# 1 Monte Carlo Payoff Function Simulation

The following project uses the Monte Carlo Method to calculate the prospective prices of different call options. The function is derived from the Black-Scholes Formula. The logic of the functions is to calculate the cumulative sum of payoff to generate an expected or average value by the law of large numbers.

$$\frac{1}{n}\sum_{i=1}^{n} C_i \longrightarrow^{n\to+\infty} E[C_i] = E[e^{-rT}(S(T)-K)]$$

### 1.0.1 European option

- generate normal observations for i={1,2,...n} $n$ number of paths: $Z_i$ ~ N(0,1)
- generate stock price observations:

$$S_i(T) = S(0)exp[(r - \frac{1}{2}\sigma^2)T + \sigma\sqrt{T}Z_i]$$

- calculate payoff:

$$C_i = e^{-rT}(S_i(T) - K)^t$$

- comupte cumulative sum:

$$\hat{C}_n = \frac{n}{i=1}C_i$$

### 1.0.2 Asian option

$$\Delta t = \frac{T}{m}$$

$$S_{t+\Delta t} \approx S_t + rS_t\Delta t + \sigma S_t\sqrt{\Delta t}Z$$

- generate normal observations for i={1,2,...n} $n$ and j={1,2,...m} number of paths: $Z_{ij}$ $N(0,1)$
- generate stock price observations:

$$S^i_{t_j} = S^i_{t_{j-1}} + rS^i_{j-1}\Delta t + \sigma S^i_{j-1}\sqrt{\Delta t}Z_{ij}$$

- calculate cumulative stock price:

$$\bar{S} = \frac{1}{m}\sum_{j=1}^{m} S^i_{t_j}$$

- calculate payoff:

$$C_i = e^{-rT}(\bar{S} - K)^t$$

- comupte cumulative sum:

$$\hat{C}_n = \sum_{i=1}^{n} C_i$$

[6]:
```python
import numpy as np

class Monte_Carlo():
    def __init__(self, n, r, sig, K, S, T):
        self.n, self.r, self.sig, self.K, self.S, self.T = n, r, sig, K, S, T
        self.euro = self.european_option(n, r, sig, K, S, T)
        self.asian = self.asian_option(n, r, sig, K, S, T)

    def european_option(self, n, r, sig, K, S, T ):
        # generating standard normal random observations
        Z_i = np.random.standard_normal(n)
        # generating stock prices as function of observations
        S_i = [S*np.exp((r - 0.5*sig**2)*T + sig*np.sqrt(T)*z) for z in Z_i]
        # generating option (call) price as a function of stock prices
        # using max to ensure non negative payoff value
        C_i = [np.exp(-r*T)*max(s - K, 0) for s in S_i]
        # summing observations
        C_n = (1/n)*np.sum(C_i, axis=0)
        return C_n

    def asian_option(self, n, r, sig, K, S, T ):
        m = 1000 # initialized to 1000 according to assignment requirements
        # generating n by m matrix (n=i, m=j)
        Z_ij = np.random.standard_normal((n,m))
        # initializing terms
        S_matrix = np.empty((n, m))
        t_delta = T/m
        # initializing first column as stock price
        # necessary to drop this b4 calculation?
        S_matrix[:, 0] = S

        for i in range(n):
            for j in range(1, m):
                S_matrix[i,j] = S_matrix[i,j-1]*(1 + r*t_delta + sig*np.
 ↪sqrt(t_delta)*Z_ij[i][j])

        # computing column sum
        S_bar = (1/m)*np.sum(S_matrix, axis=1)
        S_bar = S_bar.tolist()
        # sum observations, applying payoff functoin
        C_i = [np.exp(-r*T)*max(s - K, 0) for s in S_bar]
        C_n = (1/n)*np.sum(C_i, axis=0)
        return C_n
```

[7]:
```python
paths = [100, 1000, 10000, 100000]
mc_objs = [Monte_Carlo(n=path, r=0.03, sig=0.2, K=100, S=100, T=1) for path in␣
 ↪paths]
```

[8]:

```
np.random.seed(1) # initializing random seed for replicability
resultE = [obj.euro for obj in mc_objs]
print(resultE)
```

```
[8.455082111853704, 9.384540196308572, 9.653443248841926, 9.4327969785097]
```

[9]:

```
np.random.seed(1)
resultA = [obj.asian for obj in mc_objs]
print(resultA)
```

```
[5.710021291677132, 5.164431576749103, 5.180880528951596, 5.313919560848155]
```