# HW03: Exotic options.


- Author: alberto.suarez@uam.es
- Student 1: Gorka Crespo Bravo
- Student 2: Miguel Cuesta Altable
- Student 3: Antón Salvadores Muñiz


Please use latex for the derivations. Alternatively, insert a scanned image of the derivations, as it is here done with the Alan Turing [https://www.turing.org.uk/] picture.

 alt text

In [ ]:
```
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
import option_pricingGRAGAS as pricing

import scipy.stats as stats

from scipy.stats import norm, lognorm
from numpy.random import default_rng

from pandas.plotting import autocorrelation_plot
from scipy.stats import norm, probplot
from scipy.optimize import minimize

from tools_qfb import compare_histogram_pdf, qqplot
from my_stochastic_processes import simulate_geometric_brownian_motion

%load_ext autoreload
%autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

## Exercise 1: Option pricing over several underlyings.

The goal of this exercise is to price a European call option over three correlated assets whose payoff is

$$\text{payoff} = \max\left(c_1 S_I(t_0+T) + c_2 S_{II}(t_0+T) + c_3 S_{III}(t_0+T) K, 0\right).$$

### Exercise 1.1:

Consider the correlation matrix

$$\boldsymbol{\rho} = \begin{pmatrix} 1.0 & \rho_{12} & \rho_{13} \\ \rho_{12} & 1.0 & \rho_{23} \\ \rho_{13} & \rho_{23} & 1.0 \end{pmatrix}.$$

Derive the expression of the Cholesky decomposition of $\boldsymbol{\rho}$. That is, the elements of the lower triangular matrix

$$\boldsymbol{\rho} = \mathbf{L}\mathbf{L}^T,$$

where

$$\mathbf{L} = \begin{pmatrix} L_{11} & 0.0 & 0.0 \\ L_{12} & L_{22} & 0.0 \\ L_{13} & L_{23} & L_{33} \end{pmatrix},$$

and $\mathbf{L}$ its transpose.

Si hacemos la multiplicacion de $\mathbf{L}$ por su transpuesta $\mathbf{L}^T$ nos queda la siguiente matiz:

$$\mathbf{L}\mathbf{L}^T = \begin{pmatrix} L_{11}^2 & L_{11}L_{12} & L_{11}L_{13} \\ L_{11}L_{12} & L_{12}^2 + L_{22}^2 & L_{12}L_{13} + L_{22}L_{23} \\ L_{11}L_{13} & L_{12}L_{13} + L_{22}L_{23} & L_{13}^2 + L_{23}^2 + L_{33}^2 \end{pmatrix}$$

Si aplicamos la igualdad de la descomposicion de cholesky obtenemos lo siguente:

$$\boldsymbol{\rho} = \mathbf{L}\mathbf{L}^T \begin{pmatrix} 1.0 & \rho_{12} & \rho_{13} \\ \rho_{12} & 1.0 & \rho_{23} \\ \rho_{13} & \rho_{23} & 1.0 \end{pmatrix} = \begin{pmatrix} L_{11}^2 & L_{11}L_{12} & L_{11}L_{13} \\ L_{11}L_{12} & L_{12}^2 + L_{22}^2 & L_{12}L_{13} + L_{22}L_{23} \\ L_{11}L_{13} & L_{12}L_{13} + L_{22}L_{23} & L_{13}^2 + L_{23}^2 + L_{33}^2 \end{pmatrix}$$

Con esta igualdad podemos sacar las siguientes ecuaciones:

- Ecuación 1:

$$1 = L_{11}^2$$

- Ecuación 2:

$$\rho_{12} = L_{11}L_{12}$$

- Ecuación 3:

$$\rho_{13} = L_{11}L_{13}$$

- Ecuación 4:

$$\rho_{12} = L_{11}L_{12}$$

- Ecuación 5:

$$1 = L_{12}^2 + L_{22}^2$$

- Ecuación 6:

$$\rho_{23} = L_{12}L_{13} + L_{22}L_{23}$$

- Ecuación 7:

$$\rho_{13} = L_{11}L_{13}$$

- Ecuación 8:

$$\rho_{23} = L_{12}L_{13} + L_{22}L_{23}$$

- Ecuación 9:

$$1 = L_{13}^2 + L_{23}^2 + L_{33}^2$$

Si obviamos las ecuaciones equivalentes nos queda un sistema de 6 ecuaciones y 6 incognitas con las que podremos construir la matriz de Cholesky. Ahora vamos a depejar las diferentes $L_{ij}$ para contruir la matriz:

- Despejando $L_{11}$:

$$L_{11} = 1$$

- Despejando $L_{12}$:

$$L_{12} = \frac{\rho_{12}}{L_{11}} = \rho_{12}$$

- Despejando $L_{13}$:

$$L_{13} = \frac{\rho_{13}}{L_{11}} = \rho_{13}$$

- Despejando $L_{22}$:

$$L_{12}^2 + L_{22}^2 = 1 \quad \Rightarrow \quad L_{22} = \sqrt{1 - L_{12}^2} = \sqrt{1 - \rho_{12}^2}$$

- Despejando $L_{23}$:

$$L_{12}L_{13} + L_{22}L_{23} = \rho_{23} \quad \Rightarrow \quad L_{23} = \frac{\rho_{23} - L_{12}L_{13}}{L_{22}} = \frac{\rho_{23} - \rho_{12} \cdot \rho_{13}}{\sqrt{1 - \rho_{12}^2}}$$

- Despejando $L_{33}$:

$$L_{13}^2 + L_{23}^2 + L_{33}^2 = 1 \quad \Rightarrow \quad L_{33} = \sqrt{1 - L_{13}^2 - L_{23}^2} = \sqrt{1 - \rho_{13}^2 - \left( \frac{\rho_{23} - \rho_{12} \cdot \rho}{\sqrt{1 - \rho_{12}^2}} \right.}$$

Por lo que, la matriz de Cholesky queda de la siguiente manera:

$$\mathbf{L} = \begin{pmatrix} 1 & 0 & 0 \\ \rho_{12} & \sqrt{1 - \rho_{12}^2} & 0 \\ \rho_{13} & \frac{\rho_{23} - \rho_{12} \cdot \rho_{13}}{\sqrt{1 - \rho_{12}^2}} & \sqrt{1 - \rho_{13}^2 - \left( \frac{\rho_{23} - \rho_{12} \cdot \rho_{13}}{\sqrt{1 - \rho_{12}^2}} \right)^2} \end{pmatrix}$$

## Exercise 1.2:

Using the expressions derived in the previous exercise or (better) the `numpy` funtion `np.lingalg.cholesky`, to compute the MC estimate of the price (and standard deviation) of a basket call option with the parameters specified in the following cell:

In [ ]:
```
# Parameters of the underlyings.

t_0 = 0.0
S_0_1, mu_I, sigma_1 = 100.0, 0.15, 0.3
S_0_2, mu_II, sigma_2 = 150.0, -0.05, 0.2
S_0_3, mu_III, sigma_3 = 80.0, 0.2, 0.25

rho = np.array(
    [[ 1.0,  0.4, -0.3],
     [ 0.4,  1.0,  0.6],
     [-0.3,  0.6,  1.0]]
)

# Parameters of the option.
r = 0.05 # Risk-free interest rate.
T = 2.5  # lifetime of the option.

c = np.array([1.0, 1.5, 2.0])
K = 500.0 # Strike.

# You can use the expressions derived in the previous exercise
# or the Cholesky decomposition of the correlation matrix.
print(np.linalg.cholesky(rho))
```

```
[[ 1.         0.         0.        ]
 [ 0.4        0.91651514 0.        ]
 [-0.3        0.7855844  0.54116277]]
```

In [ ]:
```
# Your code goes here.

payoff = (lambda S_T_1, S_T_2, S_T_3:
          np.maximum(c[0] * S_T_1 + c[1] * S_T_2 + c[2] * S_T_3 - K, 0.0))
price_MC, stdev_MC  = pricing.price_european_3_underlyings_MC(
       S_0_1, sigma_1, S_0_2, sigma_2, S_0_3, sigma_3, rho, r, T, payoff,
       n_simulations=int(1.0e5), seed=0)
print('price (MC) = {:.3f} ({:.3f})'.format(price_MC, stdev_MC))
```

```
price (MC) = 75.975 (0.339)
```

# Exercise 2: Barrier options.

A barrier option is a derivative option whose payoff depends on whether the trajectory of the underlying prices has been above or below certain thresolds (barriers) at specified instants during the lifetime of the option.

## Up-and-out barrier option.

An up-and-out call barrier option with has a payoff at maturity that is different from zero only if the level of the underlying is always below a threshold $B_{up}$:

$$\text{Payoff}_{up-and-out} = \begin{cases} \max\left(S(t_0+T)-K, 0\right) & \text{if } S(t) < B_{up}, \quad \forall t \in [t_0, t_0+T], \\ & \text{otherwise.} \end{cases}$$

The price of this option cannot computed exactly because one needs continuous monitoring of $S(t)$ for $t \in [t_0, t_0+T]$ to determine whether or not there has been a barrier crossing.

However, we can approximate its price by monitoring barrier crossings at a discrete set of times $\mathbf{t} = \{t_0, t_1, \ldots, t_N\}$. That is, the price of the barrier option with countinuous monitoring of barrier crossings can be approximated by the price of a barrier option whose payoffs is different from zero only if the underlying is above the barrier at $\mathbf{t} = \{t_0, t_1, \ldots, t_N\}$.

In [ ]:
```
# MC pricing of an up-and-out barrier option.

# Parameters of the underlying.
t_0 = 0.0
S_0, mu, sigma = 100.0, 0.15, 0.3

# Parameters of the option.
r = 0.05 # Risk-free interest rate.
T = 2.5  # lifetime of the option.
K = 90.0 # Strike.
B_up = 150.0 # Barrier.

# Parameters of the simulation.
n_trajectories = 10000
n_times = 10

# Simulation.
times =  np.linspace(t_0, t_0 + T, num=n_times)   # simulation grid
S = simulate_geometric_brownian_motion(times, S_0, r, sigma, n_trajectories, see

# Payoffs.

payoff_call = lambda S_T: np.maximum(S_T - K, 0.0)
index_up_and_out = np.all(S < B_up, axis=1)

payoff_MC = payoff_call(S[:, -1]) * index_up_and_out

# MC pricing.

discount_factor = np.exp(-r * (times[-1] - times[0]))

price_MC = discount_factor * np.mean(payoff_MC)
stdev_MC = discount_factor * np.std(payoff_MC) / np.sqrt(n_trajectories)

print('price (MC) = {:.2f} ({:.2f})'.format(price_MC, stdev_MC))
```

```
price (MC) = 5.95 (0.12)
```

In [ ]:
```
# Trace the MC simulation and pricing process.

n_trajectories_print = 5
```

```
print('Simulation times:')
print(np.around(times, 3))
print('\nTrajectories simulated:')
print(np.around(S[:n_trajectories_print, :], 1))
print('\nPayoff European vanilla call:')
print(np.around(payoff_call(S[:n_trajectories_print, -1]), 2))
print('\nTrajectories below the barrier:')
print(index_up_and_out[:n_trajectories_print])
print('\nPayoff up-and-out call option:')
print(np.around(payoff_MC[:n_trajectories_print], 2))
```

```
Simulation times:
[0.    0.278 0.556 0.833 1.111 1.389 1.667 1.944 2.222 2.5  ]

Trajectories simulated:
[[100.  105.8 120.6 127.2 103.7 119.8 128.8 118.4 130.  137.9]
 [100.  104.9 105.5 115.2 102.7 100.2  93.  102.4 103.2  98.6]
 [100.   88.5  85.1  85.3  81.8 100.5 118.   77.   57.2  55.7]
 [100.   93.7  97.  100.6 140.7 118.2 111.5 154.3 171.1 190.3]
 [100.   92.3  71.2  73.3  74.6  61.6  55.3  54.8  47.2  46.6]]

Payoff European vanilla call:
[ 47.93   8.63   0.   100.27   0.  ]

Trajectories below the barrier:
[ True  True  True False  True]

Payoff up-and-out call option:
[47.93  8.63  0.    0.    0.  ]
```

## Exercise 2.1: Pricing a simple barrier option.

1. Order from smaller to larger the price of the following options with the same common characteristics (lifetime of the option, strike, location of the barrier, etc.)

   A. A European call option.
   B. An up-and-out call option in which the barrier crossings are monitored continuously.
   C. An up-and-out call option with discrete monitoring of barrier crossings.
   D. An option with two barriers, in which the barrier crossings are monitored continuously. The upper barrier is the same as in (B) and (C). The payoff is different from zero only if the price of the underlying remains within the band defined by these two barriers during the lifetime of the option.

2. Indicate the types of errors that are made when the price of the barrier option is estimated using the procedure specified. Explain their origin and relative importance.

3. Only for the types of errors that are significant:

   A. Indicate on which parameter of the simulation they depend and what form the dependence takes (order of convergence for the approximation).
   B. Describe two different methods that can be used to reduce each of the different types of errors identified.

**Apartado 1:**

Para ordenar las siguientes opciones según su precio, debemos analizar cuál de ellas es la más restrictiva en cuanto a sus barreras. Las opciones más restrictivas tienden a tener un precio menor, ya que es menos probable que lleguen a pagar algo al vencimiento, considerando que todas tienen los mismos parámetros $T$, $K$, $B_{up}$, etc.

La más restrictiva es la **opción con dos barreras y monitoreo continuo**, ya que es la que tiene menos probabilidades de "sobrevivir" hasta el vencimiento, y por tanto, es la más barata. A continuación, está la **opción up-and-out con monitoreo continuo**, que también pierde todo su valor si se toca la barrera superior en cualquier momento, aunque es menos restrictiva que la anterior, ya que en un momento $t_0 < t_k < T$ el valor del subyacente podría tocar la barrera inferior y pagar al final, en el caso anterior la opción expiraría. Luego, sigue la **opción up-and-out con monitoreo discreto**, que es aún menos restrictiva, ya que solo se invalida si la barrera se toca en los momentos específicos de monitoreo; si el precio la sobrepasa entre esos momentos, la opción no se anula. Finalmente, la más cara es la **opción europea tipo call**, ya que no tiene ninguna barrera que limite su valor.

El orden final sería:

**Opción con dos barreras y monitoreo continuo (4)** < **Up-and-out con monitoreo continuo (2)** < **Up-and-out con monitoreo discreto (3)** < **European call option (1)**.

**Apartado 2:**

Al hacer cálculos con el ordenador, siempre estamos cometiendo errores de redondeo, aunque en este caso son los menos relevantes, ya que no estamos realizando operaciones muy complejas o con números muy semejantes.

Por otro lado, como estamos tratando de aproximar una trayectoria (con un número infinito de puntos) a partir de **n_times** puntos, estamos cometiendo también errores de truncamiento, como hemos explicado en el apartado anterior. A medida que aumentemos el número de puntos, el precio del subyacente tocará más veces la barrera y será más barata la opción. Con cierta cautela hemos de aumentar el número de puntos para obtener una mejor aproximación del valor de la opción, ya que si aumentamos demasiado **n_times**, entonces $\Delta t \approx 0$ para el ordenador y no podremos obtener las trayectorias pertinentes. Aunque, efectivamente, fijado el valor de **n_trajectories**, cambia el valor de la opción si aumentamos o disminuimos el número de puntos de la malla temporal; nuestro medidor MC del error, la desviación estándar, no fluctúa de manera significativa, como podemos ver en el código de debajo; por tanto, no parece que este sea el tipo de error más relevante.

Sin embargo, los errores de muestreo, producidos por el valor dado para **n_trajectories**, sí que son relevantes, ya que depende el número de trayectorias que simulemos, tendremos una mejor aproximación de la media de los payoff para muchos valores del subyacente $S_T$. Por tanto, el error más relevante en los procedimientos Monte-Carlo concluimos que es el error de muestreo debido a estar generando muestras aleatorias.

**Apartado 3:**

Como hemos descrito antes, el parámetro más relevante para disminuir el error de muestreo es el número de trayectorias **n_trajectories** y como se puede ver en el código de debajo, a medida que aumentamos por un factor de $10$ el número de trayectorias, el error se disminuye por un factor $\approx \frac{1}{3}$. Este es su orden de convergencia.

Sugerimos como posibles métodos para mejorar el error de nuestra simulación, el primero y más obvio, aumentar el número de trayectorias para reducir el error, aunque este no parece el más óptimo, ya que como hemos descrito antes, para reducir un tercio el error debemos realizar $10$ veces más operaciones en el ordenador, lo que a la larga no va a dar grandes resultados. Por otro lado, podemos utilizar métodos para reducir la varianza del cálculo del precio de la opción como utilizar variables antitéticas.

In [ ]:
```
## Error de truncamiento

n_times = [1, 10, 100, 1000, 10000]
results = []

# Simulation.

for nt in n_times:
  times =  np.linspace(t_0, t_0 + T, num=nt)   # simulation grid
  S = simulate_geometric_brownian_motion(times, S_0, r, sigma, n_trajectories, s



  payoff_call = lambda S_T: np.maximum(S_T - K, 0.0)
  index_up_and_out = np.all(S < B_up, axis=1)

  payoff_MC = payoff_call(S[:, -1]) * index_up_and_out

  # MC pricing.

  discount_factor = np.exp(-r * (times[-1] - times[0]))

  price_MC = discount_factor * np.mean(payoff_MC)
  stdev_MC = discount_factor * np.std(payoff_MC) / np.sqrt(n_trajectories)
  results.append({
        'n_times': nt,
        'price_MC': price_MC,
        'stdev_MC': stdev_MC
    })

df = pd.DataFrame(results)

print(df.to_string(index=False))
```

```
n_times  price_MC  stdev_MC
      1 10.000000  0.000000
     10  5.953351  0.119244
    100  4.437409  0.100276
   1000  3.975814  0.094693
  10000  3.885691  0.092412
```

In [ ]:
```
n_trajectories = [100, 1000, 10000, 100000, 1000000]
n_times = 100
```

```
results = []

# Simulation.

for n in n_trajectories:
  times =  np.linspace(t_0, t_0 + T, num=n_times)   # simulation grid
  S = simulate_geometric_brownian_motion(times, S_0, r, sigma, n, seed=1)



  payoff_call = lambda S_T: np.maximum(S_T - K, 0.0)
  index_up_and_out = np.all(S < B_up, axis=1)

  payoff_MC = payoff_call(S[:, -1]) * index_up_and_out

  # MC pricing.

  discount_factor = np.exp(-r * (times[-1] - times[0]))

  price_MC = discount_factor * np.mean(payoff_MC)
  stdev_MC = discount_factor * np.std(payoff_MC) / np.sqrt(n)
  results.append({
        'n_trajectories': n,
        'price_MC': price_MC,
        'stdev_MC': stdev_MC
    })

df = pd.DataFrame(results)

print(df.to_string(index=False))
```

```
n_trajectories  price_MC  stdev_MC
           100  2.960552  0.814691
          1000  4.283794  0.308496
         10000  4.437409  0.100276
        100000  4.490103  0.031917
       1000000  4.486713  0.010087
```

## Exercise 2.2: Pricing a more complex barrier option.

Compute the MC price estimate and its standard deviation of a barrier call option whose payoff is different from zero if the price of the underlying is above $B_{down}$ in the interval $\left[t_0, t_0 + \frac{T}{2}\right]$ and below $B_{up}$ in the interval $\left[t_0 + \frac{T}{2}, t_0 + T\right]$.

In [ ]:
```
# MC pricing of an up-and-out barrier option.

# Parameters of the underlying.
t_0 = 0.0
S_0, mu, sigma = 100.0, 0.15, 0.3

# Parameters of the option.
r = 0.05 # Risk-free interest rate.
T = 2.5  # lifetime of the option.
K = 90.0 # Strike.
B_up = 150.0 # Barrier.
B_down = 80 #Low barrier

# Parameters of the simulation.
```

```
n_trajectories = 10000
n_times = 100

# Simulation.
times =  np.linspace(t_0, t_0 + T, num=n_times)
S = simulate_geometric_brownian_motion(times, S_0, r, sigma, n_trajectories, see

#Dividir la trayectoria en dos

midpoint = S.shape[1] // 2
S_1 = S[:, :midpoint]
S_2 = S[:, midpoint:]


# Payoffs.
payoff_call = lambda S_T: np.maximum(S_T - K, 0.0)
index_up_and_out_1 = np.all(S_1 > B_down, axis=1)
index_up_and_out_2 = np.all(S_2 < B_up, axis=1)

payoff_MC = payoff_call(S[:, -1]) * index_up_and_out_1 * index_up_and_out_2

# MC pricing.

discount_factor = np.exp(-r * (times[-1] - times[0]))

price_MC = discount_factor * np.mean(payoff_MC)
stdev_MC = discount_factor * np.std(payoff_MC) / np.sqrt(n_trajectories)

print('price (MC) = {:.2f} ({:.2f})'.format(price_MC, stdev_MC))
```

```
price (MC) = 2.76 (0.08)
```

Como se puede comprobar con los resultados obtenidos en el anterior apartado, el precio de esta opción es más barato que con una única barrera.

In [ ]:
```
!jupyter nbconvert --to html /content/HW_03_exercises.ipynb
from google.colab import files
files.download('HW_03_exercises.html')
```

```
[NbConvertApp] Converting notebook /content/HW_03_exercises.ipynb to html
[NbConvertApp] WARNING | Alternative text is missing on 1 image(s).
[NbConvertApp] Writing 417618 bytes to /content/HW_03_exercises.html
```