# Evil Random Numbers

[Pseudorandom number generation (PRNG)](https://en.wikipedia.org/wiki/Pseudorandom_number_generator) is a crucial aspect of many computer applications, particularly those involving simulation, cryptography, and statistical sampling. PRNGs are algorithms that aim to produce sequences of numbers that appear random, even though they are generated by deterministic processes. The numbers are called "pseudorandom" because they only exhibit properties of randomness but are not genuinely random in the strictest sense. A good PRNG is characterized by a combination of factors, including high-quality randomness, a long period, and efficient implementation.

A bad pseudorandom number generator can have various shortcomings, which include:

Poor randomness: A PRNG should generate numbers that appear random, meaning they are [uniformly distributed](https://en.wikipedia.org/wiki/Continuous_uniform_distribution) and [independent](https://en.wikipedia.org/wiki/Independence_(probability_theory)) of one another. A bad PRNG might produce numbers with discernible patterns or [correlations](https://en.wikipedia.org/wiki/Correlation), which can lead to [biased outcomes](https://en.wikipedia.org/wiki/Bias_(statistics)) in simulations or [statistical sampling](https://en.wikipedia.org/wiki/Sampling_(statistics)), and even compromise security in cryptographic applications.

Short period: The period of a PRNG refers to the length of the sequence it generates before it starts repeating. A bad PRNG will have a short period, which can be problematic in applications that require a long sequence of random numbers. Repeated sequences can introduce patterns and correlations that affect the results of the application.

Predictability: A secure PRNG should be difficult to predict even if an attacker has knowledge of previous outputs. If a PRNG is easy to predict, it can be exploited in cryptographic applications, leading to potential security breaches. A bad PRNG might have weak algorithms or insufficient entropy sources, making its output predictable to some extent.

Poor seeding: The initial value used to start the generation of pseudorandom numbers is called the seed. A bad PRNG might have a weak seeding mechanism, resulting in the same sequence of numbers being generated across different instances or applications. This can have serious implications for security and randomness.

Inefficient implementation: A PRNG should be computationally efficient to generate random numbers quickly. A bad PRNG might be slow or resource-intensive, which can hinder performance in applications where large quantities of random numbers are needed rapidly.

Lack of cryptographic strength: Some PRNGs are designed specifically for cryptographic applications, where the quality of randomness and unpredictability are paramount. A bad PRNG might not meet the stringent requirements of these applications, potentially compromising the security of sensitive data and communications.

## Evil Random Numbers Example R Code

The random number generation techniques employed in the code are:

[Box-Muller method](https://en.wikipedia.org/wiki/Box%E2%80%93Muller_transform): This method generates normally distributed random numbers from uniformly distributed random numbers by performing coordinate transformations. It requires two independent uniform random numbers (u1 and u2) to generate two normally distributed random numbers (z0 and z1). All of the following generate the uniform random variates that are then transformed using the Box-Muller to create the plots.

Built-in RNG (runif\_wrapper): This function is a wrapper for R's built-in runif() function, which generates uniformly distributed random numbers. R's runif() function uses the [Mersenne-Twister algorithm](https://en.wikipedia.org/wiki/Mersenne_Twister), a widely used and well-tested pseudorandom number generator with good statistical properties. The Mersenne-Twister algorithm provides a long period and high-quality random numbers, making it suitable for a wide range of applications..

[Linear Congruential Generator (LCG)](https://en.wikipedia.org/wiki/Linear_congruential_generator): The LCG is a simple method to generate random numbers using a linear equation with specified parameters (lcg\_a, lcg\_c, and lcg\_m). The generated numbers have a uniform distribution. The code includes various LCG configurations: one good "desert island" LCG and four bad LCGs.

[Xorshift RNG](https://en.wikipedia.org/wiki/Xorshift): The Xorshift algorithm is a fast and efficient method for generating random numbers using [bitwise operations](https://en.wikipedia.org/wiki/Bitwise_operation) (bitwise XOR and bitwise shift). The algorithm has better statistical properties than LCGs and is known for its simplicity and speed.

[Tausworthe RNG](https://homes.luddy.indiana.edu/kapadia/project2/node9.html): The Tausworthe method is a [linear feedback shift register-based](https://en.wikipedia.org/wiki/Linear-feedback_shift_register) random number generator. It generates a binary sequence by XORing the bits at specific positions in the register and uses these bits to form the random number.

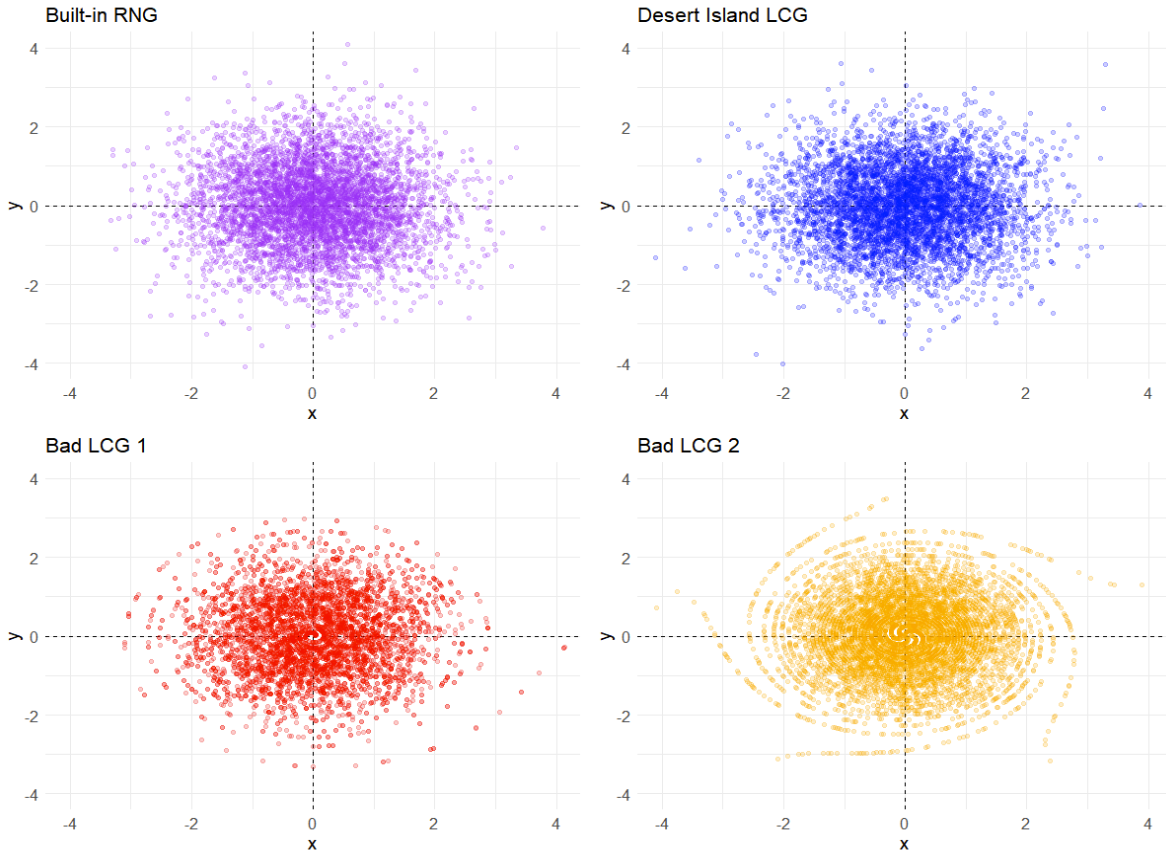
The code generates 5,000 random numbers using each of these techniques and visualizes the distributions using ggplot2. Each generated dataset is plotted on a 2D plane to visually compare the distributions, revealing the quality of the random number generation. The random numbers are transformed using Box-Muller.

When examining the plots, notice the built in R functionality and the “desert island” LCG look like scattered clouds of points. The bad LCGs all have some sort of pattern to them, especially LCG 3 and LCG 4. A zoomed in plot is provided at the end to see even more of the patterns present in Bad LCG 1 and Bad LCG 2.

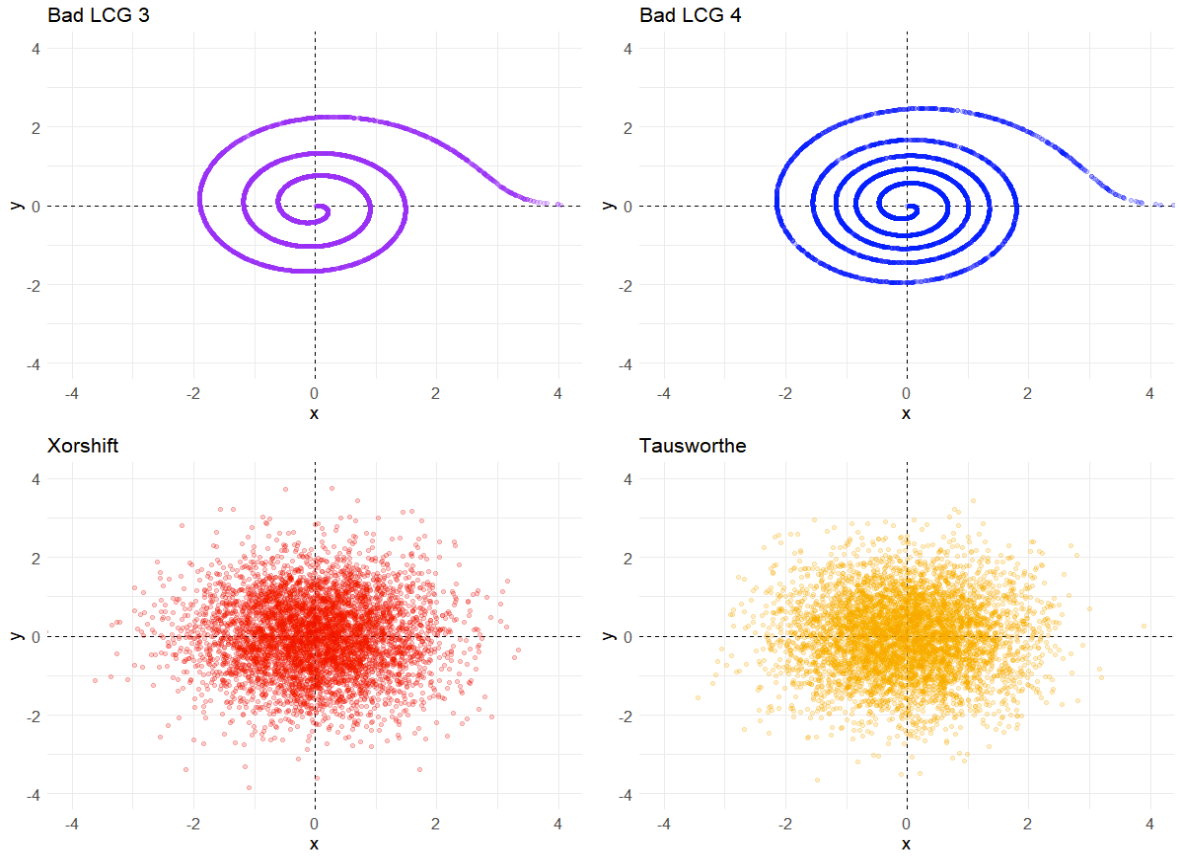
library(ggplot2)  
library(gridExtra)

*# Function to create a 4 quadrant coordinate axis*  
create\_4\_quadrant\_axis <- **function**() {  
 list(  
 geom\_vline(aes(xintercept = 0), linetype = "dashed", color = "black"),  
 geom\_hline(aes(yintercept = 0), linetype = "dashed", color = "black"),  
 coord\_cartesian(xlim = c(-4, 4), ylim = c(-4, 4))  
 )  
}  
  
*# Box-Muller method*  
box\_muller <- **function**(n, rng, seed) {  
 random\_numbers <- rng(n \* 2, seed)  
 uniform\_random\_numbers <- random\_numbers / max(random\_numbers)  
   
 u1 <- uniform\_random\_numbers[seq(1, n \* 2, 2)]  
 u2 <- uniform\_random\_numbers[seq(2, n \* 2, 2)]  
   
 z0 <- sqrt(-2 \* log(u1)) \* cos(2 \* pi \* u2)  
 z1 <- sqrt(-2 \* log(u1)) \* sin(2 \* pi \* u2)  
   
 return(list(x = z0, y = z1))  
}  
  
runif\_wrapper <- **function**(n, seed) {  
 set.seed(seed)  
 return(runif(n))  
}  
  
*# LCG function*  
lcg <- **function**(n, seed, lcg\_a, lcg\_c, lcg\_m) {  
 random\_numbers <- numeric(n)  
 x <- seed  
 **for** (i **in** 1:n) {  
 x <- (lcg\_a \* x + lcg\_c) %% lcg\_m  
 random\_numbers[i] <- x / lcg\_m  
 }  
 return(random\_numbers)  
}  
  
*# Xorshift function*  
xorshift <- **function**(n, seed) {  
 a <- 13  
 b <- 7  
 c <- 17  
   
 xorshift32 <- **function**(state) {  
 state <- bitwXor(state, bitwShiftL(state, a))  
 state <- bitwXor(state, bitwShiftR(state, b))  
 state <- bitwXor(state, bitwShiftL(state, c))  
 return(state)  
 }  
   
 random\_integers <- integer(n)  
 state <- seed  
 **for** (i **in** 1:n) {  
 state <- xorshift32(state)  
 random\_integers[i] <- state  
 }  
   
 random\_numbers <- as.double(random\_integers %% (2^31 - 1)) / (2^31 - 1)  
 return(random\_numbers)  
}  
  
tausworthe <- **function**(n, seed, r, q, l) {  
   
*# [Tausworthe Code Redacted]*

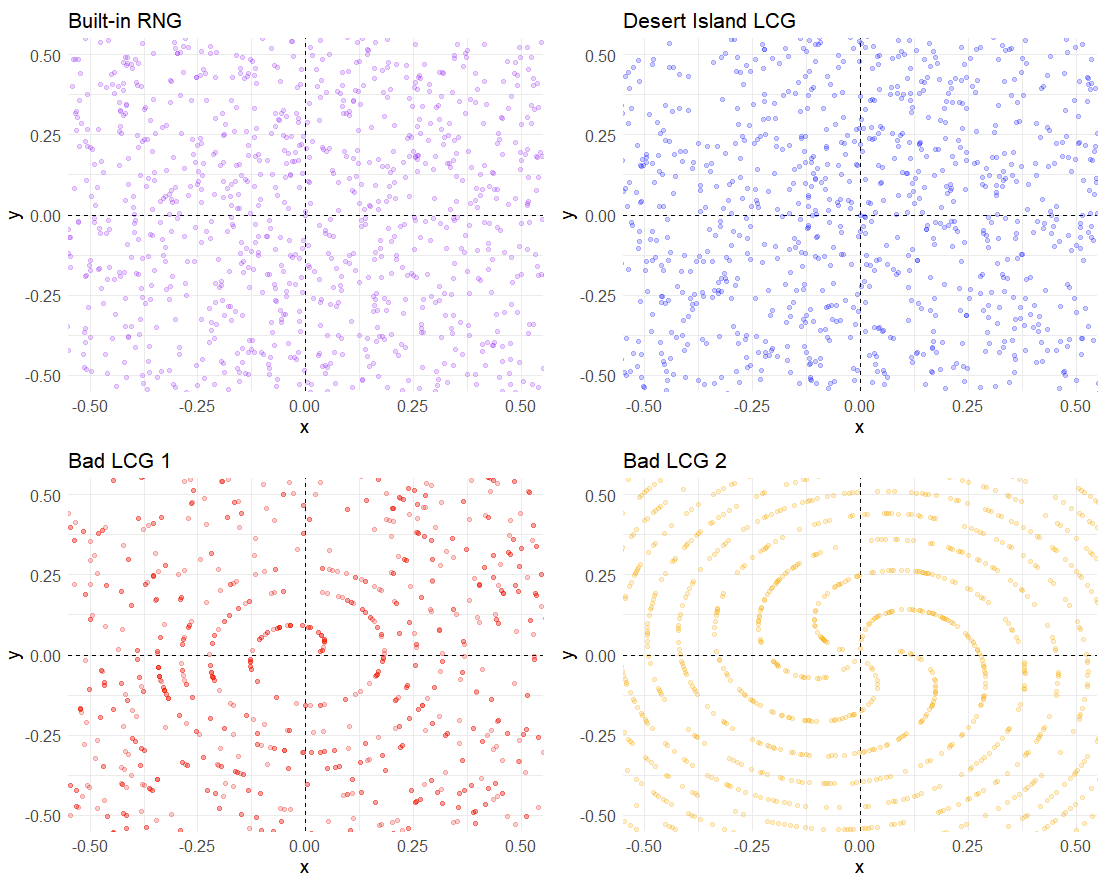
}  
  
n <- 5000  
  
*# Good random number generator (R's built-in)*  
builtin <- box\_muller(n, runif\_wrapper, seed = 1)  
  
*# Good "desert island" LCG*  
desert <-   
 box\_muller(n, **function**(n, seed) lcg(n, seed = 1, lcg\_a = 16807, lcg\_c = 0,   
 lcg\_m = 2^31 - 1 ))  
  
*# Bad LCG 1*  
bad\_lcg1 <-   
 box\_muller(n, **function**(n, seed) lcg(n, seed = 1, lcg\_a = 2147483587,   
 lcg\_c = 1664525, lcg\_m = 2^32))  
  
*# Bad LCG 2*  
bad\_lcg2 <-   
 box\_muller(n, **function**(n, seed) lcg(n, seed = 2, lcg\_a = 25214903917,   
 lcg\_c = 115, lcg\_m = 2^48))  
  
*# Bad LCG 3*  
bad\_lcg3 <-   
 box\_muller(n, **function**(n, seed) lcg(n, seed = 3, lcg\_a = 3, lcg\_c = 1,   
 lcg\_m = 2^31))  
  
*# Bad LCG 4*  
bad\_lcg4 <-   
 box\_muller(n, **function**(n, seed) lcg(n, seed = 4, lcg\_a = 5, lcg\_c = 3,   
 lcg\_m = 2^31))  
  
*# Xorshift RNG*  
xor <- box\_muller(n, xorshift, seed = 1)  
  
*# Tausworthe RNG*  
taus <- *# [Tausworthe Code Redacted]*  
  
plot\_list <- list(builtin = builtin, desert = desert, bad\_lcg1 = bad\_lcg1,   
 bad\_lcg2 = bad\_lcg2)  
colors <- c("purple", "blue", "red", "orange")  
titles <- c("Built-in RNG", "Desert Island LCG", "Bad LCG 1", "Bad LCG 2")  
  
plots <- lapply(1:length(plot\_list), **function**(i) {  
 p <- ggplot() +  
 create\_4\_quadrant\_axis() +  
 geom\_point(data = data.frame(x = plot\_list[[i]]$x, y = plot\_list[[i]]$y),  
 aes(x = x, y = y), col = colors[i], alpha = 0.2) +  
 labs(title = titles[i], x = "x", y = "y") +  
 theme\_minimal() +  
 theme(plot.title = element\_text(size = 16),  
 axis.title = element\_text(size = 14),  
 axis.text = element\_text(size = 12))  
 return(p)  
})  
  
grid.arrange(grobs = plots, ncol = 2)



plot\_list <- list(bad\_lcg3 = bad\_lcg3, bad\_lcg4 = bad\_lcg4, xor = xor,   
 taus = taus)  
colors <- c("purple", "blue", "red", "orange")  
titles <- c("Bad LCG 3", "Bad LCG 4", "Xorshift", "Tausworthe")  
  
plots <- lapply(1:length(plot\_list), **function**(i) {  
 p <- ggplot() +  
 create\_4\_quadrant\_axis() +  
 geom\_point(data = data.frame(x = plot\_list[[i]]$x, y = plot\_list[[i]]$y),  
 aes(x = x, y = y), col = colors[i], alpha = 0.2) +  
 labs(title = titles[i], x = "x", y = "y") +  
 theme\_minimal() +  
 theme(plot.title = element\_text(size = 16),  
 axis.title = element\_text(size = 14),  
 axis.text = element\_text(size = 12))  
 return(p)  
})  
  
grid.arrange(grobs = plots, ncol = 2)



Here is a zoomed in version of the first graph to better see the two bad LCGs.



## Evil Random Numbers Example Python Code

[Python Notebook](https://colab.research.google.com/drive/1L5pHD4MARFccOBN14kKnVyrA9qSdFM2j#scrollTo=yJeXYGZ4Q4EA)

Note that the “Bad LCG2” looks better in Python due to it using 64-bit data types.

# Queues ‘R’ Us

[Kendall's notation](https://en.wikipedia.org/wiki/Kendall%27s_notation) is a widely used system for describing [queueing systems](https://en.wikipedia.org/wiki/Queueing_theory). It consists of three parts: A/B/C, where A refers to the distribution of interarrival times, B refers to the distribution of service times, and C refers to the number of servers in the system.

For the [M/M/1 queue](https://en.wikipedia.org/wiki/M/M/1_queue), the Kendall's notation is M/M/1, where "M" indicates that the interarrival times and service times follow an exponential distribution, and "1" indicates that there is only one server in the system. Therefore, the M/M/1 queue is a specific type of queueing system that assumes a [Poisson arrival process](https://en.wikipedia.org/wiki/Poisson_point_process) and an exponential service distribution with a single server.

Kendall's notation is useful in characterizing and analyzing queueing systems, as it provides a standardized way of describing the system's behavior and performance. It helps to compare different queueing systems and make predictions about their behavior under different scenarios.

For example, in the M/M/1 queue, if the arrival rate of customers is greater than the service rate, the queue will grow indefinitely. Conversely, if the arrival rate is less than the service rate, the queue will eventually empty. As the arrival rate approaches the service rate, the average queue length and waiting time in the queue increase indefinitely, and the system becomes unstable.

When analyzing queueing systems, [numerical](https://en.wikipedia.org/wiki/Numerical_analysis) or [exact methods](https://en.wikipedia.org/wiki/Closed-form_expression) are often used to determine the system's performance measures such as the average number of customers in the system, the average waiting time, and the server utilization. However, these methods may become computationally complex and time-consuming, especially for more complex systems. In such cases, simulation-based methods can be used to efficiently estimate these measures and analyze system behavior.

Regarding the words "queueing" and "queueoid," "queueing" refers to the process of forming a line or a queue, typically used in the context of waiting in line for a service or product. "Queueoid" is a less commonly used term that refers to something that resembles or acts like a queue. Queueing has 5 consecutive vowels and queueoid has 6. It’s also interesting think of the letters u and e lining up inside the word queue forming a line inside the word itself!

## Queues ‘R’ Us Code

Please see the Excel or Visual Basic based simulation provided in the course learning management system in Module 1.

# Stock Market Follies

Simulation can be used to analyze a portfolio of various stocks by modeling the stochastic nature of stock prices. Stock prices are known to be highly volatile and subject to unpredictable changes due to various market and economic factors. Stock prices can be modeled as random variables that follow a stochastic process, such as the [geometric Brownian motion model](https://en.wikipedia.org/wiki/Geometric_Brownian_motion).

The stochastic process of stock prices is characterized by its [drift](https://en.wikipedia.org/wiki/Stochastic_drift) and [volatility](https://en.wikipedia.org/wiki/Volatility_(finance)). The drift represents the average rate of return of the stock, while volatility represents the standard deviation of the stock returns. The volatility of a stock is a measure of the uncertainty or risk associated with the stock. Stocks with higher volatility are riskier investments, as they are more likely to experience large price swings.

The simulation of a portfolio of stocks involves modeling the stochastic processes of each stock and combining them into a portfolio. The mix of stocks in a portfolio can be adjusted to achieve a desired level of risk and return. A portfolio with a higher mix of stocks with higher volatility will have a higher risk and potential for higher returns, while a portfolio with a lower mix of high-volatility stocks will have lower risk and potential for lower returns.

Simulation allows for the analysis of various scenarios and the estimation of the probability of certain outcomes. This can be useful in making investment decisions and managing risk. For example, simulation can be used to estimate the probability of a portfolio experiencing a certain level of loss or achieving a certain level of return over a certain period of time.

One limitation of simulation is that it is based on assumptions and models that may not accurately reflect the real-world behavior of the stock market. In addition, simulation requires the specification of various input parameters, such as the mean and standard deviation of stock returns, which may be difficult to estimate accurately. Furthermore, simulation can be computationally intensive and may require significant computing resources for large portfolios.

## Stock Market Follies Code

Please see the Excel or Visual Basic based simulation provided in the course learning management system in Module 1.

# Taking a Random Walk

[Random walk](https://en.wikipedia.org/wiki/Random_walk) is a mathematical model that describes the behavior of a variable that changes randomly over time. In the context of taking a normal step up or down every time unit, it means that at each time step, the variable takes a random value that can be higher or lower than the current value by a certain amount, which is determined by a probability distribution.

As the time steps become smaller and the number of steps becomes larger, random walk converges to [Brownian motion](https://en.wikipedia.org/wiki/Brownian_motion), which is a [continuous-time stochastic process](https://en.wikipedia.org/wiki/Continuous_stochastic_process) that describes the random movement of particles in a fluid or gas. In Brownian motion, the position of a particle changes continuously over time, and its velocity is proportional to the square root of time.

[Einstein](https://en.wikipedia.org/wiki/Albert_Einstein) was the first to provide a mathematical explanation for Brownian motion in 1905, which helped to confirm the existence of atoms and molecules. [Black](https://en.wikipedia.org/wiki/Fischer_Black) and [Scholes](https://en.wikipedia.org/wiki/Myron_Scholes), on the other hand, applied the concept of Brownian motion to develop the [Black-Scholes model](https://en.wikipedia.org/wiki/Black%E2%80%93Scholes_model), which is a widely used method for pricing options. Their contributions to the field of finance earned them the [Nobel Prize in Economics](https://en.wikipedia.org/wiki/Nobel_Memorial_Prize_in_Economic_Sciences) in 1997.

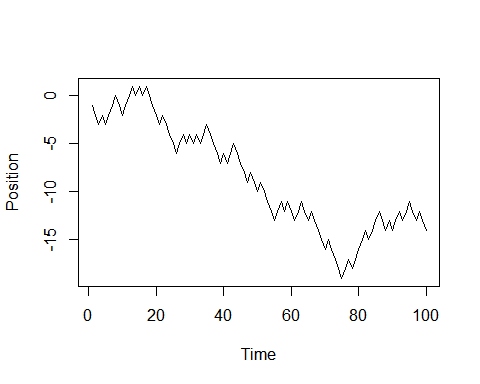
## Taking a Random Walk Example R Code

This code generates a random walk with 100 steps, where the direction of each step is determined randomly by sampling from a set of values that only contains -1 and 1. The set.seed(123) line is used to ensure that the same sequence of random numbers is generated each time the code is run.

The cumsum() function is used to calculate the cumulative sum of the steps, which gives the position of the random walk at each time step. The resulting vector of positions is stored in the variable random\_walk.

Finally, the code creates a plot of the random walk using the plot() function. The resulting plot shows the position of the random walk over time.

*# Set seed for reproducibility*  
set.seed(123)  
  
*# Generate random walk*  
steps <- sample(c(-1, 1), size = 100, replace = TRUE)  
random\_walk <- cumsum(steps)  
  
*# Plot random walk*  
plot(random\_walk, type = "l", xlab = "Time", ylab = "Position")



## Taking a Random Walk Example Python Code

[Python Notebook](https://colab.research.google.com/drive/1L5pHD4MARFccOBN14kKnVyrA9qSdFM2j#scrollTo=dhnAWmyeQ_oM)