customer arrival to vary over time.

Upon arrival, each customer's actions depend on the state of the server:

- If the server is free, the customer is immediately entered into service.
- If the server is busy, the customer joins a waiting queue.

The service mechanism is as follows:

- When a server finishes serving a customer, it checks the queue.
- If the queue is not empty, the server begins serving the next customer, typically the one who has been waiting the longest if a *First-In, First-Out (FIFO)* queue discipline is used.
- If the queue is empty, the server remains free until the next customer arrives.

The service time for each customer is a random variable, independent of other service times, and follows a known probability distribution. This randomness is a key aspect of the system that introduces variability into the simulation.

A final time t_l is set, after which no additional arrivals are allowed in the system. This parameter allows for the simulation to have a finite duration.

Relevant Variables in Queuing Systems

- **Time variable**, t: Similarly to discrete event simulation, this tracks the progression of time in the system.
- **Counter variables**, N_A and N_D: These represent the number of arrivals (N_A) and departures (N_D) by time t. They are crucial for understanding the flow of customers through the system.
- **System state variables**, n: This denotes the number of customers in the system at time t. It is a key variable for assessing the load on the system and the waiting queue.

Events in the Queuing System:

The primary events in this system are **arrivals** and **departures**, occurring at times t_A and t_D respectively. These events are fundamental to the dynamics of the queuing system.

Output Variables of Interest:

The output variables to consider in this system include the following:

- A_i: This represents the arrival time of the i-th customer. Tracking these times is essential for analyzing the distribution and frequency of customer arrivals.

These output variables are key to evaluating the performance of the queueing system. They allow for the evaluation of various aspects, such as customer waiting times, service efficiency, and overall system throughput.

In both discrete event simulation and queuing systems, the simulation process involves generating events according to specified probabilistic rules, updating system variables as these events occur, and recording relevant output variables for subsequent analysis. The insights gained from these simulations can be used to make informed decisions about system design, capacity planning, and operational strategies.

In an M/M/1 queue:

- The first M stands for memoryless inter-arrival times (exponential distribution).
- The second M indicates memoryless service times (also an exponential distribution).
- The 1 denotes a single server in the system.

```r
library(queueing)

# Define the parameters
lambda = 1/4     # Arrival rate (tasks per minute)
# on average, a new task arrives every 4 minutes

mu = 1/3         # Service rate (tasks per minute)
# on average, one task is serviced every 3 minutes

# Create the queue model
model = NewInput.MM1(lambda, mu)

# Analyze the model
results = QueueingModel(model)

# You can also analyze specific metrics like average number
# in the system, average time in the system, etc.

cat("Average number of tasks in the system: ", L(results), "\n")
# both waiting and being serviced
```

```
cat("Average time a task spends in the system: ",
W(results), " minutes\n")
# waiting time plus service time
```

```
Average number of tasks in the system:  3
Average time a task spends in the system:  12  minutes
```

```
# c = 3 servers
model = NewInput.MMC(lambda=5, mu=10, c=3)

# Analyze the model
results = QueueingModel(model)

cat("Average number of tasks in the system: ", L(results), "\n")

cat("Average time a task spends in the system: ",
W(results), " minutes\n")
```

```
Average number of tasks in the system:  0.5030303
Average time a task spends in the system:  0.1006061  minutes
```

Inventory Model

This section describes the inventory model for a retail shop that stocks a specific type of good. The model incorporates various parameters and variables to capture the dynamics The model describes how the shop manages its stock levels, responds to customer demand, and handles ordering and financial transactions.

1. Shop Operations and Customer Demand

- **Product Price:** The product is sold at a constant price r per unit.
- **Customer Arrivals:** Customers arrive according to a Poisson process with rate λ. This means that, on average, λ customers arrive per unit time.
- **Demand per Customer:** The number of units each customer demands is a random variable with distribution G, capturing the variability in demand.

2. Inventory Management and Ordering Policy

- **Inventory Level:** The current stock level is denoted by S and is continuously monitored.

- **Replenishment Policy:** An (s_1, s_2) ordering policy is used:
  - When S falls below the threshold s_1, an order is placed to bring the stock up to s_2.
  - The order quantity is $s_2 - S$ units.
- **Ordering Cost:** The cost of ordering y units is given by a function $c(y)$, which can include fixed costs or volume discounts.
- **Lead Time:** There is a delivery delay of L time units between ordering and receiving the goods.
- **Holding Cost:** The shop incurs a holding cost of h per unit per time unit for maintaining inventory.

3. Customer Service and Sales

- If a customer demands more units than what is available, the shop sells the available stock and loses the sale for the remaining units. This situation results in **lost sales**.

4. Model Variables and Events

Variables:

- **Time, t:** Represents the current time.
- **Cost Counters:**
  - $C(t)$: Total ordering cost incurred up to time t.
  - $H(t)$: Total holding cost incurred up to time t.
  - $R(t)$: Total revenue earned up to time t.
- **System States:**
  - $x(t)$: Inventory on hand at time t.
  - $y(t)$: Inventory that has been ordered but not yet delivered at time t.

Events:

- The two main events in this model are customer arrivals and order deliveries.

- Let t_c be the time of the next customer arrival and t_o be the time of the next order delivery.
- The model updates its state variables at the minimum of t_c and t_o, i.e., when the next event occurs.

R Example: Simulating the Inventory Model

Here:

1. Initialize the system with a starting inventory.
2. Simulate customer arrivals using a Poisson process.
3. Update inventory based on customer demand.
4. Apply the (s_1, s_2) ordering policy when needed.

```r
# Set parameters
r = 10              # Price per unit
lambda = 1          # Average customer arrival rate per time unit
s1 = 20             # Reorder threshold
s2 = 50             # Order up-to level
L = 2               # Lead time for order delivery
h = 0.5             # Holding cost per unit per time unit

# Simulation parameters
simulation_time = 50  # Total simulation time
set.seed(123)          # For reproducibility

# Initialize system state
current_time = 0
inventory = s2        # Start with full inventory
pending_orders = list()  # List to hold orders (delivery time and quantity)
total_revenue = 0
total_order_cost = 0
total_holding_cost = 0

# Define ordering cost function (for simplicity, assume cost per unit)
c = function(y) { 2 * y }  # Cost is 2 per unit ordered

# Function to simulate customer demand (random, e.g., uniform between 1 and 5)
customer_demand = function() {
  sample(1:5, 1)
}

# Simulation loop
```

```r
while (current_time < simulation_time) {
  # Determine next event times: customer arrival or order delivery
  next_customer_time = current_time + rexp(1, rate=lambda)

  # Next order delivery time: if no pending orders, set to Inf
  if (length(pending_orders) == 0) {
    next_delivery_time = Inf
  } else {
    next_delivery_time = min(sapply(pending_orders,
    function(order) order$delivery_time))
  }

  # Determine next event time
  next_event_time = min(next_customer_time, next_delivery_time)

  # Calculate holding cost over the time interval
  delta_time = next_event_time - current_time
  total_holding_cost = total_holding_cost + h * inventory * delta_time

  # Update current time
  current_time = next_event_time

  # Check if the event is an order delivery
  if (abs(current_time - next_delivery_time) < 1e-8) {
    # Process order delivery(s)
    delivered_orders = which(sapply(pending_orders, function(order)
      abs(order$delivery_time - current_time) < 1e-8))
    for (i in delivered_orders) {
      inventory = inventory + pending_orders[[i]]$quantity
    }
    # Remove delivered orders
    pending_orders = pending_orders[-delivered_orders]
  } else {
    # Process customer arrival
    demand = customer_demand()
    sale = min(inventory, demand)
    inventory = inventory - sale
    total_revenue = total_revenue + sale * r
    # If demand > inventory, the extra demand is lost (lost sale)
  }

  # Check if inventory is below the reorder threshold
  if (inventory < s1 && length(pending_orders) == 0) {
    order_quantity = s2 - inventory
    delivery_time = current_time + L
    # Place an order (delivery will occur after L time units)
    pending_orders[[length(pending_orders) + 1]] = list(
      quantity = order_quantity,
      delivery_time = delivery_time
    )
    # Add ordering cost
    total_order_cost = total_order_cost + c(order_quantity)
  }
}

# Display results
cat("Total Revenue:", total_revenue, "\n")
cat("Total Ordering Cost:", total_order_cost, "\n")
cat("Total Holding Cost:", total_holding_cost, "\n")
cat("Final Inventory Level:", inventory, "\n")
```

```
Total Revenue: 1900
Total Ordering Cost: 320
Total Holding Cost: 692.3218
Final Inventory Level: 20
```

Comments

- **Parameter Setup:** The code sets the selling price r, customer arrival rate λ, reorder thresholds s_1 and s_2, lead time L, and holding cost h. It also defines a simple ordering cost function $c(y)$ which multiplies the ordered quantity by a unit cost.
- **System Initialization:** The simulation starts with full inventory (s_2) and initializes variables to track time, revenue, ordering cost, and holding cost.
- **Event Simulation:** The simulation loop runs until the designated simulation time is reached. At each iteration, it determines whether the next event is a customer arrival or an order delivery.
- **Event Handling:**
  - **Customer Arrival:** When a customer arrives, a random demand is generated. The shop sells the available stock up to the demand amount, and any excess demand is lost.
  - **Order Delivery:** When an order is delivered, the inventory is updated by adding the ordered quantity.
- **Reorder Check:** After processing an event, the code checks whether the inventory has dropped below s_1. If so, and if there are no pending orders, it places a new order to replenish the stock to s_2.
- **Cost Accumulation:** Throughout the simulation, holding costs are accumulated based on the inventory level and elapsed time.

Collective Risk Model

The collective risk model is widely used in insurance mathematics to analyze the financial dynamics of an insurance company over time. It takes into account several random processes:

Claims Process

- Claims occur at random times according to a Poisson process $N_C(t)$ with rate λ.
- Each claim has a random amount X_i drawn independently from a distribution F.

Client Enrolment and Departure

- New clients join following a Poisson process $N_A(t)$ with rate ν.
- Once enrolled, a client's duration in the company is exponentially distributed with rate μ. Departures are modeled by $N_D(t)$.

Revenue and Capital Accumulation

- Each policyholder pays a premium of c per unit time.
- The company starts with n_0 policyholders and an initial capital $a_0 \geq 0$.
- The capital at time t is given by:

$$A(t) = a_0 + c\,t\,(n_0 + N_A(t) - N_D(t)) - \sum_{i=1}^{N_C(t)} X_i.$$

Ruin Probability

The *ruin probability* is the chance that the companys capital falls below zero at any time during a period $[0, t_l]$:

$$\Pr\left(\inf_{0\leq t\leq t_l} A(t) < 0\right).$$

Event Dynamics

At any given time t with n policyholders, the time until the next event (enrolment, claim, or departure) is exponentially distributed with rate:

$$\nu + n\lambda + n\mu.$$

The probabilities for the next event are:

- New enrolment: $\dfrac{\nu}{\nu + n\lambda + n\mu}$.
- Claim occurrence: $\dfrac{n\lambda}{\nu + n\lambda + n\mu}$.
- Policyholder departure: $\dfrac{n\mu}{\nu + n\lambda + n\mu}$.

Output Variable

The main output of interest is an indicator showing whether the companys capital $A(t)$ remains nonnegative throughout the period $[0, t_l]$.

Example Simulation in R

Example that simulates the insurance companys capital over a fixed time period and estimates the ruin probability.

```
set.seed(123)

# Modified Parameters for increased ruin probability
t_l = 10            # simulation time horizon
a0 = 100            # lower initial capital to make ruin more likely
n0 = 20             # initial number of policyholders
c = 10              # premium per unit time per policyholder
lambda = 0.1        # claim rate per policyholder
nu = 1              # enrolment rate
mu = 0.1            # departure rate (increased departure rate)
nSim = 1000         # number of simulations

simulate_capital = function() {
  t = 0
  capital = a0
  n = n0

  while(t < t_l) {
    # Total event rate at time t
    rate = nu + n * (lambda + mu)
    # Time to next event
    delta_t = rexp(1, rate)
    t = t + delta_t

    # Update capital from premium income during delta_t
    capital = capital + c * n * delta_t

    # Determine next event type
    event_prob = runif(1)
    if(event_prob < nu / rate) {
      # New enrolment
      n = n + 1
    } else if(event_prob < (nu + n * lambda) / rate) {
      # Claim occurrence: deduct a claim amount
      # Here, we use a claim distribution with higher variance (mean 50)
      claim_amount = rexp(1, rate = 1/50)
      capital = capital - claim_amount
    } else {
      # Policyholder departure
      if(n > 0) n = n - 1
    }

    # Check for ruin: if capital falls below zero, return FALSE (ruin)
    if(capital < 0) return(FALSE)
  }
  return(TRUE)
}

# Run simulations to estimate ruin probability
results = replicate(nSim, simulate_capital())
ruin_probability = 1 - mean(results)

cat("Estimated Ruin Probability:", ruin_probability, "\n")
```

```
Estimated Ruin Probability: 0.171
```

Comments

- **Parameters:** We define parameters such as the time horizon t_l, initial capital a_0, number of policyholders n_0, premium c, claim rate λ, enrolment rate ν, and departure rate μ.
- **Simulation Function:**
  - The function `simulate_capital` simulates one realization of the process.
  - It updates time t, capital, and number of policyholders n according to the event rates.
  - Premium income is accumulated over time, claims decrease capital, and policyholder numbers change with enrolment and departure events.
  - If at any point the capital falls below zero, the function returns `FALSE` (indicating ruin).
- **Ruin Probability:** By running the simulation many times (e.g., 1000 simulations), we estimate the probability of ruin as one minus the proportion of simulations where the company avoided ruin.

Repair Problem

Consider a system that requires a fixed number of functioning machines to operate correctly. To ensure reliability, spare machines are kept available to immediately replace any machine that fails. When a machine fails, it is sent for repair and, once repaired, it rejoins the spare pool.

Key Concepts and Parameters

- **Time Variable, t:** The continuous time over which the system is observed.
- **State Variable, $r(t)$:** The number of machines that are currently broken and awaiting repair at time t.
- **Total Number of Machines, N:** The sum of functioning machines, machines under repair, and spare machines.
- **Spare Capacity, S:** The maximum number of spare machines available at any given time.
- **Failure Rate, μ:** The rate at which machines break down. This is derived from the cumulative distribution function (CDF) F of the machines' lifetimes.
- **Repair Rate, λ:** The rate at which broken machines are repaired, related to the CDF G of the repair times.

Events in the System

The system dynamics are governed by two main types of events:

1. **Machine Failure:** When a functioning machine fails, it is immediately replaced by a spare machine (if one is available), and the failed machine is sent for repair. In this event, the number of machines waiting for repair, $r(t)$, increases by 1.
2. **Machine Repair:** When a broken machine is repaired, it rejoins the spare pool. Here, $r(t)$ decreases by 1.

Output Variable

The primary output of interest is the time until the system collapses, denoted as T_c. System collapse occurs when a machine fails but no spare is available for replacement. Statistical properties of T_c

(such as the expected value and variance) help in assessing the system's reliability.

Example Simulation in R

Below is an example R script that simulates this repair system. In this simulation:

- Each functioning machine breaks down independently with an exponential distribution (rate μ).
- Each broken machine is repaired independently with an exponential distribution (rate λ).
- The system is initialized with a total of N machines, where a fixed number n are functioning and S are spares (with $N = n + S$).
- The simulation runs until a machine fails when no spare is available, i.e., the system collapses.

```r
set.seed(123)

# Parameters
mu = 0.1          # Failure rate
lambda = 0.5      # Repair rate
n = 5             # Number of functioning machines required for operation
S = 3             # Spare capacity
N = n + S         # Total number of machines

# Initialize state
time = 0
# 'r' is the number of machines under repair (failed)
r = 0

# Event logs for tracking events
event_log = data.frame(time = numeric(), event = character(),
r = integer(), stringsAsFactors = FALSE)

# Initialize failure times for the n functioning machines
# We assume the time until each machine fails is exponentially distributed.
failure_times = rexp(n, rate = mu)
# There are no repair times initially because no machine is broken.
repair_times = c()

# Simulation loop
while(TRUE) {
  # Determine the next failure and repair events
  next_failure_time = if(length(failure_times) > 0) min(failure_times) else Inf
  next_repair_time = if(length(repair_times) > 0) min(repair_times) else Inf
```

```r
# Determine the next event time
next_event_time = min(next_failure_time, next_repair_time)
time = next_event_time

# Check which event occurs first
if(next_failure_time <= next_repair_time) {
  # A failure occurs
  # Identify the machine that failed
  idx = which.min(failure_times)

  # Check if a spare is available: available spares = S - r
  if((S - r) > 0) {
    # Spare available: machine is replaced, and the failed machine
    # goes to repair
    r = r + 1  # Increase the number under repair

    # Log the failure event
    event_log = rbind(event_log, data.frame(time = time,
    event = "failure", r = r))

    # Remove the failed machine's failure time
    failure_times = failure_times[-idx]

    # For the spare machine that replaces it, generate a new failure time
    new_failure_time = time + rexp(1, rate = mu)
    failure_times = c(failure_times, new_failure_time)

    # Generate a repair time for the failed machine
    new_repair_time = time + rexp(1, rate = lambda)
    repair_times = c(repair_times, new_repair_time)
  } else {
    # No spare available: system collapse
    event_log = rbind(event_log, data.frame(time = time,
    event = "collapse", r = r))
    cat("System collapsed at time", time, "\n")
    break
  }
} else {
  # A repair occurs
  # Identify which machine is repaired
  idx = which.min(repair_times)
  r = r - 1  # One machine repaired
  # Log the repair event
  event_log = rbind(event_log, data.frame(time = time,
  event = "repair", r = r))

  # Remove the completed repair time
  repair_times = repair_times[-idx]
  # The repaired machine joins the spare pool (implicitly
  # increasing spare availability)
  }

  # (Optional) Stop the simulation if a maximum simulation time is reached
  if(time > 1e4) {
    cat("Maximum simulation time reached.\n")
    break
  }
}

# Print event log
print(event_log)
```

```
         time    event r
1   0.3157736  failure 1
2   0.5621098  failure 2
3   0.9442282   repair 1
4   2.0147778  failure 2
5   2.3063123  failure 3

... ...

79 86.1668076   repair 1
80 86.8051655  failure 2
81 87.1272210  failure 3
82 87.4221259 collapse 3
```

Comments:

- **Initialization:** The simulation begins by setting the failure and repair rates, the number of functioning machines n, and the spare capacity S. The total number of machines is $N = n + S$.
- **Failure Times:** The failure times for the n functioning machines are generated using an exponential distribution with rate μ.
- **Simulation Loop:** In each iteration:
  1. The code finds the next scheduled failure and repair times.
  2. If the next event is a failure and a spare is available (i.e., $S - r > 0$), the machine is replaced, the failed machine is sent for repair (with a generated repair time), and the failure time for the replaced spare is generated.
  3. If a failure occurs when no spare is available, the system collapses and the simulation stops.
  4. Repair events reduce r by one as the repaired machine rejoins the spare pool.

- **Logging:** Events (failure, repair, collapse) are logged with their occurrence time and the current number of machines under repair.

This simulation provides a simple model of a repair system and tracks the time until system collapse. Analyzing the event log can help in understanding the system reliability and the impact of different parameters on the time to collapse.