1. Introduction

Often in the Stochastic Calculus for finance subjects, we want to focus on the application of stochastic calculus methos in fincance with both discrete-time and continuous-time stochastic models of financial markets. One of the most fundemental models is the Brownian Motion model, which lies the fundentamental for alot of other models. These models will in this project be presented and examplified using python!

History of Brownian Motion

The Brownian Motion was first introduced by botanist Robert Brown who observed the random movement of pollen particles due to water molecules under a microscope (*SimTrade*). The theory is based on the thought that price fluctuations observed over a small time period are independent of the current price along with historical behavior of price movements. Combined with the assumption of the Central Limit Theorem, it was proved that the random behavior of prices can be said to be represented by a normal distribution (Gaussian Distribution). This led to Brownian Motion becoming one of the most important fundemental of moden quantitative finance.

All in all, this led to the development of the Random Walk Theory we know today. A random walk is a statistical phenomenon where the stock prices move randomly and the best way to predict tomorrows price is to assume it will be the todays. However, when the time step of a random walk is made infinitesimally small, the random walk becomes a *Brownian Motion*.

In [7]:
```
# We will here import the most used libraries and packages in order to save space later on
import random as rd
import numpy as np
import plotly.express as px
import yfinance as yf
```

2. Random Walk

The basis of a Brownian Motion, as mentioned above, is a Random Walk. A random walk can be defined as follows:

- The position X_n is given by:
  - $X_n = X_{n-1} + Z_n, wheren >= 1$

Here the Z_n is a sequence of independent and identically distributed random variables. The intepretation of the formula could be that the variable X_n marks the position of the walk at time *n*. At each time the walk chooses a step at random- with the same step distribution at each time- and adds the result to its current position.

Simple Random Walk A Simple Random Walk is 1-dimensional version of the Random walk. Here the Z_1 takes values in {+1, -1} and the walk X_n starts from 0. However, X_1 takes both values with equal probabilites:

In [8]:
```
# Our Simple Random Walk function
def simulate_rd_walk(n_steps=1000, p=0.5, step_size=1) -> list:
    steps = [1*step_size if rd.random() < p else -1*step_size for i in range(n_steps)]
    y = np.cumsum(steps)
    x = list(range(len(y)))
    return x, list(y)


simulations = {}

# Choosing the number of simulated assets we want
num_sim = 8
for i in range(num_sim):
    x, y = simulate_rd_walk()
    simulations["Steps"] = x
    simulations["asset{col}".format(col=i)] = y

y_cols = ["asset{col}".format(col=i) for i in range(num_sim)]
fig = px.line(simulations, x="Steps", y=y_cols)
fig.update_layout(title="Simple Random Walk")
fig.show()
```

Brownian Motion / Wiener Process

Stochastic processes describe variables that evolve unpredictably over time. They can be delineated based on two characteristics: the nature of time (discrete vs. continuous) and the type of variable (continuous vs. discrete). In a discrete-time stochastic process, variable changes occur only at specific, predetermined time intervals. Contrastingly, in a continuous-time process, changes can transpire at any given moment.

Among these processes, the Markov process stands out. In a Markov process, only the present state of a variable is significant for forecasting its future states. This means that no matter how the current state was reached, it has no bearing on future probabilities. For instance, when considering stock prices which are often assumed to follow a Markov process, knowing the price of a stock right now (say, $100) provides all the information needed to predict its future movements. The price of that stock from a week, a month, or a year ago wouldn't change our predictions.

The Wiener process is a specific example of a Markov process. Named after Norbert Wiener, it characterizes a variable that follows a certain type of stochastic progression. The distinctive traits of the Wiener process are that its average change is zero, and it has a variance rate of 1.0 per annum.

In [9]:
```
def simulate_bm(n_steps=1000, t=0.01):
    steps = [np.random.randn()*np.sqrt(t) for i in range(n_steps)]
    y = np.cumsum(steps)
    x = [t*i for i in range(n_steps)]
    return x, y


simulations = {}
# Choosing the number of simulated assets we want
num_sim = 8
for i in range(num_sim):
    x, y = simulate_bm()
    simulations["x"] = x
    simulations["asset{col}".format(col=i)] = y

y_cols = ["asset{col}".format(col=i) for i in range(num_sim)]
fig = px.line(simulations, x="x", y=y_cols)
fig.update_layout(title="Brownian Motion")
fig.show()
```

Brownian Motion with Drift

A Brownian Motion with drift is also abbreviated as Generelaized Wiener process. For a stochastic process, the average change over a given time interval is termed the "drift rate," while the dispersion or spread of this change during the same interval is referred to as the "variance rate." In the basic Wiener process, represented by dz, the drift rate is set to zero, and the variance rate is 1.0. A drift rate of zero implies that the expected value of z at any forthcoming point in time remains consistent with its present value. On the other hand, a variance rate of 1.0 denotes that the fluctuation or change in z over a time span T is directly proportional to T. We can further extrapolate a generalized Wiener process for a variable x as follows:

- dx=adt+bdz Where a and b are constants.

In [10]:
```
def simulate_bm_drift(n_steps=1000, t=0.01, mu=0.5):
    steps = [mu*t+np.random.randn()*np.sqrt(t) for i in range(n_steps)]
    y = np.cumsum(steps)
    x = [t*i for i in range(n_steps)]
    return x, y


simulations = {}
# Choosing the number of simulated assets we want
num_sim = 8
for i in range(num_sim):
    x, y = simulate_bm_drift()
    simulations["x"] = x
    simulations["asset{col}".format(col=i)] = y

y_cols = ["asset{col}".format(col=i) for i in range(num_sim)]
fig = px.line(simulations, x="x", y=y_cols)
fig.update_layout(title="Brownian Motion with Drift")
fig.show()
```

Geometric Brownian Motion

Often, one might assume that stock prices adhere to a generalized Wiener process, meaning they exhibit a consistent drift and variance rate. But such an assumption misses a fundamental characteristic of stock prices: the expected percentage return an investor desires from a stock doesn't hinge on its price. For instance, if an investor anticipates a 14% annual return from a stock priced at 10 dollar, they would still expect the same 14% return if the stock's price rises to 50 dollar, all other factors remaining constant.

This observation challenges the idea of a constant drift rate. Instead, it suggests that the expected return, which is the drift rate relative to the stock price, remains fixed. If S denotes the stock price at time t, the expected drift rate for S should be represented as mS, where m is a stable parameter. In simpler terms, over a brief time span Δt, the predicted rise in S is mSΔt. Here, m represents the stock's anticipated rate of return.

A discrete-time representation of this model is:

- $\Delta S = m * S \Delta t + s * S * Z * (\Delta t)^{0.5}$

In this equation, ΔS signifies the alteration in the stock price S within the small duration Δt. The term Z follows a standard normal distribution, which means it has an average of zero and a standard deviation of 1.0. While m is the expected rate of return for each

In [11]:

```
def simulating_g_b_mot(n_steps= 1000, t=1, mu=0.0001, sigma=0.02):
    steps = [(mu - (sigma**2)/2)*t + np.random.randn()*sigma for i in range(n_steps)]
    y = np.exp(np.cumsum(steps))
    x = [t*i for i in range(n_steps)]
    return x, y


n_sims = 8
simulations = {}
for i in range(n_sims):
    x, y = simulating_g_b_mot()
    simulations["asset{col}".format(col=i)] = y
    simulations["x"] = x


y_cols = ["asset{col}".format(col=i) for i in range(n_sims)]
fig = px.line(simulations, x="x", y=y_cols)
fig.update_layout(title="Geometric Brownian Motion simulation")
fig.show()
```

Simulating Stock Prices with GBM

We can also apply the Geometric Brownian Motion theorem in a real life scenario. Below I choose to apply the GBM on the General Motor stock, in order to simulate how the stock price development could look like. The number of steps could be seen as the number of days in the future we want to look at.

In [12]:
```
# First we create our Geo_Brw_Motion function
def simulating_g_b_mot(n_steps=1000, t=1, mu=0.0001, sigma=0.02, start=1):
    steps = [(mu - (sigma**2)/2)*t + np.random.randn()*sigma for i in range(n_steps)]
    y = start*np.exp(np.cumsum(steps))
    x = [t*i for i in range(n_steps)]
    return x, y

# Setting our start and end date
start = "2015-01-01"
end = "2023-08-23"

# Selecting the necessary tickers, for each stock/index
ticker = ["GM"]

# Extracting the data from the ticker list
df = yf.download(ticker, start, end)

# Renaming the columns
df.rename(columns={"Adj Close": "AdjC"}, inplace=True)

# We only want our adjusted closing price
array_prices = df.AdjC.values

# Printing out the array in order to see what we are working
print(array_prices)


# We convert from arithmetic values to logarithmic values, and we find the return by taking (p1-p0)/p0
# First we flip the array from a descending order, starting from todays price and then ordered backwards
prices = np.flip(array_prices)

log_prices = np.log(prices)
log_returns = log_prices[1:]-log_prices[:-1]
print(log_returns)

# Our Mu in the formula is the average of all observed log_returns
obs_mu = np.mean(log_returns)
obs_mu = float(obs_mu)

# Our sigma in the formula is the standard deviation of all observed log_returns
obs_sigma = np.std(log_returns)
# Since performing the calculations from the Yfinance and Dataframe, we need to specify that these are float
obs_sigma = float(obs_sigma)
```

```
# The number of steps
obs_steps = log_prices.shape[0]

# calling our function
x, y= simulating_g_b_mot(n_steps=obs_steps, mu=obs_mu, sigma=obs_sigma, start=prices[0])


data = {"x": x, "simulation": y, "GM": prices}
fig = px.line(data, x="x", y=["simulation", "GM"])
fig.update_layout(
    xaxis_title="Time",
    yaxis_title="GM price and Simulation"
)
fig.show()
```

```
[**********100%%**********]  1 of 1 completed
[27.5384407  27.13532257 27.54633713 ... 33.03072739 33.18032455
 32.84123993]
[ 0.01027203 -0.00451881 -0.00605701 ... -0.02801161 -0.01503328
  0.01474658]
```

Sources

- Simtrade: https://www.simtrade.fr/blog_simtrade/brownian-motion-finance/ (https://www.simtrade.fr/blog_simtrade/brownian-motion-finance/)
- Math UCLA, Random Walks: https://www.math.ucla.edu/~biskup/PDFs/PCMI/PCMI-notes-1 (https://www.math.ucla.edu/~biskup/PDFs/PCMI/PCMI-notes-1)
- Investopedia Random Walk: https://www.investopedia.com/terms/r/randomwalktheory.asp (https://www.investopedia.com/terms/r/randomwalktheory.asp)

- John Hull - Options, Futures, and Other Derivatives, Global Edition-Pearson (2021)