Simulations with Jumps

Michael Miller

Applications for Jump Processes

Lévy processes are a class of stochastic processes that are pivotal in the field of mathematical finance, particularly for asset and derivative valuation. They generalize the simpler and more commonly known Brownian motion by including jumps, thus offering a more realistic model of asset price dynamics, which can display sudden and significant changes. Here are some key applications of Lévy processes in finance:

1. Modeling Asset Returns

- **Jump-Diffusion Models**: Lévy processes are used to model asset returns that incorporate both continuous Brownian motion and discontinuous jumps. This approach is more realistic than pure Brownian motion, especially for assets that exhibit sudden large movements due to market events or news releases.
- **Infinite Activity Models**: Certain Lévy processes, such as Variance Gamma or CGMY models, can model asset prices with an infinite number of small jumps occurring in a finite interval, capturing the heavy tails and excess kurtosis often observed in financial data.

2. Option Pricing

- **Exotic and Path-Dependent Options**: Lévy processes are particularly useful in pricing exotic options where the payoff depends on the path of the underlying asset price. For instance, lookback options, Asian options, and barrier options can be more accurately priced using models that incorporate jumps.
- **Realistic Volatility Smiles**: Models based on Lévy processes often better capture the observed skew and smile effects in implied volatilities across different strike prices and expirations, compared to the classic Black-Scholes model which assumes log-normal distribution without jumps.

3. Risk Management

- **Value at Risk (VaR) and Expected Shortfall**: The tail-heavy characteristics of Lévy processes make them suitable for assessing risks of extreme losses, helping financial institutions better estimate potential risks and required capital reserves.
- **Stress Testing**: Lévy models allow for the simulation of extreme market movements, aiding in stress testing and scenario analysis required by regulatory frameworks.

4. Credit Risk and Default Modeling

- **Jump-to-Default Models**: In credit risk management, jump processes can model the risk of sudden default of an entity, which is particularly relevant for credit derivatives like credit default swaps.

5. High-Frequency Trading

- **Market Microstructure Noise**: Lévy processes can model the behavior of asset prices at very high frequencies, where the market displays more apparent jumps or discontinuities, useful for high-frequency trading strategies.

6. Commodity and Energy Markets

- **Price Spikes Modeling**: Commodity prices, particularly energy commodities like electricity or natural gas, often experience sudden and large price spikes which can be effectively modeled using Lévy processes.

7. Real Options Analysis

- **Project Valuation**: The flexibility of Lévy processes in capturing jumps and downtrends is also beneficial for real options analysis, where the value of investing in a project can depend significantly on sudden changes in economic conditions or project parameters.

8. Insurance and Actuarial Science

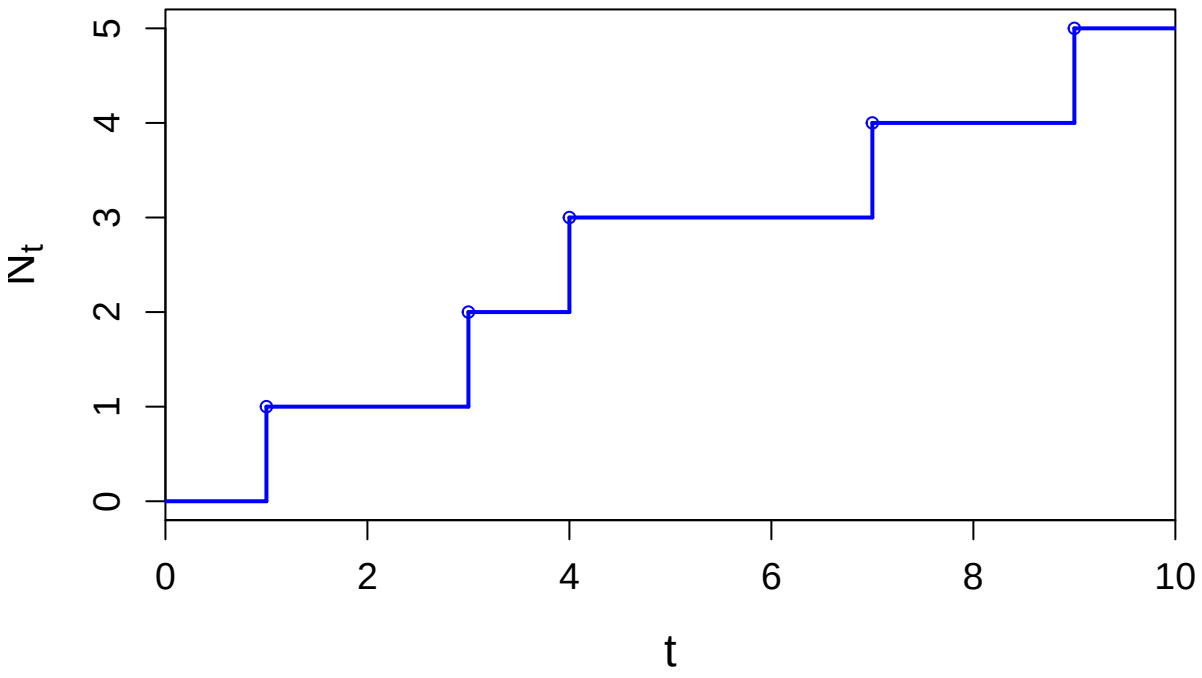
- **Claim Size Distribution**: In insurance, claim sizes may not be well modeled by normal distributions due to the possibility of very large claims, making Lévy processes suitable for such stochastic modeling.

By incorporating jumps, Lévy processes offer a more nuanced and versatile framework for modeling financial phenomena, providing better tools for pricing, hedging, and risk management in complex markets.

Simulations

The counting process $N(t)$ that can be used to model discrete arrival times for jumps.

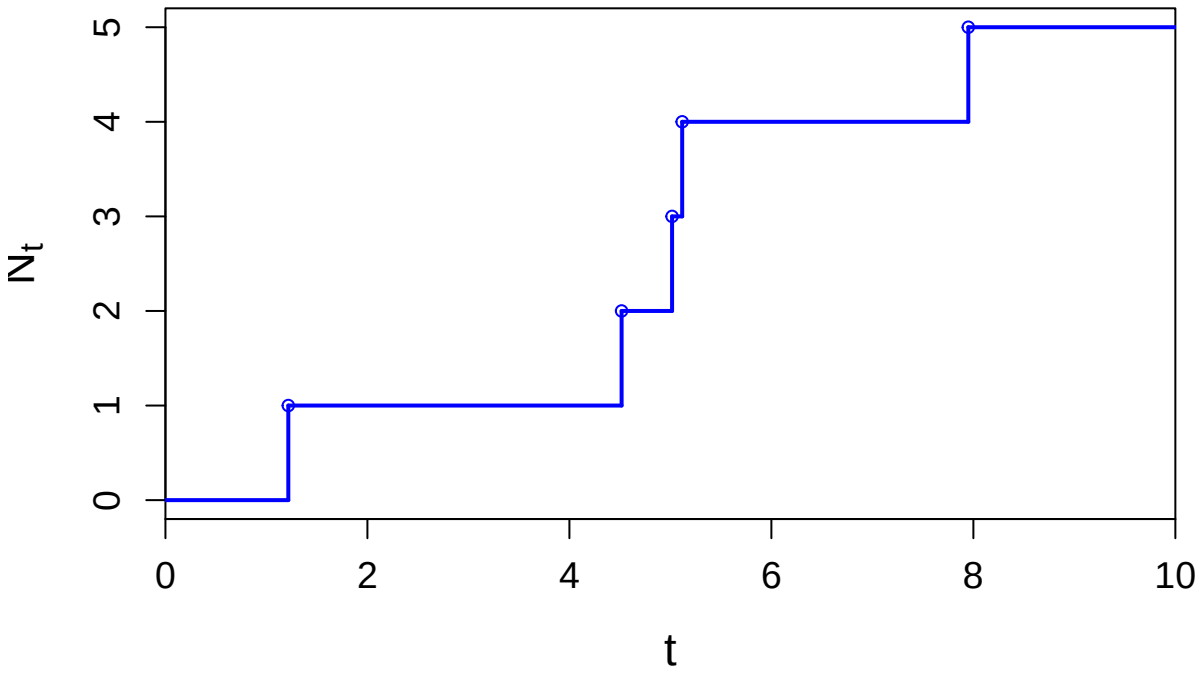
```r
set.seed(123)

# Parameters
T <- 10
Tn <- c(1, 3, 4, 7, 9)

# Plot the step function for the counting process N(t)
plot(stepfun(Tn, c(0, 1, 2, 3, 4, 5)),
     xlim = c(0, T),
     xlab = "t",
     ylab = expression('N'[t]),
     pch = 1,
     cex = 0.8,
     col = 'blue',
     lwd = 2,
     main = "",
     cex.axis = 1.2,
     cex.lab = 1.4,
     xaxs = 'i')
```



Simulation of the standard Poisson Process $N(t)$ path - Short-Term Behavior using Uniform

```r
set.seed(123)

# Parameters
lambda <- 0.6
T <- 10
N <- 1000 * lambda
h <- T / N

# Initialize variables
t <- 0
s <- numeric()

# Simulate the Poisson process
for (k in 1:N) {
  if (runif(1) < lambda * h) {
    s <- c(s, t)
  }
  t <- t + h
}

# Plot the step function for the Poisson process N(t)
plot(stepfun(s, cumsum(c(0, rep(1, length(s))))),
```

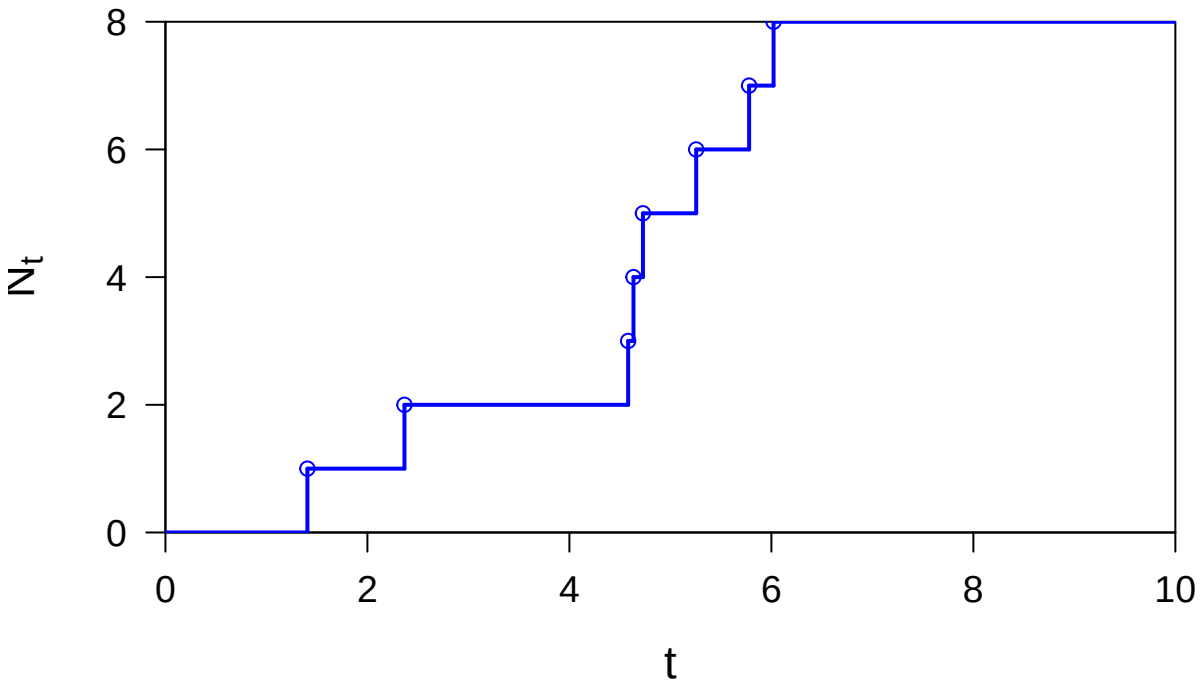
```
     xlim = c(0, T),
     xlab = "t",
     ylab = expression('N'[t]),
     pch = 1,
     cex = 0.8,
     col = 'blue',
     lwd = 2,
     main = "",
     cex.axis = 1.2,
     cex.lab = 1.4,
     xaxs = 'i')
```



Simulation of the standard Poisson Process $N(t)$ path - using Poisson

```
set.seed(123)

lambda <- 0.6
T <- 10
Tn <- c()
n <- 0

S <- 0
while (S < T) {
```

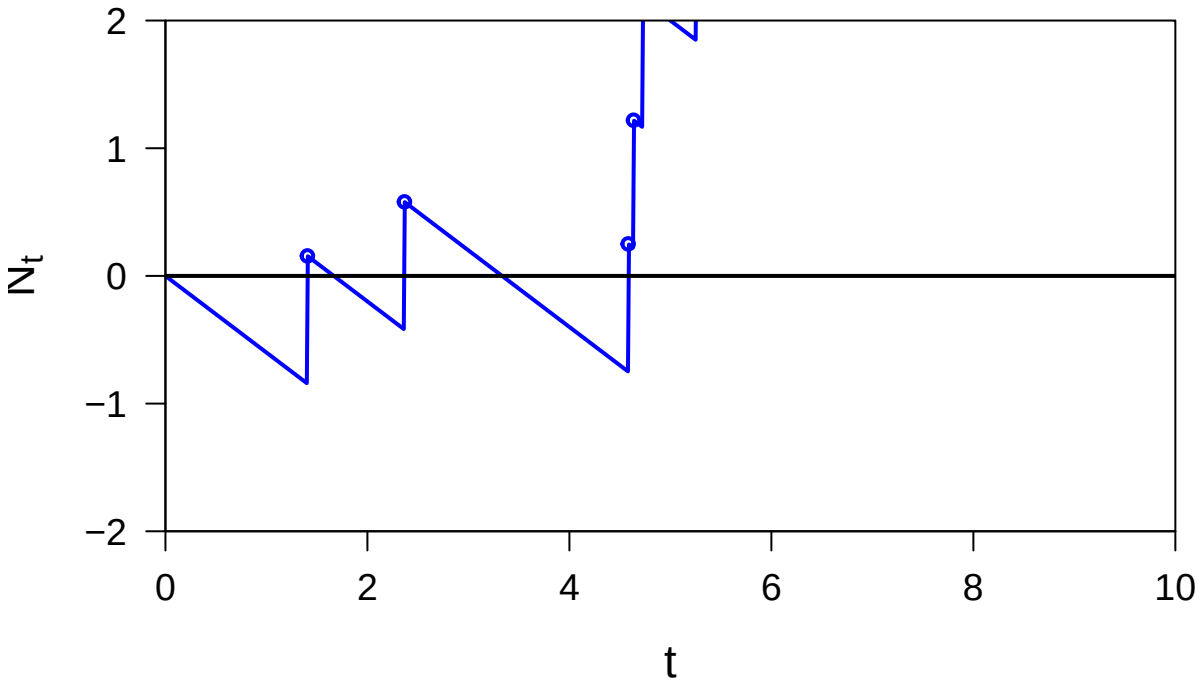
```r
  S <- S + rexp(1, rate = lambda)
  Tn <- c(Tn, S)
  n <- n + 1
}

Z <- cumsum(c(0, rep(1, n)))

plot(
  stepfun(Tn, Z),
  xlim = c(0, T),
  ylim = c(0, 8),
  xlab = "t",
  ylab = expression(N[t]),
  pch = 1,
  cex = 1,
  col = "blue",
  lwd = 2,
  main = "",
  las = 1,
  cex.axis = 1.2,
  cex.lab = 1.4,
  xaxs = 'i',
  yaxs = 'i'
)
```



Compensated Simple Poisson Process - Martingale

```r
set.seed(123)

# Parameters
lambda <- 0.6
T <- 10

# Initialize variables
Tn <- c()
S <- 0
n <- 0

# Simulate the Poisson process
while (S < T) {
  S <- S + rexp(1, rate = lambda)
  Tn <- c(Tn, S)
  n <- n + 1
}

# Calculate the cumulative sum for the Poisson process
Z <- cumsum(c(0, rep(1, n)))

# Define the N(t) function using stepfun
N <- function(t) {
  return(stepfun(Tn, Z)(t))
}

# Generate a sequence of time points
t <- seq(0, 10, 0.01)

# Plot the Poisson process minus lambda * t
plot(
  t,
  N(t) - lambda * t,
  xlim = c(0, 10),
  ylim = c(-2, 2),
  xlab = "t",
  ylab = expression(N[t]),
  type = "l",
  lwd = 2,
  col = "blue",
  main = "",
  xaxs = "i",
  yaxs = "i",
  las = 1,
  cex.axis = 1.2,
  cex.lab = 1.4
)

# Add a horizontal line at y = 0
abline(h = 0, col = "black", lwd = 2)
```

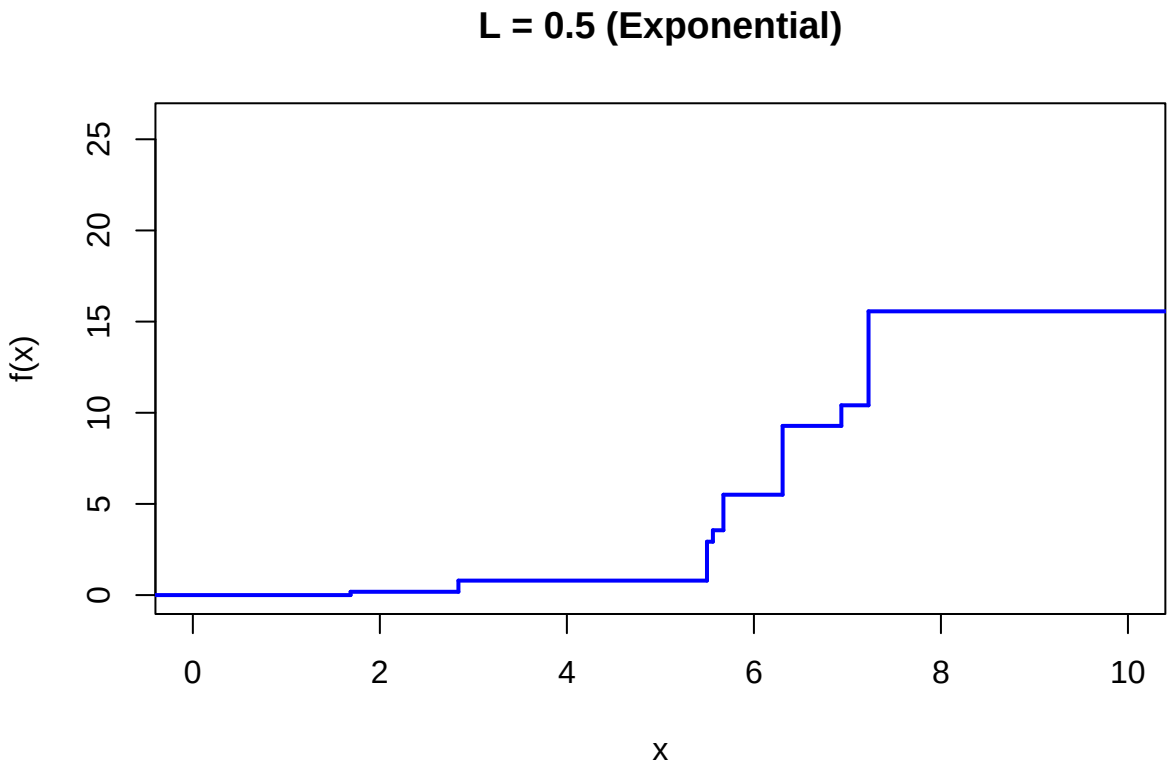
```r
# Add points for the event times
points(Tn, N(Tn) - lambda * Tn, pch = 1, cex = 0.8, col = "blue", lwd = 2)
```



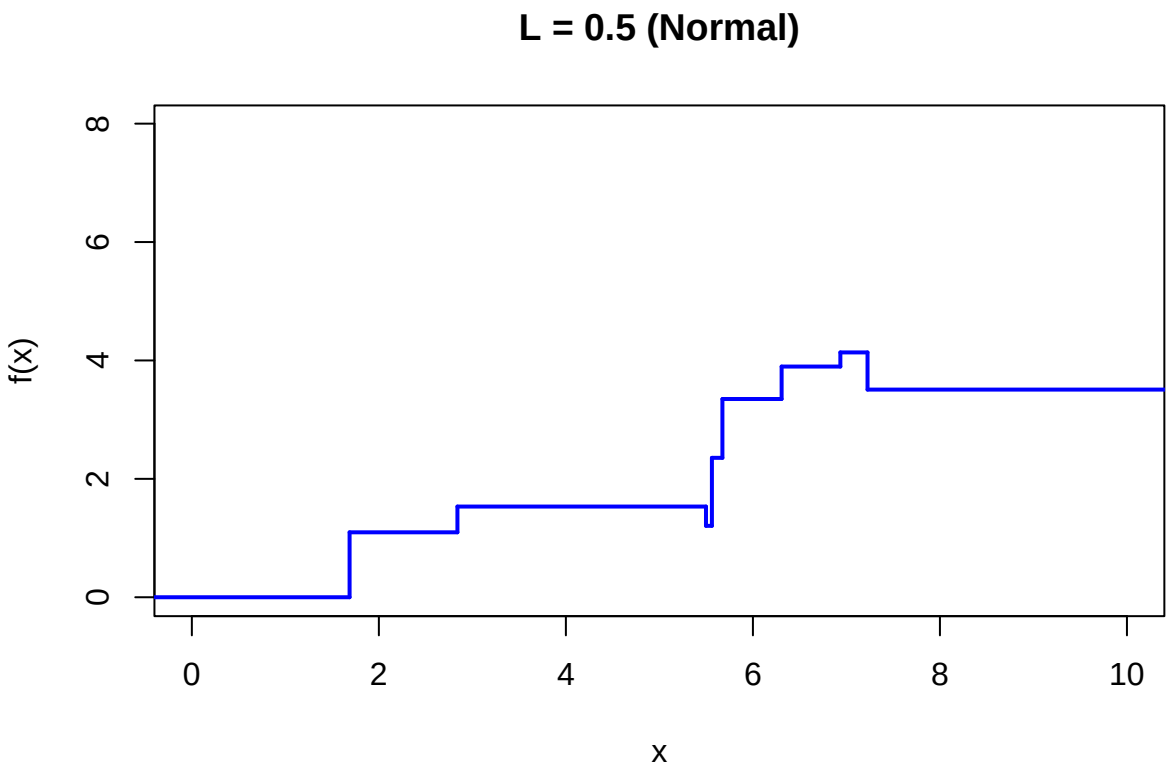
Compound Poisson Process with Gaussian distributed Jump Sizes and Exponentially distributed Jump Sizes

```r
set.seed(123)

# Parameters
N <- 50
rate <- 0.5

# Simulate the process
Tk <- cumsum(rexp(N, rate = rate))
Zk_exp <- rexp(N, rate = rate)
Yk_exp <- cumsum(c(0, Zk_exp))

# Plot the step function for the exponential case
plot(
  stepfun(Tk, Yk_exp),
  xlim = c(0, 10),
  lwd = 2,
  do.points = FALSE,
```

```
  main = "L = 0.5 (Exponential)",
  col = "blue"
)
```



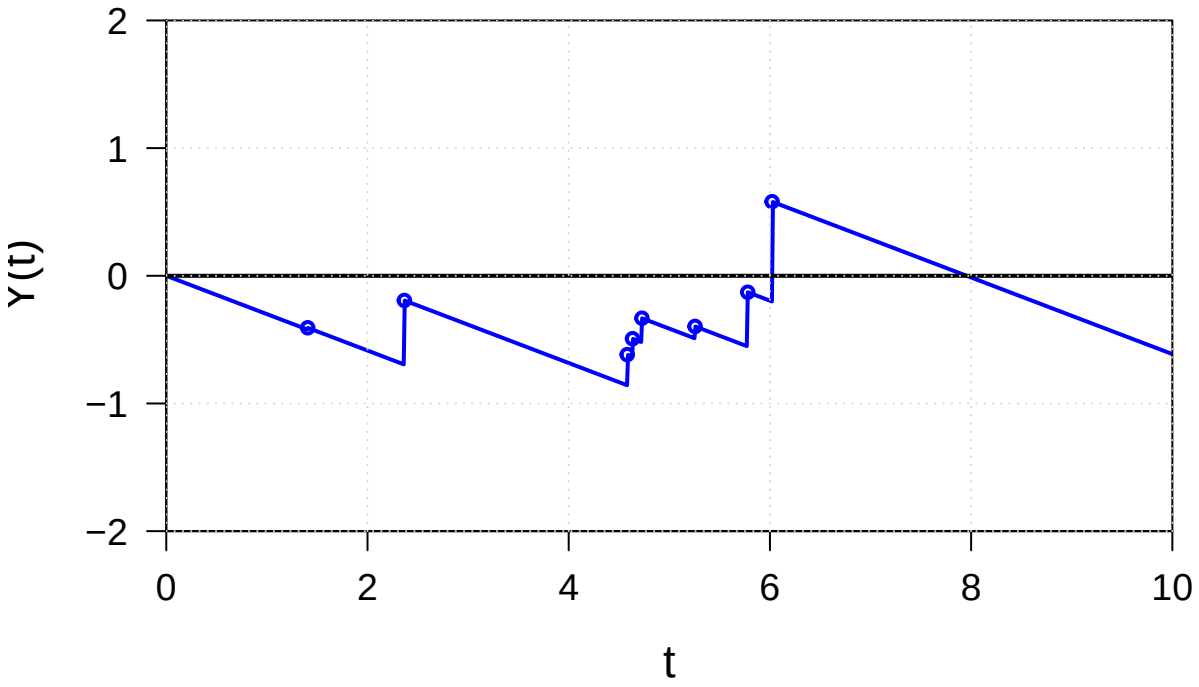

```
# Simulate the process with normal distribution
Zk_norm <- rnorm(N, mean = 0, sd = 1)
Yk_norm <- cumsum(c(0, Zk_norm))

# Plot the step function for the normal case
plot(
  stepfun(Tk, Yk_norm),
  xlim = c(0, 10),
  lwd = 2,
  do.points = FALSE,
  main = "L = 0.5 (Normal)",
  col = "blue"
)
```



Compensated Compound Poisson Process

```r
set.seed(123)

# Parameters
lambda <- 0.6
T <- 10

# Initialize variables
Tn <- c()
S <- 0
n <- 0

# Simulate the Poisson process
while (S < T) {
  S <- S + rexp(1, rate = lambda)
  Tn <- c(Tn, S)
  n <- n + 1
}

# Calculate the cumulative sums for the processes
Z <- cumsum(c(0, rep(1, n)))
Zn <- cumsum(c(0, rexp(n, rate = 2)))

# Define the Y(t) function using stepfun
Y <- function(t) {
  return(stepfun(Tn, Zn)(t))
```

```r
}

# Generate a sequence of time points
t <- seq(0, 10, 0.01)

# Set the outer margins
par(oma = c(0, 0.1, 0, 0))

# Plot Y(t) - 0.5 * lambda * t
plot(
  t,
  Y(t) - 0.5 * lambda * t,
  xlim = c(0, 10),
  ylim = c(-2, 2),
  xlab = "t",
  ylab = expression(paste("Y(t)")),
  type = "l",
  lwd = 2,
  col = "blue",
  main = "",
  xaxs = "i",
  yaxs = "i",
  las = 1,
  cex.axis = 1.2,
  cex.lab = 1.4
)

# Add a horizontal line at y = 0
abline(h = 0, col = "black", lwd = 2)

# Add points for the event times
points(Tn, Y(Tn) - 0.5 * lambda * Tn, pch = 1, cex = 0.8, col = "blue", lwd = 2)

# Add a grid to the plot
grid()
```



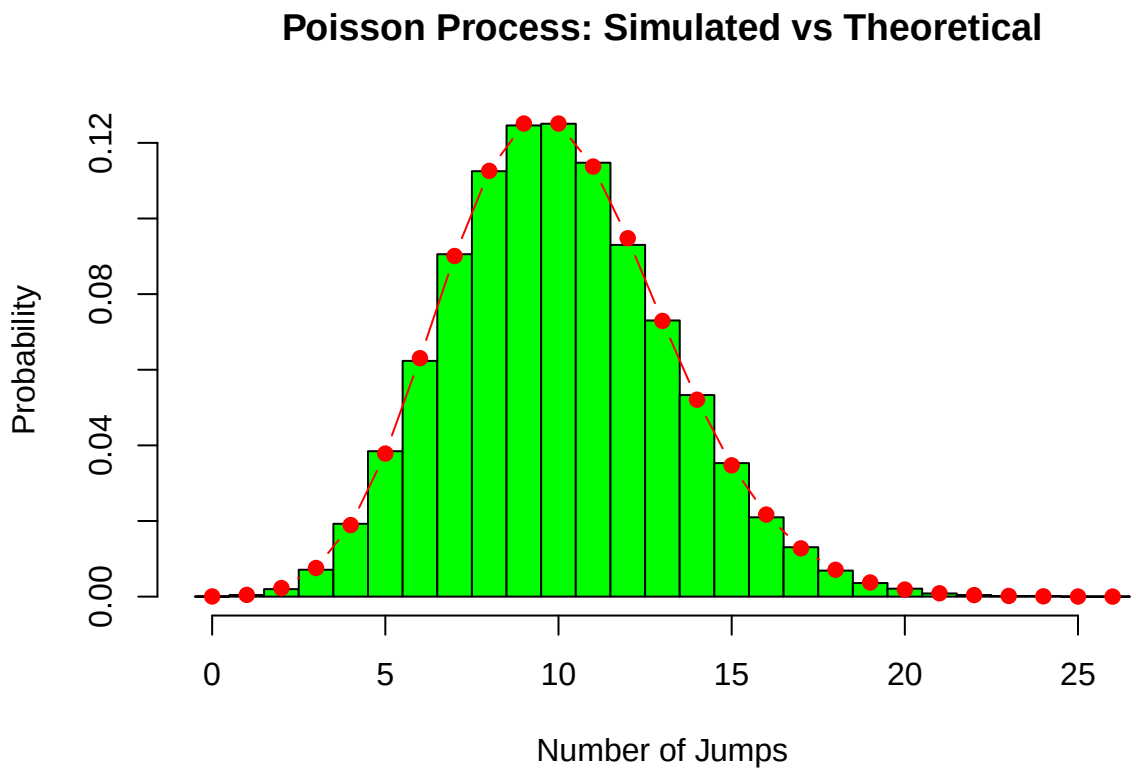
Simple Poisson Process

$$E[N(t) - N(s)] = \lambda(t - s) = 1 \times 10 = 10$$

$$\text{Var}(N(t) - N(s)) = \lambda(t - s) = 1 \times 10 = 10$$

```r
set.seed(123)

# Parameters
lambda_rate <- 1  # Intensity of the Poisson process
T <- 10  # Total time
num_simulations <- 100000  # Number of simulations

# Simulate the Poisson process
simulate_poisson_process <- function(lambda_rate, T) {
  times <- numeric()
  current_time <- 0
  while (current_time < T) {
    interarrival_time <- rexp(1, rate = lambda_rate)
    current_time <- current_time + interarrival_time
    if (current_time < T) {
      times <- c(times, current_time)
    }
  }
```

```r
  return(times)
}

# Run simulations
all_jump_times <- replicate(num_simulations,
                            simulate_poisson_process(lambda_rate, T),
                            simplify = FALSE)
jump_counts <- sapply(all_jump_times, length)

# Theoretical values
k_values <- 0:max(jump_counts)
theoretical_probs <- sapply(k_values, function(k) (lambda_rate * T)^k *
                              exp(-lambda_rate * T) / factorial(k))

# Plot results
hist(jump_counts, breaks = seq(-0.5, max(jump_counts)+0.5, by = 1),
     probability = TRUE, col = "green",
     main = "Poisson Process: Simulated vs Theoretical",
     xlab = "Number of Jumps", ylab = "Probability")

lines(k_values, theoretical_probs, type = "b", col = "red", pch = 19)
```




```r
# Print mean and variance
cat("Theoretical Mean:", lambda_rate * T, "\n")
```

```
## Theoretical Mean: 10
```

```
cat("Simulated Mean:", mean(jump_counts), "\n")
```

```
## Simulated Mean: 10.00572
```

```
cat("Theoretical Variance:", lambda_rate * T, "\n")
```

```
## Theoretical Variance: 10
```

```
cat("Simulated Variance:", var(jump_counts), "\n")
```

```
## Simulated Variance: 9.998067
```

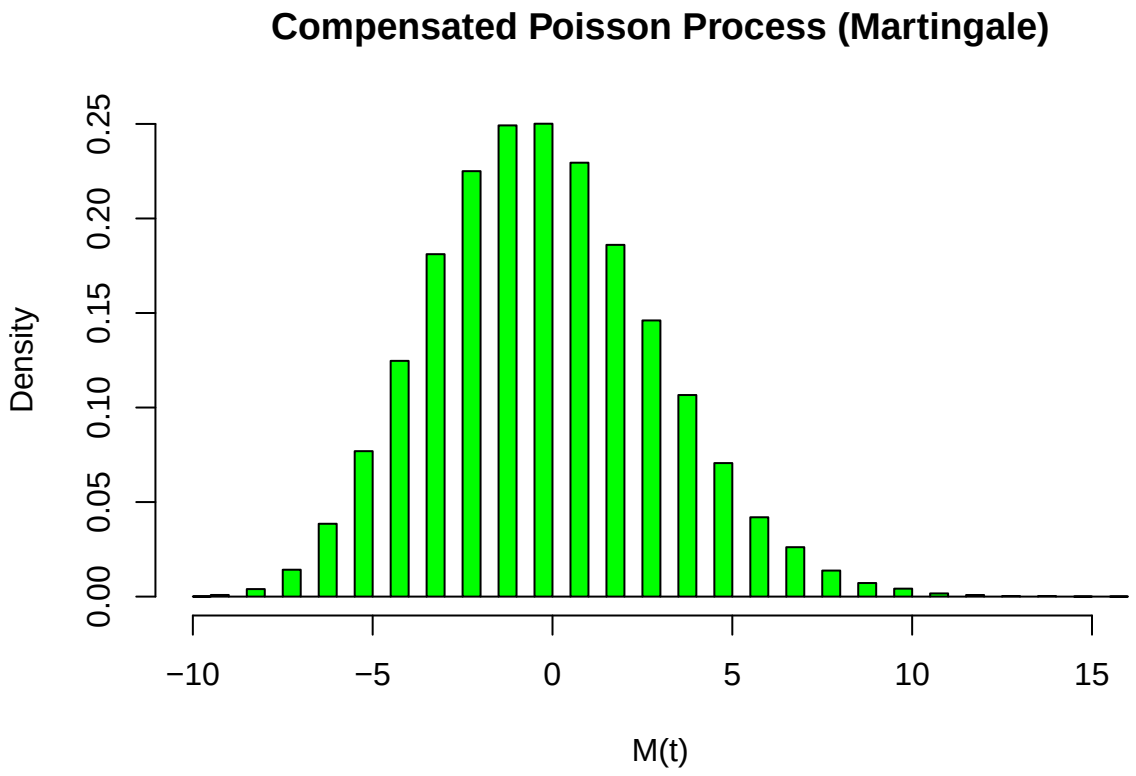
Compensated Simple Poisson Process

$$E[M(t)] = \lambda(t-s) - \lambda(t-s) = 0$$

$$\text{Var}(M(t)) = \lambda(t-s) = 1 \times 10 = 10$$

```
set.seed(123)

# Parameters
lambda_rate <- 1  # Intensity of the Poisson process
T <- 10  # Total time
num_simulations <- 100000  # Number of simulations

# Simulate the Poisson process
simulate_poisson_process <- function(lambda_rate, T) {
  times <- numeric()
  current_time <- 0
  while (current_time < T) {
    interarrival_time <- rexp(1, rate = lambda_rate)
    current_time <- current_time + interarrival_time
    if (current_time < T) {
      times <- c(times, current_time)
    }
  }
  return(times)
}

# Run simulations
all_jump_times <- replicate(num_simulations,
                            simulate_poisson_process(lambda_rate, T),
                            simplify = FALSE)
jump_counts <- sapply(all_jump_times, length)

# Compensated Poisson process
M_t <- jump_counts - lambda_rate * T
```

```r
# Plot results
hist(M_t, breaks = 50, probability = TRUE, col = "green",
     main = "Compensated Poisson Process (Martingale)",
     xlab = "M(t)", ylab = "Density")
```




```r
# Mean and variance of M(t)
mean_M_t <- mean(M_t)
variance_M_t <- var(M_t)

# Print mean and variance
cat("Mean of M(t):", mean_M_t, "\n")
```

```
## Mean of M(t): 0.00572
```

```r
cat("Variance of M(t):", variance_M_t, "\n")
```

```
## Variance of M(t): 9.998067
```

Compound Poisson Process

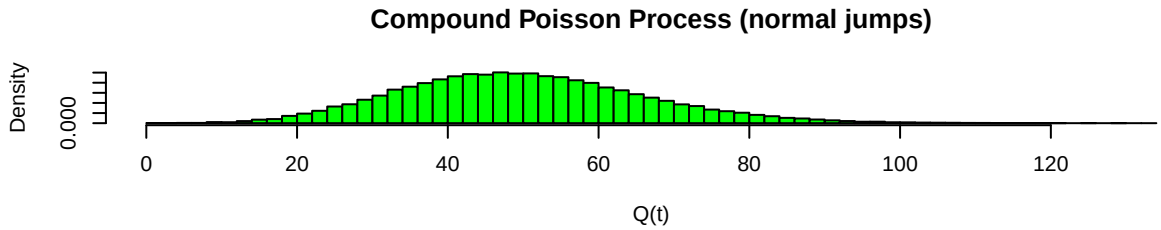
$E[Q(t) - Q(s)|F(s)] = \beta\lambda(t-s) = 5 \times 1 \times 10 = 50$

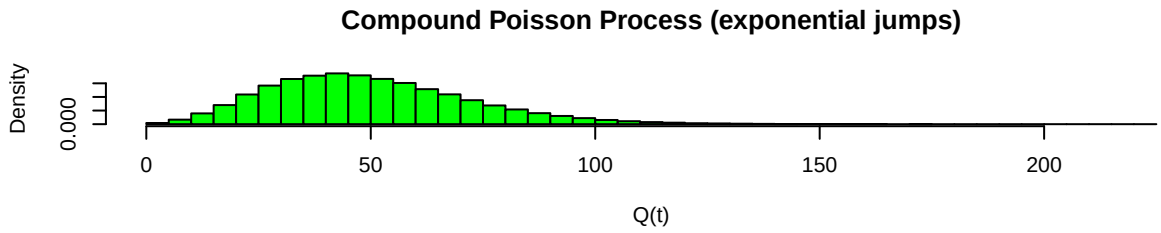
The distribution for the jump sizes is selected, not necessarily defined. The below demonstrates three common distributions for jumps sizes in the normal distirbution, exponential distribution, and gamma distribution.

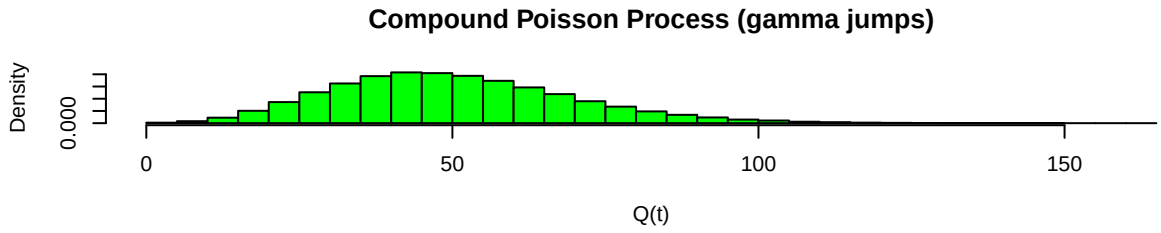
Going forward we will use the noraml distribution as that allows us to add this to the continuous part of the model alongside the Brownian Motion with less complexity.

```
set.seed(123)  # For reproducibility

# Parameters
lambda_rate <- 1  # Intensity of the Poisson process
T <- 10  # Total time
num_simulations <- 100000  # Number of simulations
beta <- 5  # Mean of the jump sizes

# Simulate the Poisson process
simulate_poisson_process <- function(lambda_rate, T) {
  times <- numeric()
  current_time <- 0
  while (current_time < T) {
    interarrival_time <- rexp(1, rate = lambda_rate)
    current_time <- current_time + interarrival_time
    if (current_time < T) {
      times <- c(times, current_time)
    }
  }
  return(times)
}

# Simulate the Compound Poisson process with different jump size distributions
simulate_compound_poisson_process <- function(lambda_rate, T, beta,
                                              distribution = "normal") {
  jump_times <- simulate_poisson_process(lambda_rate, T)
  num_jumps <- length(jump_times)

  # Generate jump sizes based on the specified distribution
  jump_sizes <- switch(distribution,
                       "normal" = rnorm(num_jumps, mean = beta, sd = 1),
                       "exponential" = rexp(num_jumps, rate = 1 / beta),
                       "gamma" = rgamma(num_jumps, shape = 2, scale = beta / 2),
                       stop("Unsupported distribution"))

  Q_t <- sum(jump_sizes)
  return(Q_t)
}

# Run simulations for each distribution
distributions <- c("normal", "exponential", "gamma")
results <- list()

par(mfrow = c(3, 1))  # Set up the plotting area to have 3 rows and 1 column

for (dist in distributions) {
  Q_t_values <- replicate(num_simulations,
                          simulate_compound_poisson_process(lambda_rate,
                                                            T, beta, dist))

  # Store results
```

```
  results[[dist]] <- list(
    Q_t_values = Q_t_values,
    mean = mean(Q_t_values)
  )

  # Plot results
  hist(Q_t_values, breaks = 50, probability = TRUE, col = "green",
       main = paste("Compound Poisson Process (", dist, " jumps)", sep = ""),
       xlab = "Q(t)", ylab = "Density")
}
```








```
# Print results for the normal distribution
dist <- "normal"
cat("Distribution:", dist, "\n")
```

```
## Distribution: normal
```

```
cat("Simulated Mean of Q(t):", results[[dist]]$mean, "\n")
```

```
## Simulated Mean of Q(t): 50.02113
```

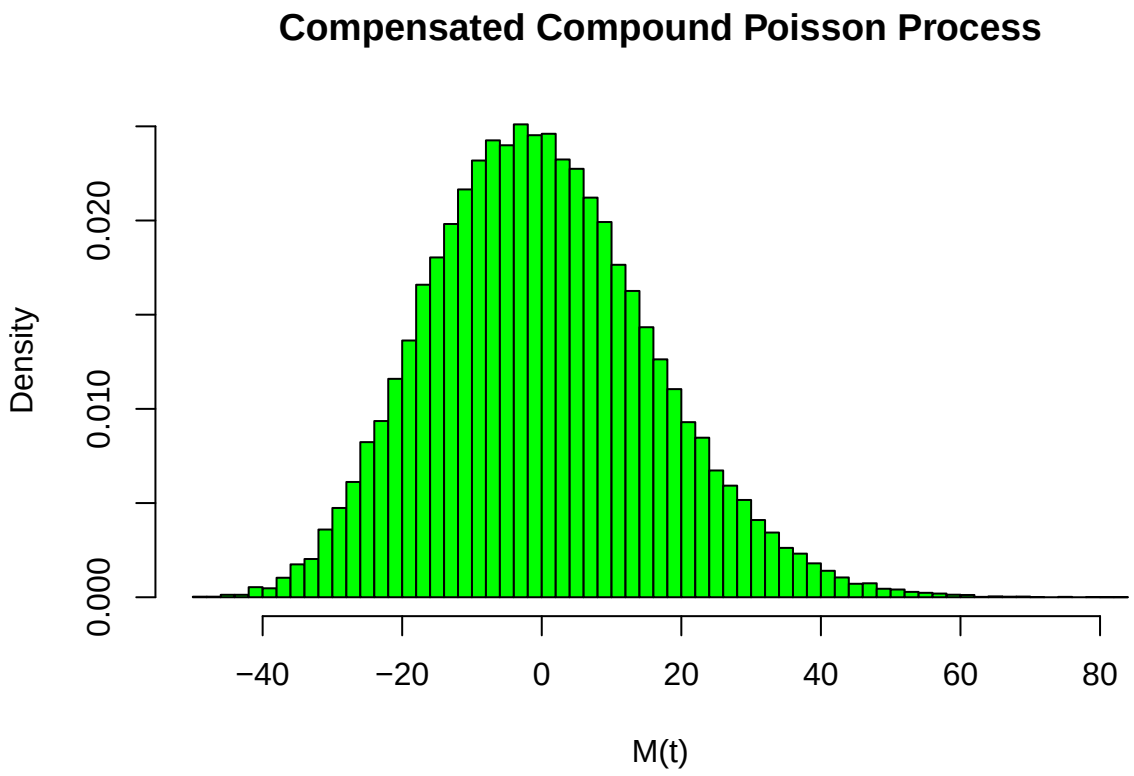
```
theoretical_mean_Q_t <- beta * lambda_rate * T

cat("Theoretical Mean of Q(t):", theoretical_mean_Q_t, "\n")
```

```
## Theoretical Mean of Q(t): 50
```

Compensated Compound Poisson Process

Again we show that the compensated formula provides a zero mean, allowing the process to be a martingale.

```r
set.seed(123)

# Parameters
lambda_rate <- 1  # Intensity of the Poisson process
T <- 10  # Total time
num_simulations <- 100000  # Number of simulations
beta <- 5  # Mean of the jump sizes

# Simulate the Poisson process
simulate_poisson_process <- function(lambda_rate, T) {
  times <- numeric()
  current_time <- 0
  while (current_time < T) {
    interarrival_time <- rexp(1, rate = lambda_rate)
    current_time <- current_time + interarrival_time
    if (current_time < T) {
      times <- c(times, current_time)
    }
  }
  return(times)
}

# Simulate the Compound Poisson process
simulate_compound_poisson_process <- function(lambda_rate, T, beta) {
  jump_times <- simulate_poisson_process(lambda_rate, T)
  num_jumps <- length(jump_times)
  jump_sizes <- rnorm(num_jumps, mean = beta, sd = 1)
  # Normally distributed jump sizes with mean beta
  Q_t <- sum(jump_sizes)
  return(Q_t)
}

# Run simulations for Compound Poisson process
Q_t_values <- replicate(num_simulations,
                        simulate_compound_poisson_process(lambda_rate, T, beta))

# Compensated Compound Poisson process
M_t_values <- Q_t_values - beta * lambda_rate * T

# Plot results
hist(M_t_values, breaks = 50, probability = TRUE, col = "green",
     main = "Compensated Compound Poisson Process",
     xlab = "M(t)", ylab = "Density")
```




```
# Mean
mean_M_t <- mean(M_t_values)

# Theoretical mean and variance
theoretical_mean_M_t <- 0
# Theoretical mean of M(t) should be 0 since it's compensated

# Print mean and variance
cat("Simulated Mean of M(t):", mean_M_t, "\n")
```

```
## Simulated Mean of M(t): 0.02112735
```

```
cat("Theoretical Mean of M(t):", theoretical_mean_M_t, "\n")
```

```
## Theoretical Mean of M(t): 0
```

Simulating Geometric Brownian Motion Asset Model

Geometric Brownian motion (GBM) S is defined by $S_0 > 0$ and the dynamics as defined in the following Stochastic Differential Equation (SDE):

$$dS_t = \mu S_t \, dt + \sigma S_t \, dW_t$$

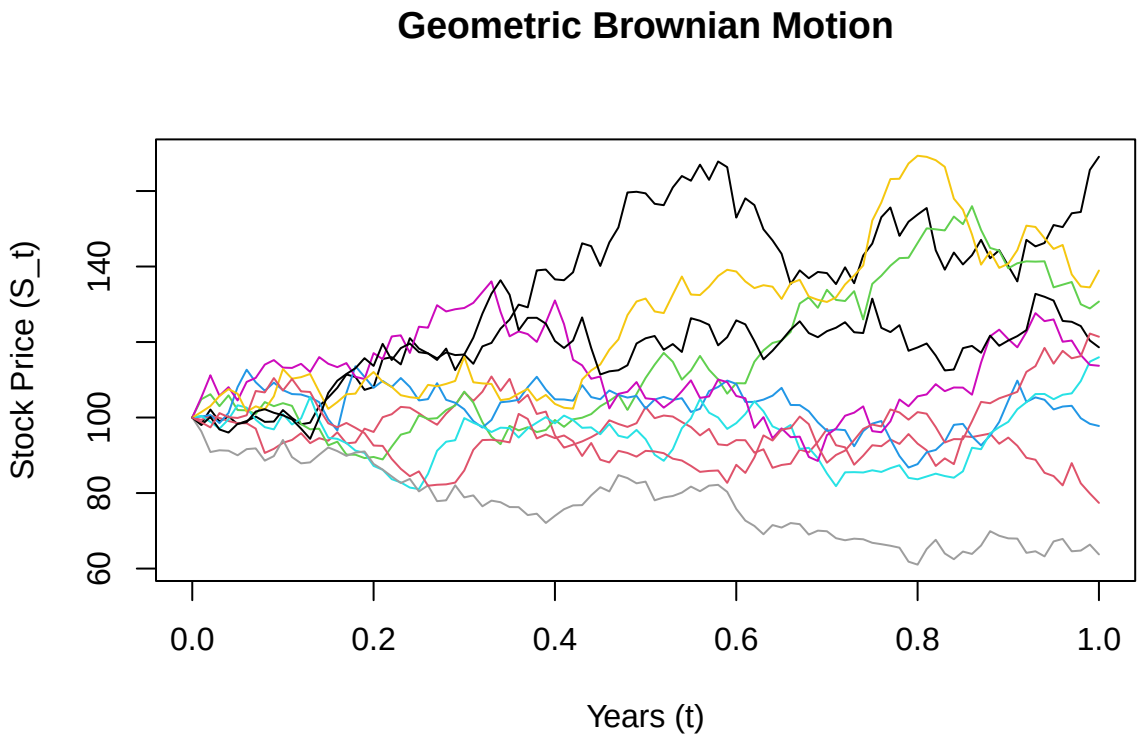
Integrated Form

- $\log S_t = \log S_0 + \int_0^t \left(\mu - \frac{\sigma^2}{2}\right) ds + \int_0^t \sigma \, dW_s$
- $\log S_t = \log S_0 + \left(\mu - \frac{\sigma^2}{2}\right) t + \sigma W_t$
- $\log S_t \sim \mathcal{N}\left(\log S_0 + \left(\mu - \frac{\sigma^2}{2}\right) t, \sigma^2 t\right)$

Explicit Expression

$$S_t = S_0 \, e^{\left(\mu - \frac{\sigma^2}{2}\right) t + \sigma W_t}$$

```r
set.seed(123)

# Parameters
mu <- 0.1  # Drift coefficient
n <- 100  # Number of steps
T <- 1  # Time in years
M <- 10  # Number of simulations
S0 <- 100  # Initial stock price
sigma <- 0.3  # Volatility

# Calculate each time step
dt <- T / n

# Simulation using matrices
St <- matrix(0, nrow = n + 1, ncol = M)
St[1, ] <- S0  # Initial stock price

# Generate random paths
for (i in 2:(n + 1)) {
  Z <- rnorm(M, mean = 0, sd = sqrt(dt))
  St[i, ] <- St[i - 1, ] * exp((mu - sigma^2 / 2) * dt + sigma * Z)
}

# Define time interval
time <- seq(0, T, length.out = n + 1)

par(mar = c(5, 5, 6, 2))

# Plot
matplot(time, St, type = "l", lty = 1, col = 1:M, lwd = 1,
        xlab = "Years (t)", ylab = "Stock Price (S_t)",
        main = "Geometric Brownian Motion")
```



Simulating Stock Model with Brownian Motion and Jump Processes

$$S(t) = S(0)e^{\sigma W(t)+(\alpha-\beta\lambda-\frac{1}{2}\sigma^2)t}\prod_{i=1}^{N(t)}(Y_i+1)$$

The continuous stochastic process portion is:

$$X(t) = S(0)e^{\sigma W(t)+(\alpha-\beta\lambda-\frac{1}{2}\sigma^2)t}$$

The jump process portion is:

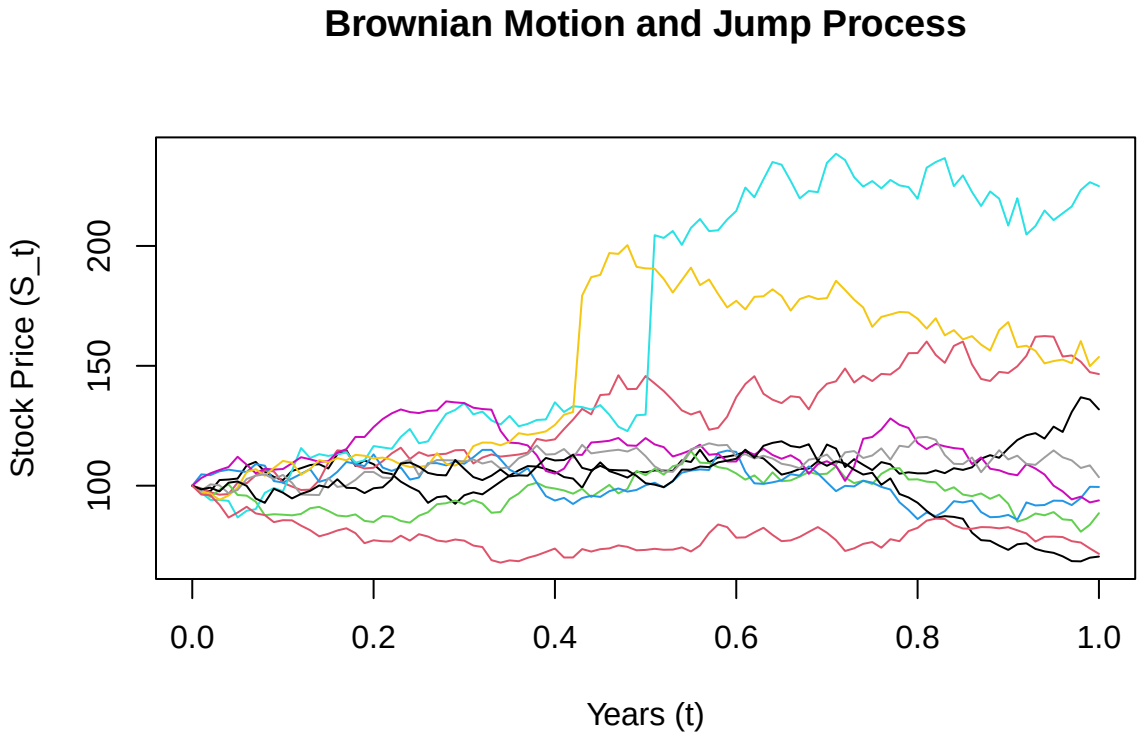
$$J(t) = \prod_{i=1}^{N(t)}(Y_i+1)$$

Thus to simplify:

$$S(t) = X(t)J(t)$$

```r
set.seed(123)

# Parameters
lambda_jump <- 0.1  # Average number of jumps per unit time
```

```r
Beta <- 0.5  # Jump size
mu <- 0.1  # Drift coefficient
n <- 100  # Number of steps
T <- 1  # Time in years
M <- 10  # Number of simulations
S0 <- 100  # Initial stock price
sigma <- 0.3  # Volatility

# Calculate each time step
dt <- T / n

# Simulation using matrices
St <- matrix(0, nrow = n + 1, ncol = M)
St[1, ] <- S0  # Initial stock price

# Generate random paths
for (j in 1:M) {
  Wt <- cumsum(rnorm(n, mean = 0, sd = sqrt(dt)))  # Brownian motion
  Nt <- rpois(n, lambda = lambda_jump * dt)  # Poisson process for jumps
  Yt <- rnorm(n, mean = Beta, sd = 0.1)  # Jump sizes

  # Continuous process X(t)
  Xt <- S0 * exp(sigma * Wt + (mu - Beta * lambda_jump -
                                 0.5 * sigma^2) * (1:n) * dt)

  # Jump process J(t)
  Jt <- cumprod(1 + Yt * Nt)

  # Full process S(t)
  St[2:(n + 1), j] <- Xt * Jt
}

# Define time interval
time <- seq(0, T, length.out = n + 1)

par(mar = c(5, 5, 6, 2))

# Plot
matplot(time, St, type = "l", lty = 1, col = 1:M, lwd = 1,
        xlab = "Years (t)", ylab = "Stock Price (S_t)",
        main = "Brownian Motion and Jump Process")
```



Stock Model - Geometric Poisson Process

$$S(t) = S(0)e^{\alpha t + N(t)\log(\sigma+1) - \lambda\sigma t} = S(0)e^{(\alpha-\lambda\sigma)t}(\sigma+1)^{N(t)}$$

- $\sigma > -1$
- $\sigma \neq 0$
- $N(t)$ is a Poisson process with intensity λ under measure $\mathbb{P}$
- $S(t)$ has a mean rate of return of α

Differences in Sigma for Jump Processes

We show how differences in sigma will affect the jumps by nullifying the affects of the mean rate of return:

If $\sigma > 0$ then the jump process moves up in value during jumps and down between jumps. If $-1 < \sigma < 0$ the value moves down during jumps and up between jumps.

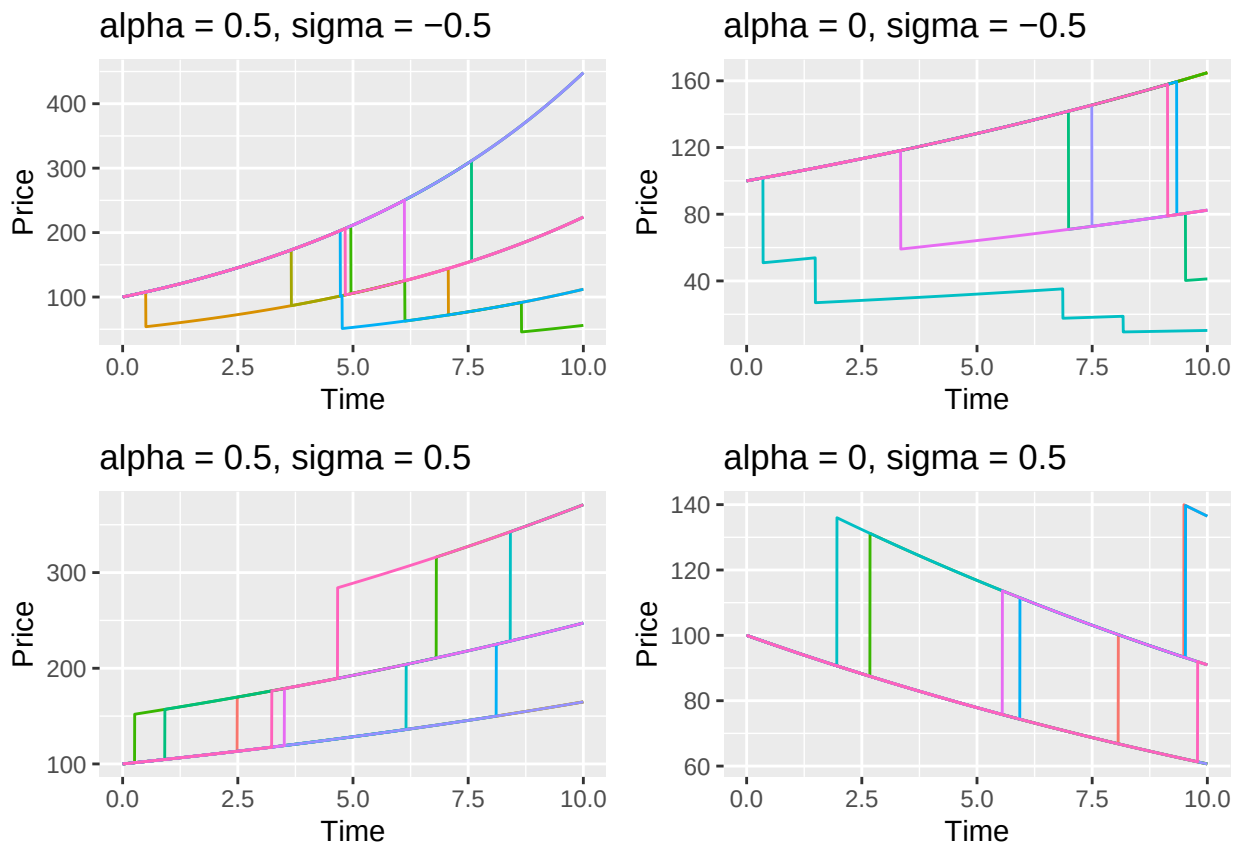
```
set.seed(123)

library(ggplot2)
```

```
## Warning: package 'ggplot2' was built under R version 4.3.3
```

```r
library(gridExtra)
```

```
## Warning: package 'gridExtra' was built under R version 4.3.3
```

```r
# Parameters
lambda <- 0.1  # Intensity of the Poisson process
T <- 10  # Total time in years
dt <- 1/252  # Time step, assuming daily steps in a trading year
n <- as.integer(T / dt)  # Number of steps
M <- 10  # Number of simulations
S0 <- 100  # Initial stock price

# Function to simulate paths for given alpha and sigma
simulate_paths <- function(alpha, sigma) {
  t <- seq(0, T, length.out = n + 1)
  St <- matrix(0, nrow = n + 1, ncol = M)
  St[1, ] <- S0  # Initial stock price

  for (j in 1:M) {
    N <- cumsum(c(0, rpois(n, lambda = lambda * dt)))
    # Cumulative Poisson process
    J <- (sigma+1) ^ N  # Jump process
    St[, j] <- S0 * exp((alpha - lambda * sigma) * t) * J
  }

  df <- data.frame(Time = rep(t, M), Price = as.vector(St),
                   Path = rep(1:M, each = n + 1))
  return(df)
}

# Define scenarios
scenarios <- list(
  list(alpha = 0.1, sigma = -0.5, title = "alpha = 0.5, sigma = -0.5"),
  list(alpha = 0, sigma = -0.5, title = "alpha = 0, sigma = -0.5"),
  list(alpha = 0.1, sigma = 0.5, title = "alpha = 0.5, sigma = 0.5"),
  list(alpha = 0, sigma = 0.5, title = "alpha = 0, sigma = 0.5")
)

# Plot each scenario
plots <- lapply(scenarios, function(scenario) {
  df <- simulate_paths(scenario$alpha, scenario$sigma)
  ggplot(df, aes(x = Time, y = Price, color = factor(Path), group = Path)) +
    geom_line() +
    ggtitle(scenario$title) +
    xlab("Time") +
    ylab("Price") +
    theme(legend.position = "none")
})

# Arrange plots side by side
grid.arrange(grobs = plots, ncol = 2)
```



High-Frequency Trading and Lévy Processes

High-frequency trading (HFT) involves the rapid buying and selling of securities using automated trading systems to capitalize on small price discrepancies. In such an environment, traditional models like the Brownian motion often fall short because they cannot adequately capture the abrupt price changes and market microstructure noise seen at very high frequencies. Lévy processes, with their ability to incorporate jumps, provide a more realistic framework for modeling asset prices in HFT settings.

Market Microstructure Noise Market microstructure noise refers to the deviations of the observed prices from the efficient price due to market mechanisms like bid-ask spreads, order flow, and discrete trading. This noise is especially pronounced at higher frequencies, where data sampling intervals are smaller and the effects of market mechanisms are more observable.

```
set.seed(123)

library(ggplot2)

# Parameters
lambda_jump <- 0.1  # Average number of jumps per unit time
mu <- 0.05  # Expected return
sigma <- 0.2  # Volatility
Beta <- 0.5  # Jump size
```

```r
T <- 10  # Total time
dt <- 1/252  # Time step, assuming daily steps in a trading year
n <- as.integer(T / dt)  # Number of steps

# Time vector
t <- seq(0, T, length.out = n)

# Simulate Brownian motion
W <- cumsum(rnorm(n, mean = 0, sd = sqrt(dt)))

# Simulate Poisson process for jump counts
N <- rpois(n, lambda = lambda_jump * dt)

# Calculate jumps
J <- cumsum(N * Beta)

# Merton jump-diffusion model
X <- mu * t + sigma * W + J

# Plotting the results
df <- data.frame(Time = t, Price = X)
p <- ggplot(df, aes(x = Time, y = Price)) +
  geom_line() +
  ggtitle("Simulated Merton Jump-Diffusion Model") +
  xlab("Time") +
  ylab("Price")

# Save the plot with specified dimensions
ggsave("merton_jump_diffusion.png", plot = p, width = 8, height = 4)
```

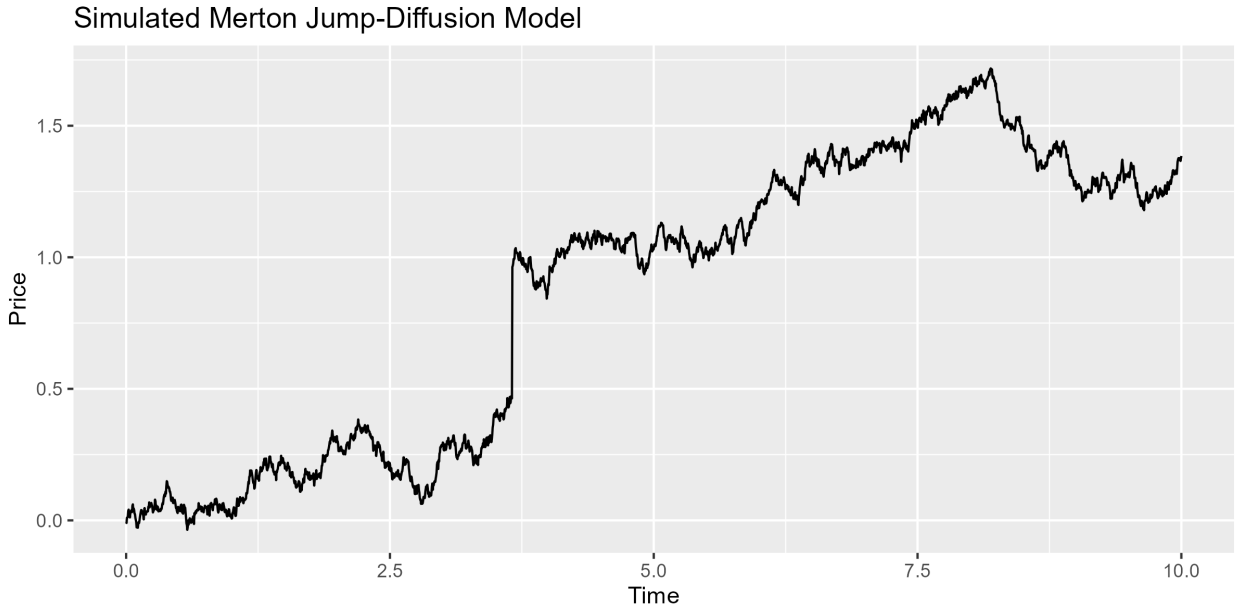


Figure 1: Simulated Merton Jump-Diffusion Model

```r
set.seed(123)

library(ggplot2)

# Parameters
lambda_jump <- 0.1  # Average number of jumps per unit time
mu <- 0.05  # Expected return
sigma <- 0.2  # Volatility
Beta <- 0.5  # Jump size
T <- 10  # Total time in years
dt <- 1/252  # Time step, assuming daily steps in a trading year
n <- as.integer(T / dt)  # Number of steps
M <- 10  # Number of simulations

# Time vector
t <- seq(0, T, length.out = n + 1)

# Initialize matrix to store paths
St <- matrix(0, nrow = n + 1, ncol = M)
St[1, ] <- 100  # Initial stock price

# Simulate multiple paths
for (j in 1:M) {
  W <- cumsum(c(0, rnorm(n, mean = 0, sd = sqrt(dt))))
  # Brownian motion
  N <- cumsum(c(0, rpois(n, lambda = lambda_jump * dt)))
  # Cumulative Poisson process
  J <- Beta * N  # Jump process
  St[, j] <- St[1, j] * exp((mu - 0.5 * sigma^2) * t + sigma * W + J)
}

# Convert matrix to long format for ggplot2
df <- data.frame(Time = rep(t, M), Price = as.vector(St),
                 Path = rep(1:M, each = n + 1))

# Plotting the results
p <- ggplot(df, aes(x = Time, y = Price, color = factor(Path), group = Path)) +
  geom_line() +
  ggtitle("Simulated Merton Jump-Diffusion Model") +
  xlab("Time") +
  ylab("Price") +
  theme(legend.position = "none")

ggsave("merton_jump_diffusion_multiple_paths.png", plot = p, width = 8, height = 4)
```

Example: Implementing a Simple Lévy Process in Python In this code: - We simulate the continuous part as a Brownian motion (`W`). - Jumps are modeled by a Poisson process (`N`) that decides the number of jumps at each interval, multiplied by the jump size (`Beta`). - The total price path (`X`) is the sum of the drift component, the continuous component, and the jump component.

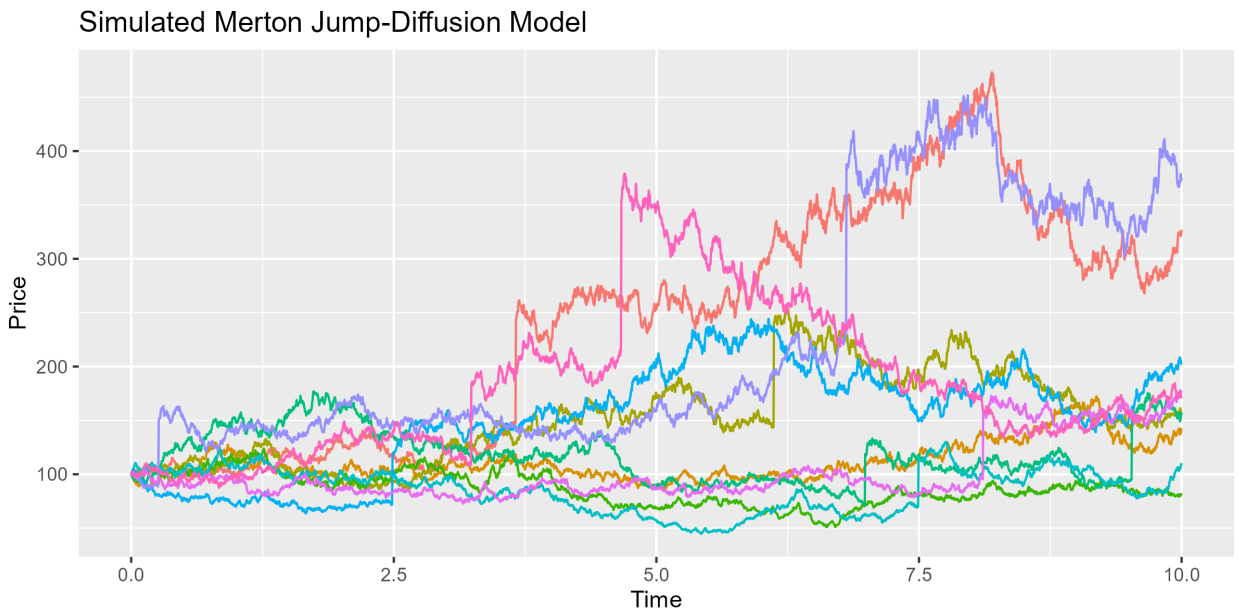


Figure 2: Simulated Merton Jump-Diffusion Model with Multiple Paths

Application in HFT

In high-frequency trading, models like this help in designing algorithms that can quickly respond to price jumps and drops, optimizing the trading strategy in microseconds to capitalize on rapid market movements. They are used in algorithmic trading to simulate possible price paths, evaluate the impact of jumps on trading strategies, and manage the risk associated with such abrupt price changes.