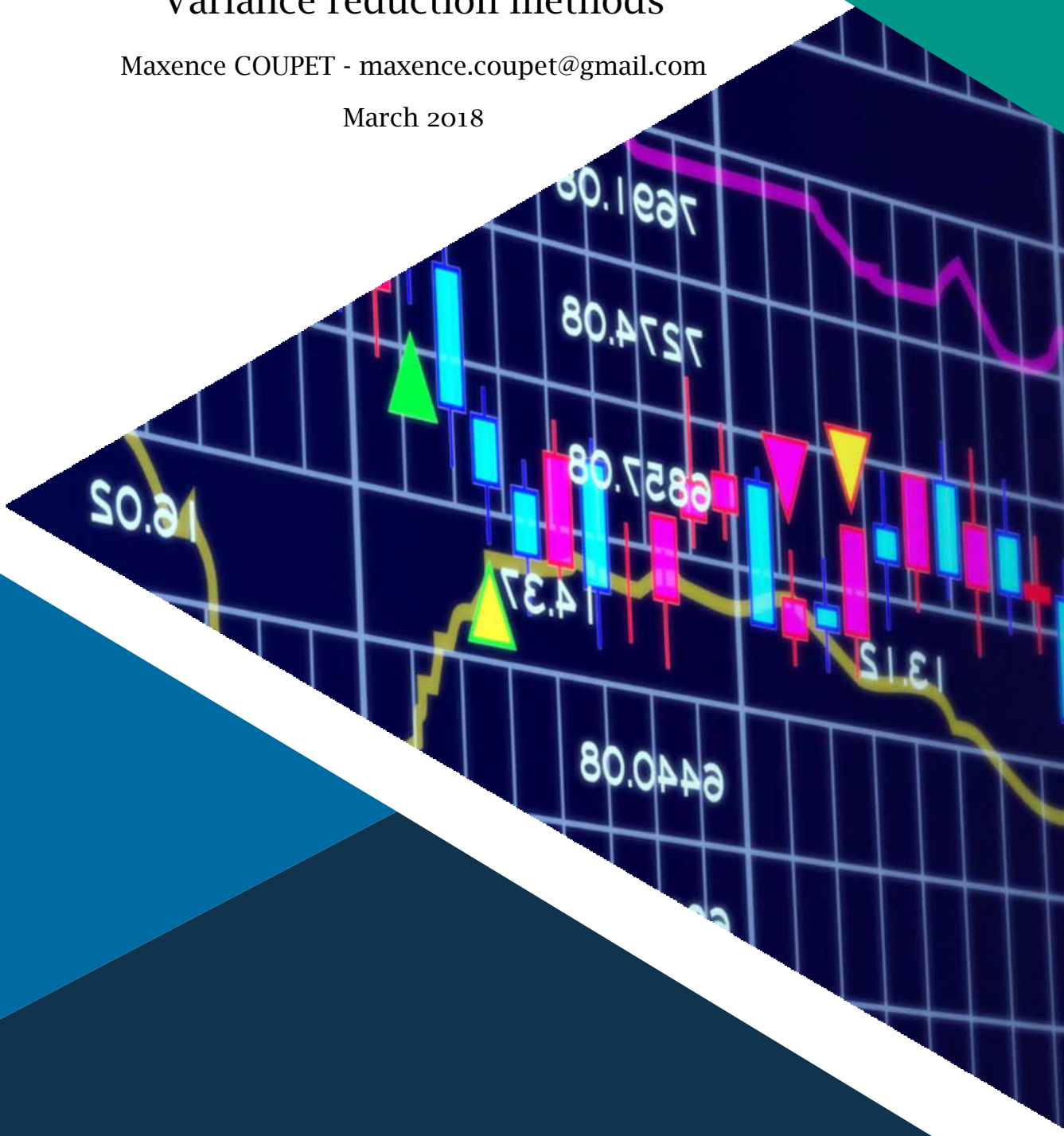
Variance reduction methods

Maxence COUPET - maxence.coupet@gmail.com

March 2018



When simulating stochastic processes in order to compute the price of a derivative, we have to compute a lot of paths in order to be accurate, which could be very time consumming. All the following methods allow one to make the simulated price converging faster to its limit.

1 Confidence bounds

Computing the price of a derivative using the Monte-Carlo method implies to generate N different paths for the underlying, then computing the payoff for each of these paths, computing the mean of all payoffs and finally discounting the value in order to get the present value of the derivative.

According to the central limit theorem, we have the following confidence bounds at a 95% level :

$$\hat{\mu} - \frac{1.96\hat{\sigma}}{\sqrt{N}} < derivative \quad price < \hat{\mu} + \frac{1.96\hat{\sigma}}{\sqrt{N}}$$

with $\hat{\mu}$ the empirical mean and $\hat{\sigma}$ the empirical standard variation.

With this formula we have only two options to improve the confidence bounds : increasing the number of simulations N or reducing $\hat{\sigma}$. The following methods are meant to reduce the variance and while they are presented individually they are often applied together.

We will test the method on a standard laptop within a single threaded program. While using multi threading is really usefull for reducing time of execution, it is used to increase the number of simulations and thus is not in the scope of this document which focuses on variance reduction.

2 Antithetic variates

The main idea with this method is to compute two values of the derivative price at each simulation. The first value will be computed as usual with the random path, but another will be computed with the opposite path. For example if for the path n the value at the time t for the underlying is : $S_n(t) = S_n(0)e^{(r-\frac{\sigma^2}{2})t+\sigma\epsilon}$, the value of the opposite path at the same time will be : $S_n^{'}(t) = S_n(0)e^{(r-\frac{\sigma^2}{2})t-\sigma\epsilon}$, where ϵ is a sample from a random variable following a standard normal law (see figure 1).

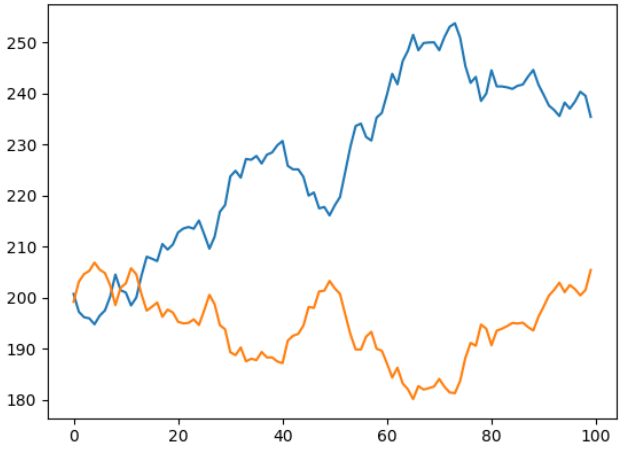


FIGURE 1 – Example of paths used for the antithetic variates method

It is very important to understand that we take opposite values for the standard normal random variable which represent the random part of the price, not the opposite values of prices themselves. Since the stochastic process representing the underlying has log normal returns, the two path will not be exactly symetric when we plot then together, as we can see on figure 1.

Since with the same number of simulations, we get twice as much price observations for the derivative, we need half of the initial number of simulations to get the same amount of observations. This is interesting because generating a (pseudo-)random number is a process that is time consuming in a computer program, and with this method we are generator half of the number generated normally, we then could expect the time of execution to be reduced.

In order to empirically test this, we will use a Python script (see the annexe for source code). We will compute the price of an european call

on a stock paying no dividend for several number of simulations, and will look very carefully to the difference of time of execution and the speed of convergence to the mean for a classical simulation and the antithetic variates method.

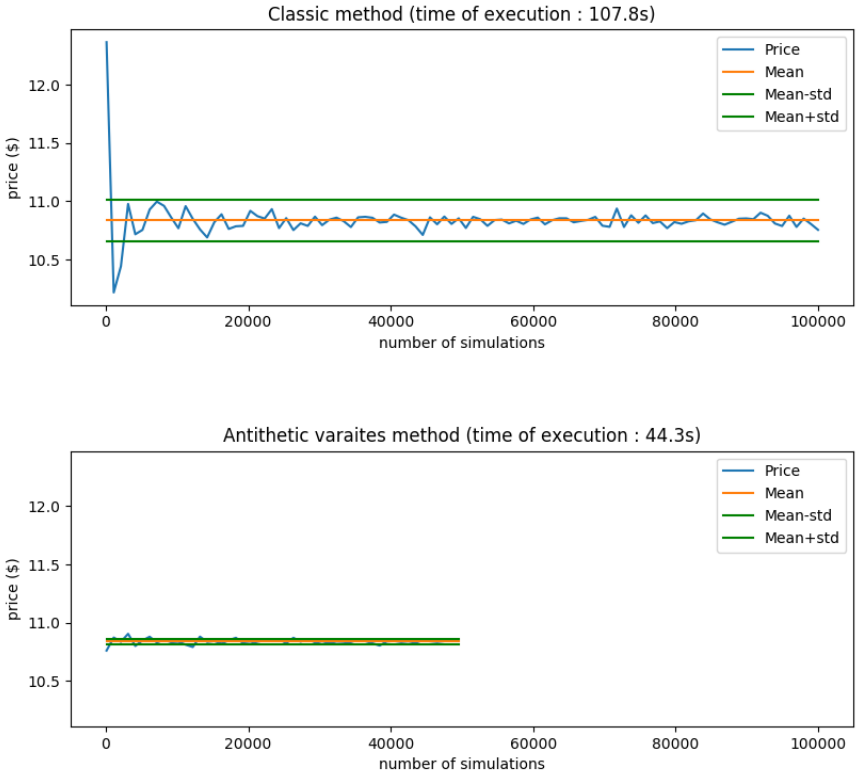


FIGURE 2 – Comparaison of the two methods

As we can see on figure 2, the antithetic variates method is much faster to converge to the mean than the classic method (x and y axis are the same for both charts). Moreover execution time is half the one of classical method, therefore it is both faster to converge and faster in execution time. Nonetheless, we have to keep in mind that we are here dealing with vanilla option with a payoof very easy (and fast) to compute. For more exotic options, especially path-dependant options, the amount of execution time spent of computing the payoff would be much more important and thus the net gain in execution time would not be that big, but it will not change anything to the relative speed of convergence to the mean value as long as the price is monotonic with

respect to the underlying price.

In order to explain why this method converge much faster, we have to remender how to reduce the confidence interval : reduce the variance.

Let's consider S_1 and S_2 two sample paths. We want an estimation of $\theta = \mathbb{E}(S)$, which is given by the unbiased estimator : $\hat{\theta} = \frac{\hat{\theta_1}+\hat{\theta_2}}{2}$. We then have :

$$Var(\hat{\theta}) = \frac{Var(S_1) + Var(S_2) + 2Cov(S_1,S_2)}{4}$$

With the above formula, it is clear why we are taking opposite values for the random path : the minimal variance is obtained when both path are negatively correlated ($Cov(S_1,S_2) = -1$).

3 Control variates

The key idea with the control variates method is that if a simulation misprice a derivative A with an error ϵ, it will also misprice another related derivative B with the same error ϵ. In order to use this method, we first compute the analytic price of the derivative A, then we compute with a simulation an approximation of the price of derivative A and we store the error. Finally we compute an approximation of the price of derivative B and then substract the error. We then have the following equality :

$$P_B = \hat{P}_B - \hat{P}_A + P_A$$

with P_B the price we want to compute, $\hat{P}_B$ the price of derivative B obtained with a simulation, $\hat{P}_A$ the price of derivative A obtained with a simulation and P_A the analytic price of derivative A.

A more general method will use a multiplicating factor for the error :

$$P_B = \hat{P}_B - k(\hat{P}_A - P_A), \quad k \in \mathbb{R}$$

Again, the goal is to reduce the variance. Here the variance of P_B is given by :

$$Var(P_B) = Var(\hat{P}_B) + k^2 Var(\hat{P}_A) + 2kCov(\hat{P}_B,\hat{P}_A)$$

One can proove (by computing the partial derivative of the above equation with respect to k, and setting it equal to 0) that the optimal value for k is :

$$k_{opt} = -\frac{Cov(\hat{P}_B,\hat{P}_A)}{Var(\hat{P}_A)}$$

Unfortunately, gathering data in order to compute k_{opt} could be very time consuming and void any gain of speed with the method, therefore one should be very cautious about computing k_{opt}.

4 Importance sampling

A Antithetic variates

```
import matplotlib.pyplot as plt
import numpy as np
from time import time
import pandas as pd

# Underlying informations
So = 100
mu = 0.1
sigma = 0.1

# European option informations
T = 1.0
K = 100.0
r = 0.05

# Simulation parameters
nbr_steps = 100
dt = T/nbr_steps
t = np.linspace(0, T, nbr_steps)
min_nbr_sim, max_nbr_sim = 100, 100000
nbr_steps_sims = 100
nbr_sims = np.linspace(min_nbr_sim, max_nbr_sim, nbr_steps_sims)

# Global variables for results storage
prices_standard = []
prices_antithetic = []

# Classic Monte-Carlo simulation
time_begin_classic = time()

for i,nbr_sim in enumerate(nbr_sims):
    print(i)
    nbr_sim = int(nbr_sim)
    price = 0.0
    for _ in range(nbr_sim):
        W = np.random.standard_normal(size = nbr_steps)
        W = np.cumsum(W)*np.sqrt(dt)
        X = (mu-0.5*sigma**2)*t + sigma*W
        S = So*np.exp(X)

        # Payoff computation of a european call
        if(S[-1]>K):
            price += S[-1]-K
    prices_standard.append((price/nbr_sim)*np.exp(-r*T))

calculation_time_classic = round(time()-time_begin_classic, 1)

# Antithetic variates method
time_begin_antithetic = time()
half_len = int(len(nbr_sims)/2)

for i,nbr_sim in enumerate(nbr_sims[0:half_len]):
    print(i)
    price = 0.0
```

```
    nbr_sim = int(nbr_sim)
    for _ in range(nbr_sim):
        W = np.random.standard_normal(size = nbr_steps)
        W_2 = -W
        W = np.cumsum(W)*np.sqrt(dt)
        W_2 = np.cumsum(W_2)*np.sqrt(dt)
        X = (mu-0.5*sigma**2)*t + sigma*W
        X_2 = (mu-0.5*sigma**2)*t + sigma*W_2
        S = So*np.exp(X)
        S_2 = So*np.exp(X_2)

        # Computation of both payoffs and then the mean
        if(S[-1]>K):
            price += S[-1]-K
        if(S_2[-1]>K):
            price += S_2[-1]-K
    prices_antithetic.append((price/(2*nbr_sim))*np.exp(-r*T))

calculation_time_antithetic = round(time()-time_begin_antithetic, 1)

# Computing mean and standard deviation
prices = np.array(prices_standard)
mean_val = np.mean(prices)
std_val = round(np.std(prices),4)

# Plotting classical Monte-Carlo simulation
plt.figure(1)
ax1 = plt.subplot(211)
ax1.set_title("Classic method (time of execution : {}s)".format(
    calculation_time_classic))
ax1.plot(nbr_sims, prices, label='Price')
ax1.plot(nbr_sims, np.linspace(mean_val, mean_val,100), label='Mean')
ax1.plot(nbr_sims, np.linspace(mean_val-std_val, mean_val-std_val,100),
    'g', label='Mean-std')
ax1.plot(nbr_sims, np.linspace(mean_val+std_val, mean_val+std_val,100),
    'g', label='Mean+std')
ax1.legend(loc="upper right")
ax1.set_xlabel('number of simulations')
ax1.set_ylabel('price ($)')

# Computing mean and standard deviation
prices = np.array(prices_antithetic)
mean_val = np.mean(prices)
std_val = round(np.std(prices),3)

# Plotting with the antithetic variates method
ax2 = plt.subplot(212, sharex=ax1, sharey=ax1)
ax2.set_title("Antithetic varaites method (time of execution : {}s)".
    format(calculation_time_antithetic))
ax2.plot(nbr_sims[0:half_len], prices, label='Price')
ax2.plot(nbr_sims[0:half_len], np.linspace(mean_val, mean_val,50),
    label='Mean')
ax2.plot(nbr_sims[0:half_len], np.linspace(mean_val-std_val, mean_val-
    std_val,50),'g', label='Mean-std')
ax2.plot(nbr_sims[0:half_len], np.linspace(mean_val+std_val, mean_val+
    std_val,50),'g', label='Mean+std')
ax2.legend(loc="upper right")
```

```
ax2.set_xlabel('number of simulations')
ax2.set_ylabel('price ($)')
plt.tight_layout()
plt.show()
```

antithetic_variates_method.py