Lesson 4 - Montecarlo Methods for Option Pricing

March 21, 2021

1 Variance Reduction Techniques

[9]:
```
%matplotlib inline

import math
import pandas               as pd
import numpy                as np
import matplotlib.pyplot    as plt

from gdl_finance.analytic    import BlackScholes
```

1.1 Variance Reduction, Efficiency Improvement and Error Estimation

In this lesson we most follow the Paul Glasserman's book, "Monte Carlo Methods in Financial Engineering" (Springer, 2004). Chapter 4 ("Variance Reduction Techniques") is used for basic definitions, formulations.

1.1.1 Error Estimation

Reducing simulation error is often at odds with convenient estimation of the simulation error itself; in order to supplement a reduced-variance estimator with a valid confidence interval, we sometimes need to sacrifice some of the potential variance reduction.

1.1.2 Efficiency Issues

First of all remember that the greatest gains from variance reduction techniques result from exploiting specific features of a problem rather than from generic applications of generic methods. This means that in general not only some techniques can work better than others but that the same technique applied in different conditions (e.g. different values of the simulation parameters) can sometimes produce different results.

Suppose we want to estimate a generic derivative price α.

$$\hat{\alpha}_n = \frac{1}{n}\sum_{i=1}^{n} f(x_i) \tag{1}$$

Equation (1) provides the estimator of α based on n Monte Carlo replications. The standar deviation of the estimate is given by

$$s_f = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}\left(f(x_i) - \hat{\alpha}_n\right)^2} \tag{2}$$

The standard error of the estimator is $s_f/\sqrt{n}$. This implies that decreasing the standard deviation by a factor of 10 while leaving everything else unchanged is equivalent to increasing the number of simulations by a factor of 100 in terms of error reduction.

Suppose now that we have at our disposal two unbiased Monte Carlo estimators $\hat{\alpha}^1$ and $\hat{\alpha}^2$. Since both are unbiased, we have

$$E[\hat{\alpha}^1] = E[\hat{\alpha}^2] = \alpha$$

Let's assume that the standard deviations are σ_1 and σ_2 respectively but with $\sigma_1 < \sigma_2$. From the previous observation, given a number n of replications, $\hat{\alpha}^1$ will present a lower error than $\hat{\alpha}^2$. However this analysis oversimplifies the comparison because it does not take into account eventual differences in the computational time required by the estimators. Smaller variance is not a sufficient ground for preferring one estimator over the other if its calculation is more time-consuming. A more exhaustive procedure is required to compare estimators with different computational requirements as well as different variances.

Suppose that $\tau_i, i = 1, 2$ is the time required to generate one replication of $\hat{\alpha}^1$ and $\hat{\alpha}^2$ respectively. With a given time budget τ the number of replications of $\hat{\alpha}^i$ is t/τ_i. The two estimators with computing time t are:

$$\frac{\tau_1}{t}\sum_{i=1}^{t/\tau_1}\hat{\alpha}_i^1 \qquad \text{and} \qquad \frac{\tau_2}{t}\sum_{i=1}^{t/\tau_2}\hat{\alpha}_i^2$$

and for large enough t they are normally distributed with mean α and standard deviations

$$\sigma_1\sqrt{\frac{\tau_1}{t}} \qquad \text{and} \qquad \sigma_2\sqrt{\frac{\tau_2}{t}}$$

Boyle et al. (1997) provided the condition that should be satisfied in order to prefer the first estimator over the second:

$$\sigma_1^2\tau_1 < \sigma_2^2\tau_2$$

The inverse of the product of variance times the time necessary to perform a single run is indicated in the literature with the name of *efficiency* (Hammersley and Handscomb, 1964). Using efficiency as a basis for comparing different estimators, we can conclude that the low variance estimator is preferable to the other only if the ratio of variances is smaller than the ratio between the times of single replication.

1.2 Antithetic Variates

1.2.1 Description

The method of antithetic variates is one of the simplest and widely used techniques in financial pricing problems. It attempts to reduce the estimator variance by introducing negative dependence between pairs of replications.

The core observation is that if a random variable U is uniformly distributed over $[0,1]$ then $1-U$ is too. Thus if we generate paths using as inputs $U_1,\ldots,U_n$ it is possible to generate additional paths using $1-U_1,\ldots,1-U_n$ without changing the law of the simulated process. The combinations of variables $(U_i, 1-U_i)$ constitute antithetic pairs in the sense that generally a large value of one is followed by a small value of the other.

These observations extend to other distributions through the inverse transform method. Specifically, in a simulation of stochastic processes based on independent standard normal random variables, antithetic pairs may be constructed by pairing a sequence $Z_1,\ldots,Z_n$ of i.i.d. $N(0,1$ variables with the sequence $-Z_1,\ldots,-Z_n$ of i.i.d.$N(0,1$ variables. If the $Z_i s$ are used to simulate the increments of a Brownian path the $-Z_i s$ simulate the increments of the reflection of the path about the origin.

To analyze the variance reduction produced by the method imagine we have to estimate an expectation $E[Y]$ and that using antithetic sampling we produce a series of pairs of observations $(Y_1,\tilde{Y}_1),\ldots,(Y_n,\tilde{Y}_n)$. The procedure presents the following characteristics:

- the pairs $(Y_1,\tilde{Y}_1),\ldots,(Y_n,\tilde{Y}_n)$ are i.i.d.;
- For each i, Y_i and $\tilde{Y}_i$ have the same distribution, though they are not independent.

The antithetic variates estimator is defined as

$$\hat{Y}_{AV} = \frac{1}{2n}\Big(\sum_{i=1}^{n} Y_i + \sum_{i=1}^{n} \tilde{Y}_i\Big) = \frac{1}{n}\sum_{i=1}^{n}\Big(\frac{Y_i+\tilde{Y}_i}{2}\Big)$$

The central limit theorem implies that

$$\frac{\hat{Y}_{AV} - E[Y]}{\sigma_{AV}/\sqrt{n}} \Rightarrow N(0,1)$$

with

$$\sigma^2_{AV} = var\Big[\frac{Y_i+\tilde{Y}_i}{2}\Big]$$

As for the case of the Monte Carlo estimator the limit distribution continues to hold if we replace the population standard deviation σ_{AV} with the sample standard deviation of the n values. This justify the construction of a $1-\delta$ confidence interval of the form

$$\hat{Y}_{AV} \pm z_{\delta/2}\frac{s_{AV}}{\sqrt{n}}$$

Before comparing the antithetic variates estimator with the Monte Carlo one we make the assumption that the computational time required to simulate a pair $(Y_i, \tilde{Y}_i)$ is approximately twice the effort required to simulate a single observation Y_i. Following this assumption it is reasonable to compare the variance of $\hat{Y}_{AV}$ with the variance of a Monte Carlo estimator based on $2n$ independent replications. In particular antithetic variates reduce variance if:

$$var[\hat{Y}_{AV}] < var\Big[\frac{1}{2n}\sum_{i=1}^{2n} Y_i\Big]$$

or

$$var[Y_i + \tilde{Y}_i] < 2var[Y_i] \quad \Rightarrow \quad 2var[Y_i] + 2cov[Y_i, \tilde{Y}_i] < 2var[Y_i] \quad \Rightarrow \quad cov[Y_i, \tilde{Y}_i] < 0$$

The condition requires that the negative dependence of the input random variables produces negative covariance between the estimates of two paired replications. A sufficient condition ensuring this is monotonicity of the mapping function from inputs to outputs defined in the algorithm. Therefore if $Y = f(Z_1, \ldots, Z_n)$ for some increasing function f then $\tilde{Y} = f(-Z_1, \ldots, -Z_n)$ is a decreasing function of $(Z_1, \ldots, Z_n)$. In this case we have

$$E[f(Z_i)f(-Z_i)] \leq E[f(Z_i)]E[f(-Z_i)] \Rightarrow cov[Y_i, \tilde{Y}_i] = E[f(Z_i)f(-Z_i)] - E[f(Z_i)]E[f(-Z_i)] \leq 0$$

This argument can be adapted to show that the method of antithetic variates increases efficiency in pricing options that depend monotonically on inputs (e.g. European, American or Asian options). Note that **in the case of non-monotome payoffs, the method of antithetical variables does not necessarily provide better performance than the standard Monte Carlo, indeed in some conditions the results can be significantly worse**.

1.2.2 A Practical Example: Estimation of a Simple Integral

Antithetic variates

We would like to estimate

$$I = \int_0^1 \frac{1}{1+x}\,\mathrm{d}x$$

using Monte Carlo integration. This integral is the expected value of $f(U)$, where

$$f(U) = \frac{1}{1+U}$$

and U follows a uniform distribution $[0, 1]$. Using a sample of size n denote the points in the sample as $u_1, \cdots, u_n$. Then the estimate is given by

$$I \approx \frac{1}{n}\sum_i f(u_i)$$

[10]:
```
n = 1000

u = np.random.uniform(0,1,n)

f = 1.0 /(1.0 + u)

print('\nNormal Simulation : \n')
print('True value          = {}'.format(round(np.log(2),6)))
print('Integral estimation = {}'.format(round(np.average(f),6)))
print('Estimation variance = {}'.format(round(np.std(f)**2,6)))

u1 = np.random.uniform(0,1,n/2)
u2 = 1 - u1

f1 = 1.0 / (1.0 + u1)
f2 = 1.0 / (1.0 + u2)
f  = 0.5 * (f1 + f2)

print('\nAntithetic Variate : \n')
print('Integral estimation = {}'.format(round(np.average(f),6)))
print('Estimation variance = {}'.format(round(np.std(f)**2,6)))
```

```
Normal Simulation :

True value          = 0.693147
Integral estimation = 0.695925
Estimation variance = 0.019994

Antithetic Variate :

Integral estimation = 0.692958
Estimation variance = 0.000562
```

1.2.3 Python Example

Let's see a very simple implementation of the antithetic methods in the case of european option pricing.

[11]:
```
# np.random.seed(1234)
#
# Model Parameters␣
 ↪-----------------------------------------------------------------
#
S0     =  70.0   # initial stock level
K      = 100.0   # strike price
T      =   1.0   # time-to-maturity
r      =   0.05  # short rate
```

```
delta  =   0.0   # dividend yield
sigma  =   0.20  # volatility
payout =   -1     # 1 stay for call and -1 stay for put
#
# Simulation Parameters␣
 ↪-------------------------------------------------------------
#
b = 10000       # number of paths
M = 10          # number of points for each path
dt = float(T) /float( M)
df = math.exp(-r * T)
#
# We're going to use the Black and Scholes price as a reference for the Monte␣
 ↪Carlo
# simulation
#
option_price = BlackScholes(payout, S0, K, r, delta, sigma, T)
```

In order to study the effect of the Antithetic Method, we generate two distinct samples of random numbers. The first one use a single generation using the usual randn function.

[12]:
```
#
# Random numbers generations. We use the function 'randn' which returns
# a sample (or samples) from the "standard normal" distribution.
# If positive, int_like or int-convertible arguments are provided,
# 'randn' generates an array of shape (d0, d1, ..., dn), filled
# with random floats sampled from a univariate "normal" (Gaussian)
# distribution of mean 0 and variance 1
#
z1  = np.random.randn(M, b)
#
# Stock Price Paths. The function 'cumsum' returns the cumulative sum
# of the elements along a given axis, in this case we want to sum along
# the time axis (axis=0)
#
S1 = S0 * np.exp(np.cumsum((r - 0.5 * sigma ** 2) * dt
    + sigma * math.sqrt(dt) * z1, axis=0))
S1 = np.insert(S1, 0, S0, axis=0)

print('The shape of z1 is : {}'.format(z1.shape))
```

```
The shape of z1 is : (10, 10000)
```

To apply the antithetic variate technique, we generate standard normal random numbers Z and define two set of samples of the undelying price

$$S_T^+ = S_0 e^{(r-\sigma^2/2)T+\sigma\sqrt{T}Z} \quad S_T^- = S_0 e^{(r-\sigma^2/2)T+\sigma\sqrt{T}(-Z)}$$

Similarly we define two sets of discounted payoff samples ...

$$V_T^+ = \max[S^+(T) - K, 0] \quad V_T^- = \max[S^-(T) - K, 0]$$

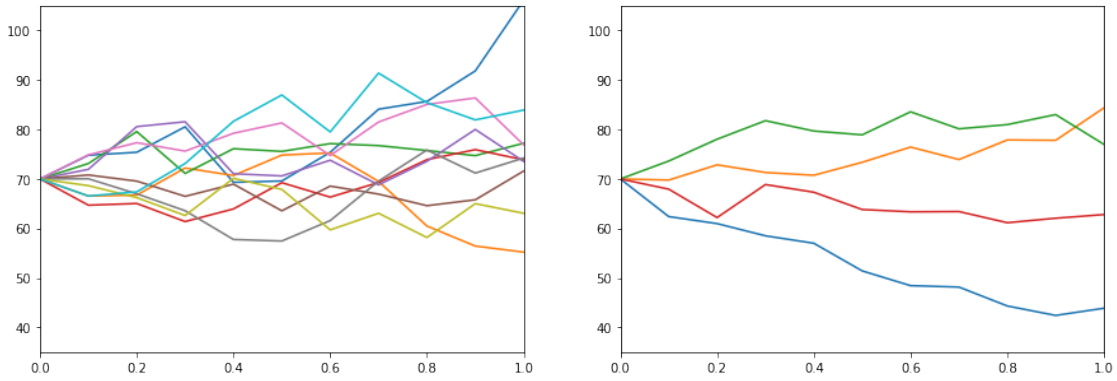
... and at last we construct our mean estimator by averaging these samples

$$\bar{V}_0 = \frac{1}{n} \sum_{j=1}^{n} \frac{1}{2} \left(V_j^+ + V_j^- \right)$$

Here we generate the two distinct set of paths. The first one uses b/2 normally distributed random numbers generated in the usual way while the second set simulate the increments of the reflection of the path about the origin (antithetic sample).

[13]:
```
z2  = np.random.randn(M, int(b/2))
#
S21 = S0 * np.exp(np.cumsum((r - 0.5 * sigma ** 2) * dt
       + sigma * math.sqrt(dt) * z2, axis=0))
S22 = S0 * np.exp(np.cumsum((r - 0.5 * sigma ** 2) * dt
       + sigma * math.sqrt(dt) * (-z2), axis=0))
S21 = np.insert(S21, 0, S0, axis=0)
S22 = np.insert(S22, 0, S0, axis=0)
```

[14]:
```
t = np.linspace(0, T, M+1)

# plotting the first n paths

paths_1 = S1[:,0:10]
fig = plt.figure(figsize=(15,5))
axis = fig.add_subplot(121, autoscale_on=False, xlim=(0,T), ylim=(.5 * S0,1.
 ↪5*S0))
plt.plot(t, paths_1)

S2 = np.append(S21, S22, axis=1)

paths_1 = S2[:,0:4]
axis = fig.add_subplot(122, autoscale_on=False, xlim=(0,T), ylim=(.5 * S0,1.
 ↪5*S0))
plt.plot(t, paths_1)

plt.show()
```



[15]:
```
#
# first estimation without antithetic variables
#
option_price_est_1 = np.maximum(payout*(S1[-1] - K), 0)
std_dev_1          = np.std(option_price_est_1)
# average on montecarlo simulations
option_price_est_1 = np.average(option_price_est_1)  * df
#
# ... second estimation using the antithetic path set
#
option_price_est_21 = np.maximum(payout*(S21[-1] - K), 0)
option_price_est_22 = np.maximum(payout*(S22[-1] - K), 0)
# we take the average of the previous estimator
option_price_est_2 = 0.5 * (option_price_est_21 + option_price_est_22)
std_dev_2          = np.std(option_price_est_2)
# average on montecarlo simulations
option_price_est_2 = np.average(option_price_est_2) * df
```

[17]:
```
print('BS price             = ' + str("%.3f" % option_price))
print('Antithetic MC        = ' + str("%.3f" % option_price_est_2) + ' +/- ' +␣
 ↪str("%.3f" % (1.96 * std_dev_2/math.sqrt(b))))
print('Simple     MC        = ' + str("%.3f" % option_price_est_1) + ' +/- ' +␣
 ↪str("%.3f" % (1.96 * std_dev_1/math.sqrt(b))))
```

```
BS price             = 25.564
Antithetic MC        = 25.524 +/- 0.022
Simple     MC        = 25.426 +/- 0.271
```

1.3 Analysis of Variance Reduction

1.3.1 Whit respect to K and σ

[19]:
```
from pandas import DataFrame

Strikes = [70, 80, 90, 100, 110, 120]
Sigmas  = [0.01, 0.05, 0.1, 0.2, 0.25, 0.30]

col1 = []
col2 = []
col3 = []
col4 = []
col5 = []
col6 = []
col7 = []
for K in Strikes:
    for sigma in Sigmas:
        option_price = BlackScholes('C', S0, K, r, delta, sigma, T)
        #
        # Standard path generation
        #
        z1  = np.random.randn(M, b)
        #
        S1 = S0 * np.exp(np.cumsum((r - 0.5 * sigma ** 2) * dt
            + sigma * math.sqrt(dt) * z1, axis=0))
        S1 = np.insert(S1, 0, S0, axis=0)
        #
        # Antithetic path generation
        #
        z2  = np.random.randn(M, int(b/2))
        #
        S21 = S0 * np.exp(np.cumsum((r - 0.5 * sigma ** 2) * dt
               + sigma * math.sqrt(dt) * z2, axis=0))
        S22 = S0 * np.exp(np.cumsum((r - 0.5 * sigma ** 2) * dt
               + sigma * math.sqrt(dt) * (-z2), axis=0))
        S21 = np.insert(S21, 0, S0, axis=0)
        S22 = np.insert(S22, 0, S0, axis=0)
        #
        # option price estimation without variance reduction
        #
        option_price_est_1 = np.maximum(S1[-1] - K, 0) * df
        std_dev_1          = np.std(option_price_est_1)
        option_price_est_1 = np.average(option_price_est_1)
        #
        # option price estimation with antithetic variables
        #
        option_price_est_21 = np.maximum(S21[-1] - K, 0) * df
```

```
        option_price_est_22 = np.maximum(S22[-1] - K, 0) * df
        option_price_est_2 = 0.5 * (option_price_est_21 + option_price_est_22)
        std_dev_2          = np.std(option_price_est_2)
        option_price_est_2 = np.average(option_price_est_2)
        #
        # building pandas dataframe
        #
        col1.append(K)
        col2.append(sigma)
        col3.append('{:10.6f}'.format(option_price))
        col4.append('{:10.6f}'.format(option_price_est_1))
        col5.append('{:10.6f}'.format(option_price_est_2))
        col6.append(1.96 * std_dev_1/math.sqrt(b))
        col7.append(1.96 * std_dev_2/math.sqrt(b))

data = {'strike'    : col1,
        'sigma'     : col2,
        'BS price'  : col3,
        'MC simple' : col4,
        'SE simple' : col6,
        'MC AV'     : col5,
        'SE AV'     : col7}

frame = DataFrame(data, columns=['strike','sigma','BS price','MC simple','SE␣
 ↪simple','MC AV','SE AV'])
frame.head()
```

[19]:

	strike	sigma	BS price	MC simple	SE simple	MC AV	SE AV
0	70	0.01	0.000000	3.406356	0.013676	3.413966	0.000097
1	70	0.05	0.284348	3.681406	0.059992	3.701598	0.013738
2	70	0.10	1.349530	4.772160	0.106592	4.799519	0.041533
3	70	0.20	3.901468	7.184025	0.198775	7.262755	0.101276
4	70	0.25	5.221259	8.518402	0.252734	8.753773	0.134306

1.3.2 With Respect to the Number of Simulations

[18]:
```
import sys

def antithetic_simulation(pars, start, finish, print_output=True):
    #
    # Simulation Parameters␣
 ↪-------------------------------------------------------------
    #
    S0     = pars['S0']
    K      = pars['K']
    r      = pars['r']
    delta  = pars['delta']
```

```
    sigma  = pars['sigma']
    T      = pars['T']
    payout = pars['payout']
    M      = 3      # number of points for each path
    dt     = float(T) /float( M)
    df     = math.exp(-r * T)
    #
    #␣
↪-------------------------------------------------------------------------------
    #
    x   = []
    y1  = []
    y2  = []
    option_price = BlackScholes(payout, S0, K, r, delta, sigma, T)
    for b in range(start, finish, 2):
        #
        z1  = np.random.randn(M, b)
        #
        S1 = S0 * np.exp(np.cumsum((r - 0.5 * sigma ** 2) * dt
            + sigma * math.sqrt(dt) * z1, axis=0))
        S1 = np.insert(S1, 0, S0, axis=0)

        option_price_est_1 = np.maximum(payout*(S1[-1] - K), 0) * df
        std_dev_1          = np.std(option_price_est_1)
        option_price_est_1 = np.average(option_price_est_1)
        #
       ␣
↪#-------------------------------------------------------------------------------
        #
        z2  = np.random.randn(M, int(b/2))
        #
        S21 = S0 * np.exp(np.cumsum((r - 0.5 * sigma ** 2) * dt
               + sigma * math.sqrt(dt) * z2, axis=0))
        S22 = S0 * np.exp(np.cumsum((r - 0.5 * sigma ** 2) * dt
               + sigma * math.sqrt(dt) * (-z2), axis=0))
        S21 = np.insert(S21, 0, S0, axis=0)
        S22 = np.insert(S22, 0, S0, axis=0)

        option_price_est_21 = np.maximum(payout*(S21[-1] - K), 0) * df
        option_price_est_22 = np.maximum(payout*(S22[-1] - K), 0) * df

        option_price_est_2 = 0.5 * (option_price_est_21 + option_price_est_22)
        std_dev_2          = np.std(option_price_est_2)
        option_price_est_2 = np.average(option_price_est_2)
        #
       ␣
↪#-------------------------------------------------------------------------------
```

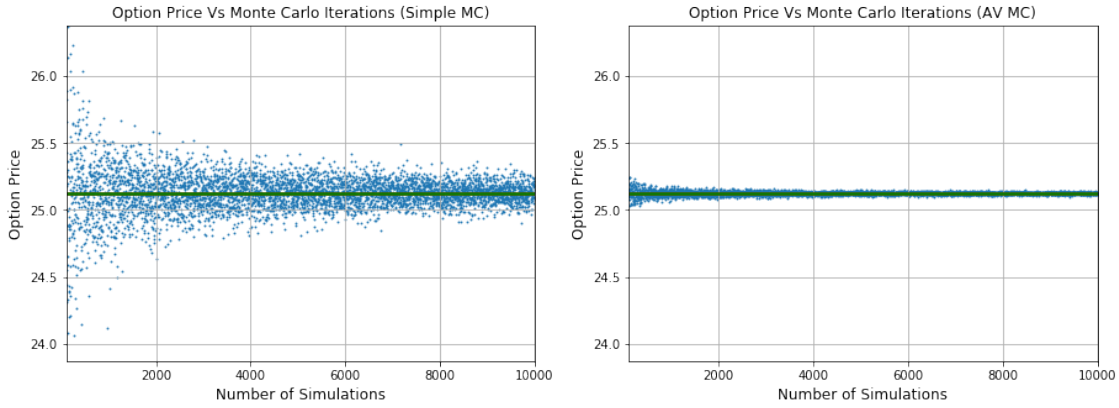
```
        #
        x.append(b)
        y1.append(option_price_est_1)
        y2.append(option_price_est_2)

        sys.stdout.write("\r" + 'Simulation number    = ' + str(b))
        sys.stdout.flush()

    return x, y1, y2, option_price
```

[21]:
```
#
# Model Parameters␣
 ↪--------------------------------------------------------------------
#
param = {'S0':70.0,      # initial stock level
         'K':100.0,      # strike price
         'T':1.0,        # time-to-maturity
         'r':0.05,       # short rate
         'delta':0.0,    # dividend yield
         'sigma':0.10,   # volatility
         'payout':-1,    # 1 stay for call and -1 stay for put
}

x, y1, y2, bs_price = antithetic_simulation(param,100,10002)
print('\nDone!')
```

```
Simulation number    = 10000
Done!
```

[22]:
```
if len(x) > 1:

    y_inf = 0.95
    y_sup = 2.0 - y_inf

    plt.figure(figsize=(15,5))

    plt.subplot(121)
    plt.xlim((min(x), max(x)))
    plt.ylim((y_inf * bs_price, y_sup * bs_price))
    plt.title('Option Price Vs Monte Carlo Iterations (Simple MC)', fontsize=12)
    plt.xlabel('Number of Simulations',                             fontsize=12)
    plt.ylabel('Option Price',                                      fontsize=12)
    plt.grid(True)
    plt.scatter(x,y1,marker='.',s=3)
    plt.hlines(np.average(y1), min(x), max(x), colors='red',        linewidth=2.
 ↪5)
    plt.hlines(bs_price,   min(x), max(x), colors='green',      linewidth=2.5)
```

```
    plt.subplot(122)
    plt.xlim((min(x), max(x)))
    plt.ylim((y_inf * bs_price, y_sup * bs_price))
    plt.title('Option Price Vs Monte Carlo Iterations (AV MC)',     fontsize=12)
    plt.xlabel('Number of Simulations',                             fontsize=12)
    plt.ylabel('Option Price',                                      fontsize=12)
    plt.grid(True)
    plt.scatter(x,y2,marker='.',s=3)
    plt.hlines(np.average(y2), min(x), max(x), colors='red',        linewidth=2.
↪5)
    plt.hlines(bs_price,   min(x), max(x), colors='green',     linewidth=2.5)

    plt.show()
```



As mentioned at the beginning, we must not assume too much from this result. The method does not always produce such striking results. However, since the implementation is very simple and that it cannot produce worse results than a standard simulation (**in the case of monotonous payoffs**), in practice it is widely used.

1.4 Control Variate

The method of control variate is among the most effective of the variance reduction techniques. It generally exploits information about the errors in estimates of known quantities to reduce the error in an estimate of an unknown quantity.

Suppose again we want to estimate the derivative price α. The Monte Carlo estimator from n independent and identically distributed replications $\alpha_1, \ldots, \alpha_n$ is $\hat{\alpha} = (\alpha_1 + \cdots + \alpha_n)/n$.

Imagine now, that for each replication it is possible to calculate another output X_i along with α_i. The pairs $(X_i, \alpha_i), i = 1, \ldots, n$ are i.i.d. and suppose that the expectation $E[X]$ is known. Thus for any fixed b it is possible to calculate

$$\alpha_i(b) = \alpha_i - b\,(X_i - E[X])$$

for each replication i. The control variate Monte Carlo estimator would then be

$$\hat{\alpha}(b) = \hat{\alpha} - b\left(\hat{X} - E[X]\right) = \frac{1}{n}\sum_{i=1}^{n}\left(\alpha_i - b\left(X_i - E[X]\right)\right)$$

and the observed error $\hat{X} - E[X]$ is used to control the estimate $E[\alpha]$. We can demonstrate that the control variate estimator is unbiased and consistent.

The variance of each replication $\alpha_i(b)$ is

$$var[\alpha_i(b)] = var[\alpha_i - b(X_i - E[X])] = \sigma_\alpha^2 - 2b\sigma_\alpha\sigma_X\rho + b^2\sigma_X^2 = \sigma^2(b) \tag{3}$$

where $\sigma_X^2 = var[X], \sigma_\alpha^2 = var[\alpha_i]$ and ρ is the correlation between X and α. Comparing the control variate estimator variance $\sigma^2(b)/n$ and the ordinary Monte Carlo estimator variance σ^2/n it is possible to infer that the control variate estimator $\hat{\alpha}(b)$ has smaller variance than the standard estimator $\hat{\alpha}$ if $b^2\sigma_X < 2b\sigma_\alpha\rho$.

The optimal coefficient that minimizes the variance in (3) is

$$b^\star = \frac{\sigma_\alpha}{\sigma_X}\rho = \frac{cov[X,\alpha]}{var[X]} \tag{4}$$

Plugging this value into (3) and arranging it is possible to find the ratio of the variance of the optimally controlled estimator to the one of the uncontrolled estimator

$$\frac{var[\hat{\alpha} - b^\star(\hat{X} - E[X])]}{var[\hat{\alpha}]} = 1 - \rho^2 \tag{5}$$

From equation (5) it is possible to observe some important features of the method:

- With the optimal coefficient $b^\star$, the effectiveness of a control variate is determined by the size of the correlation between α and X. Moreover the sign of the correlation is irrelevant.
- If the computational effort required per replication is roughly the same with and without the control variate then equation (2.6) measures the computational improvement from the use of a control variate. Specifically $n/(1-\rho^2)$ replications of α_i are necessary to achieve the same variance as n replications of the control variate estimator.
- The variance reduction factor $1/(1-\rho^2)$ increases very sharply as $|\rho|$ approaches 1 and it drops off quickly as $|\rho|$ decreases away from 1.

In practice since $E[\alpha]$ is unknown it is likely that also σ_α and ρ are unknown thus creating a problem in the calculation of the optimal coefficient $b^\star$. However it is still possible to get the most of the benefit of a control variate by using an estimate of $b^\star$. Replacing the population parameters in (4) with their sample counterparts yields the estimate

$$\hat{b}_n = \frac{\sum_{i=1}^{n}(X_i - \hat{X})(\alpha_i - \hat{\alpha})}{\sum_{i=1}^{n}(X_i - \hat{X})^2} \tag{6}$$

The expression in (6) is the slope of the least-squares regression line through the points $(X_i, \alpha_i), i = 1, \ldots, n$ which can be calculated easily.

1.4.1 A Practical Example: Estimation of a Simple Integral

Control variates

We would like to estimate

$$I = \int_0^1 \frac{1}{1+x}\,\mathrm{d}x$$

using Monte Carlo integration. This integral is the expected value of $f(U)$, where

$$f(U) = \frac{1}{1+U}$$

and U follows a uniform distribution $[0, 1]$. Using a sample of size n denote the points in the sample as $u_1, \cdots, u_n$. Then the estimate is given by

$$I \approx \frac{1}{n}\sum_i f(u_i)$$

[21]:
```
n = 1000

u = np.random.uniform(0,1,n)

f = 1.0 /(1.0 + u)

print('True value          = {}'.format(round(np.log(2),6)))
print('Integral estimation = {}'.format(round(np.average(f),6)))
print('Estimation variance = {}'.format(round(np.std(f)**2,6)))
```

```
True value          = 0.693147
Integral estimation = 0.69137
Estimation variance = 0.01948
```

Now we introduce $g(U) = 1 + U$ as a control variate with a known expected value

$$\mathbb{E}\left[g\left(U\right)\right] = \int_0^1 (1+x)\,\mathrm{d}x = \tfrac{3}{2}$$

[22]:
```
g          = 1 + u
g_expected = 3.0/2.0

covariance      = np.cov(g, f)
variance        = np.var(g)
beta            = covariance[0][1]/variance
```

```
print(beta)
```

```
-0.47723776858122086
```

and combine the two into a new estimate

$$I \approx \frac{1}{n}\sum_i f(u_i) + c\left(\frac{1}{n}\sum_i g(u_i) - 3/2\right)$$

[24]:
```
I = f - beta*(g - g_expected)

print('True value           = {}'.format(round(np.log(2),6)))
print('Integral estimation  = {}'.format(round(np.average(I),6)))
print('Estimation Variance  = {}'.format(round(np.std(I)**2,6)))
```

```
True value           = 0.693147
Integral estimation  = 0.693068
Estimation Variance  = 0.000641
```

1.4.2 A Classical Example: MC Pricing of an Asian Option

Asian options are options in which the underlying variable is the **average price** over a period of time. Because of this fact, Asian options have a lower volatility and hence rendering them cheaper relative to their European counterparts. They are commonly traded on currencies and commodity products which have low trading volumes. They were originally used in 1987 when Banker's Trust Tokyo office used them for pricing average options on crude oil contracts; and hence the name "Asian" option.

There are numerous permutations of Asian option; the most basic are listed below:

- Fixed strike (also known as an average rate) Asian call payout

$$C(T) = \max\left(A(0,T) - K, 0\right)$$

where A denotes the average price for the period $[0, T]$, and K is the strike price. The equivalent put option is given by

$$P(T) = \max\left(K - A(0,T), 0\right)$$

- The floating strike (or floating rate) Asian call option has the payout

$$C(T) = \max\left(S(T) - kA(0,T), 0\right)$$

where $S(T)$ is the price at maturity and k is a weighting, usually 1 so often omitted from descriptions. The equivalent put option payoff is given by

$$P(T) = \max\left(kA(0,T) - S(T), 0\right)$$

The Average A may be obtained in many ways. Conventionally, this means an arithmetic average. In the continuous case, this is obtained by

$$A(0,T) = \frac{1}{T}\int_0^T S(t)dt$$

For the case of discrete monitoring (with monitoring at the times $0 = t_0, t_1, t_2, \ldots, t_n = T$ and $t_i = i \cdot \frac{T}{n}$

we have the average given by

$$A(0,T) = \frac{1}{n}\sum_{i=1}^{n} S(t_i)$$

There also exist Asian options with geometric average; in the continuous case, this is given by

$$A(0,T) = \exp\left(\frac{1}{T}\int_0^T \ln(S(t))dt\right)$$

It's remarkable that it's possible to find a closed-form solution for the geometric Asian option; when used in conjunction with control variates in Monte Carlo simulations, the formula is useful for deriving fair values for the arithmetic Asian option. For the call we have

$$C_G = S_0 e^{(b-r)T}\Phi(d_1) - Ke^{-rT}\Phi(d_2)$$

and for the put

$$P_G = Ke^{-rT}\Phi(-d_2) - S_0 e^{(b-r)T}\Phi(-d_1)$$

where

$$b = \frac{1}{2}\left(r - \frac{1}{2}\sigma_G^2\right),\ \sigma_G = \frac{\sigma}{\sqrt{3}}$$
$$d_1 = \frac{\log\frac{S_0}{K} + \left(b + \frac{1}{2}\sigma_G^2\right)T}{\sigma_G\sqrt{T}},\ d_2 = d_1 - \sigma_G\sqrt{T}$$

[23]:
```
from ipywidgets import IntProgress
from IPython.display import display
```

[24]:
```
from scipy.stats.mstats     import gmean
from gdl_finance.analytic   import AsianGeometric
#
```

```
# Model Parameters␣
 ↪---------------------------------------------------------------
#
S0    = 100.0   # initial stock level
K     =  70.0   # strike price
T     =   1.0   # time-to-maturity
r     =   0.01  # short rate
delta =   0.0   # dividend yield
sigma =   0.4   # volatility
navg  =   24
```

[25]:
```
#
# Simulation Parameters␣
 ↪---------------------------------------------------------------
#
b_max = 10002   # number of paths
M     = 25      # number of points for each path
dt    = float(T) /float( M)
df    = math.exp(-r * T)
#
#␣
 ↪-------------------------------------------------------------------------------------
#
f = IntProgress(min=0, max=b_max) # instantiate the bar
display(f) # display the bar
#
x   = []
y1  = []
y2  = []
y3  = []
for b in range(2, b_max, 2):
    #
    z1  = np.random.randn(M, b)
    #
  ␣
 ↪#-------------------------------------------------------------------------------------
    #
    # Simple Montecarlo without variance reduction
    #
    S1 = S0 * np.exp(np.cumsum((r - 0.5 * sigma ** 2) * dt
        + sigma * math.sqrt(dt) * z1, axis=0))
    S1 = np.insert(S1, 0, S0, axis=0)
    #
    # here we compute the average of stock prices ...
    #
    avg  = np.average(S1[-navg:-1,:], axis=0)
    #
```

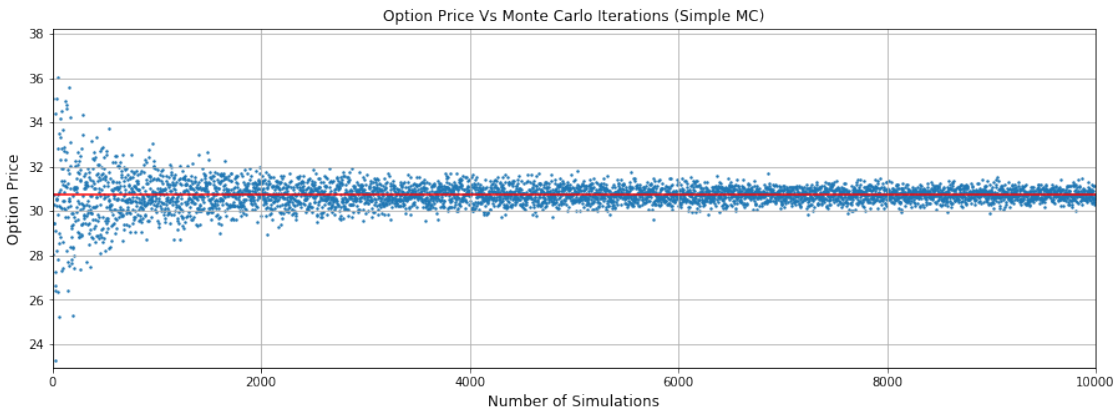
```
    # ... and here we have the geometric average
    #
    gavg = gmean     (S1[-navg:-1,:], axis=0)

    opt_asian_1      = np.maximum(avg   - K, 0) * df
    std_dev_1        = np.std(opt_asian_1)/math.sqrt(float(b))
    opt_asian_1      = np.average(opt_asian_1)
    #
  ␣
↪#-------------------------------------------------------------------------------
    #
    # Antithetic Variable
    #
    z2  = np.random.randn(M, int(b/2))
    #
    S21 = S0 * np.exp(np.cumsum((r - 0.5 * sigma ** 2) * dt
           + sigma * math.sqrt(dt) * z2, axis=0))
    S22 = S0 * np.exp(np.cumsum((r - 0.5 * sigma ** 2) * dt
           + sigma * math.sqrt(dt) * (-z2), axis=0))
    S21 = np.insert(S21, 0, S0, axis=0)
    S22 = np.insert(S22, 0, S0, axis=0)

    avg1  = np.average(S21[-navg:-1,:], axis=0)
    avg2  = np.average(S22[-navg:-1,:], axis=0)

    opt_asian_21 = np.maximum(avg1 - K, 0) * df
    opt_asian_22 = np.maximum(avg2 - K, 0) * df

    opt_asian_2 = 0.5 * (opt_asian_21 + opt_asian_22)
    std_dev_2   = np.std(opt_asian_2)/math.sqrt(float(b))
    opt_asian_2 = np.average(opt_asian_2)
    #
  ␣
↪#-------------------------------------------------------------------------------
    #
    # Control Variate
    #
    opt_estimate    = np.maximum(avg   - K, 0) * df
    control_variate = np.maximum(gavg  - K, 0) * df
    #
    # Here we estimate the correlation between the control variate and the␣
↪estimate
    #
    covariance      = np.cov(control_variate, opt_estimate)
    variance        = np.var(control_variate)
    beta            = covariance[0][1]/variance
    #
```

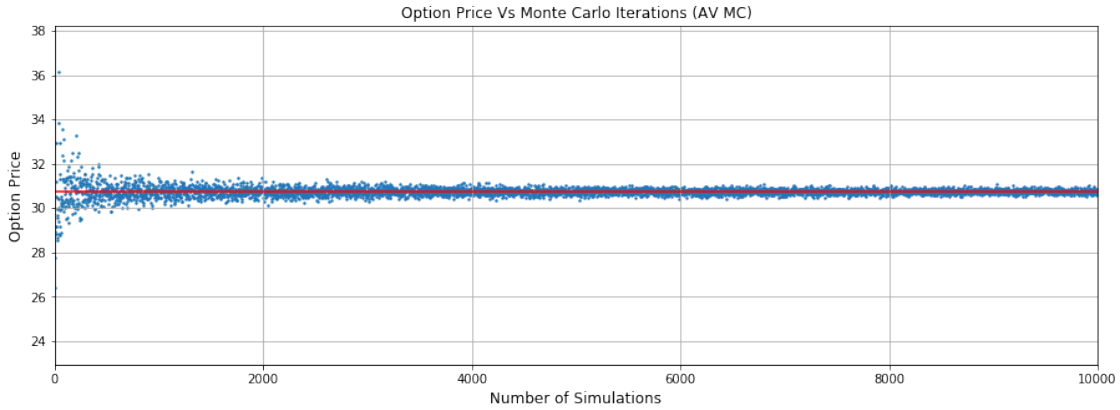
```
    # we compute the theoretical value we use in the adjustment
    #
    opt_asian_g_teo = AsianGeometric(1, S0, K, r, delta, sigma, T, navg)
    #
    # finally we calculate the control variate estimate
    #
    opt_asian_3     = opt_estimate - beta*(control_variate - opt_asian_g_teo)
    std_dev_3       = np.std(opt_asian_3)/math.sqrt(float(b))
    opt_asian_3     = np.average(opt_asian_3)

    x.append(b)
    y1.append(opt_asian_1)
    y2.append(opt_asian_2)
    y3.append(opt_asian_3)

    sys.stdout.write("\r" + 'Simulation number : ' + str(b))
    sys.stdout.flush()

    f.value += 2 # signal to increment the progress bar
#
#␣
 ↪-------------------------------------------------------------------------------
#
if len(x) > 1:

    fig_size = plt.rcParams["figure.figsize"]
    fig_size[0] = 15
    fig_size[1] = 5

    y_inf = 0.75
    y_sup = 2.0 - y_inf

    df1 = pd.DataFrame({'x':x, 'y':y1})
    ax = df1.plot('x', 'y', kind='scatter', s=2)
    ax.set_xlim((min(x), max(x)))
    ax.set_ylim((y_inf * opt_asian_1, y_sup * opt_asian_1))

    plt.hlines(np.average(y1), min(x), max(x), colors='red',        linewidth=1.
 ↪5)
    plt.title('Option Price Vs Monte Carlo Iterations (Simple MC)', fontsize=12)
    plt.xlabel('Number of Simulations',                             fontsize=12)
    plt.ylabel('Option Price',                                      fontsize=12)
    plt.grid(True)

    df2 = pd.DataFrame({'x':x, 'y':y2})
    ax = df2.plot('x', 'y', kind='scatter', s=2)
    ax.set_xlim((min(x), max(x)))
```

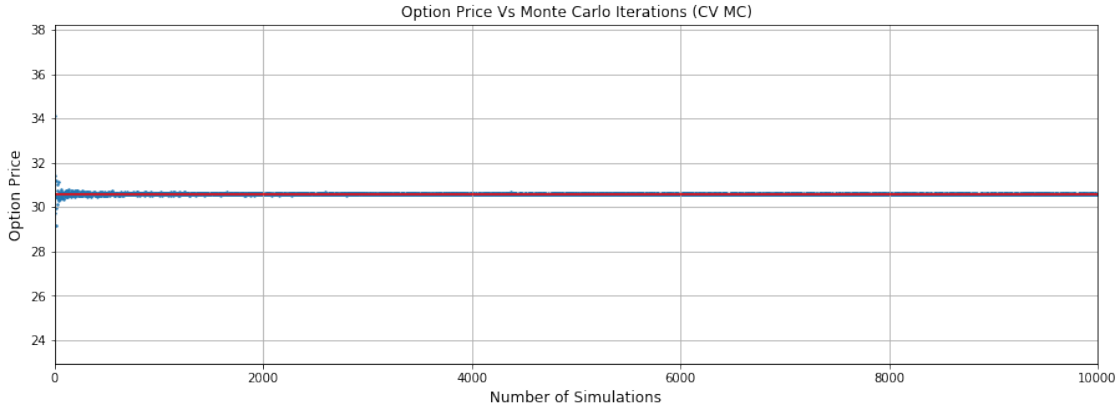
```
    ax.set_ylim((y_inf * opt_asian_1, y_sup * opt_asian_1))

    plt.hlines(np.average(y2), min(x), max(x), colors='red',        linewidth=1.
↪5)
    plt.title('Option Price Vs Monte Carlo Iterations (AV MC)',     fontsize=12)
    plt.xlabel('Number of Simulations',                             fontsize=12)
    plt.ylabel('Option Price',                                      fontsize=12)
    plt.grid(True)

    df3 = pd.DataFrame({'x':x, 'y':y3})
    ax = df3.plot('x', 'y', kind='scatter', s=2)
    ax.set_xlim((min(x), max(x)))
    ax.set_ylim((y_inf * opt_asian_1, y_sup * opt_asian_1))

    plt.hlines(np.average(y3), min(x), max(x), colors='red',        linewidth=1.
↪5)
    plt.title('Option Price Vs Monte Carlo Iterations (CV MC)',     fontsize=12)
    plt.xlabel('Number of Simulations',                             fontsize=12)
    plt.ylabel('Option Price',                                      fontsize=12)
    plt.grid(True)

    plt.show()
```

```
IntProgress(value=0, max=10002)
```

```
Simulation number : 10000
```







1.4.3 Analysis of Efficiency

With respect to stock price

[15]:
```
navg  =   9
#
# Simulation Parameters␣
 ↪-------------------------------------------------------------
#
b     = 100000   # number of paths
M     = 10       # number of points for each path
dt    = float(T) /float( M)
df    = math.exp(-r * T)
#
#␣
 ↪-------------------------------------------------------------------------------
#
x   = []
```

```
y1  = []
y2  = []
y3  = []
for S0 in np.arange(50, 150, 1):
    #
    # Random numbers generations. We use the function 'randn' which returns
    # a sample (or samples) from the "standard normal" distribution.
    # If positive, int_like or int-convertible arguments are provided,
    # 'randn' generates an array of shape (d0, d1, ..., dn), filled
    # with random floats sampled from a univariate "normal" (Gaussian)
    # distribution of mean 0 and variance 1
    #
    z1  = np.random.randn(M, b)
    #
    # Stock Price Paths. The function 'cumsum' returns the cumulative sum
    # of the elements along a given axis, in this case we want to sum along
    # the time axis (axis=0)
    #
    ␣
↪#-------------------------------------------------------------------------------
    #
    # Simple Montecarlo without variance reduction
    #
    S1 = S0 * np.exp(np.cumsum((r - 0.5 * sigma ** 2) * dt
        + sigma * math.sqrt(dt) * z1, axis=0))
    S1 = np.insert(S1, 0, S0, axis=0)

    avg  = np.average(S1[-navg:-1,:], axis=0)
    gavg = gmean     (S1[-navg:-1,:], axis=0)

    opt_asian_1      = np.maximum(avg   - K, 0) * df
    std_dev_1        = np.std(opt_asian_1)/math.sqrt(float(b))
    opt_asian_1      = np.average(opt_asian_1)
    #
    ␣
↪#-------------------------------------------------------------------------------
    #
    # Antithetic Variable
    #
    z2  = np.random.randn(M, int(b/2))
    #
    S21 = S0 * np.exp(np.cumsum((r - 0.5 * sigma ** 2) * dt
          + sigma * math.sqrt(dt) * z2, axis=0))
    S22 = S0 * np.exp(np.cumsum((r - 0.5 * sigma ** 2) * dt
          + sigma * math.sqrt(dt) * (-z2), axis=0))
    S21 = np.insert(S21, 0, S0, axis=0)
    S22 = np.insert(S22, 0, S0, axis=0)
```

```
    avg1  = np.average(S21[-navg:-1,:], axis=0)
    avg2  = np.average(S22[-navg:-1,:], axis=0)

    opt_asian_21 = np.maximum(avg1 - K, 0) * df
    opt_asian_22 = np.maximum(avg2 - K, 0) * df

    opt_asian_2 = 0.5 * (opt_asian_21 + opt_asian_22)
    std_dev_2   = np.std(opt_asian_2)/math.sqrt(float(b))
    opt_asian_2 = np.average(opt_asian_2)
    #
    ␣
 ↪#-------------------------------------------------------------------------------
    #
    # Control Variate
    #
    opt_estimate    = np.maximum(avg   - K, 0) * df
    control_variate = np.maximum(gavg  - K, 0) * df

    covariance      = np.cov(control_variate, opt_estimate)
    variance        = np.var(control_variate)

    beta            = covariance[0][1]/variance

    opt_asian_g_teo = AsianGeometric(1, S0, K, r, delta, sigma, T, navg)
    opt_asian_3     = opt_estimate - beta*(control_variate - opt_asian_g_teo)
    std_dev_3       = np.std(opt_asian_3)/math.sqrt(float(b))
    opt_asian_3     = np.average(opt_asian_3)

    x.append(S0)
    y1.append(std_dev_1)
    y2.append(std_dev_2)
    y3.append(std_dev_3)

    sys.stdout.write("\r" + 'Stock price : ' + str(S0))
    sys.stdout.flush()

#
#␣
 ↪-------------------------------------------------------------------------------
#
if len(x) > 1:

    y_inf = 0.9
    y_sup = 2.0 - y_inf
```

```
    df1 = pd.DataFrame({'simple mc':y1, 'antithetic':y2, 'control variate':y3},␣
↪index=x)
    ax = df1.plot.line()
    ax.set_xlim((min(x), max(x)))

    plt.title('Standard Error Vs Stock Price', fontsize=12)
    plt.xlabel('Stock Price',                  fontsize=12)
    plt.ylabel('MC Standard Error',            fontsize=12)
    plt.grid(True)
    plt.show()
```

```
Stock price : 149
```

With respect to σ

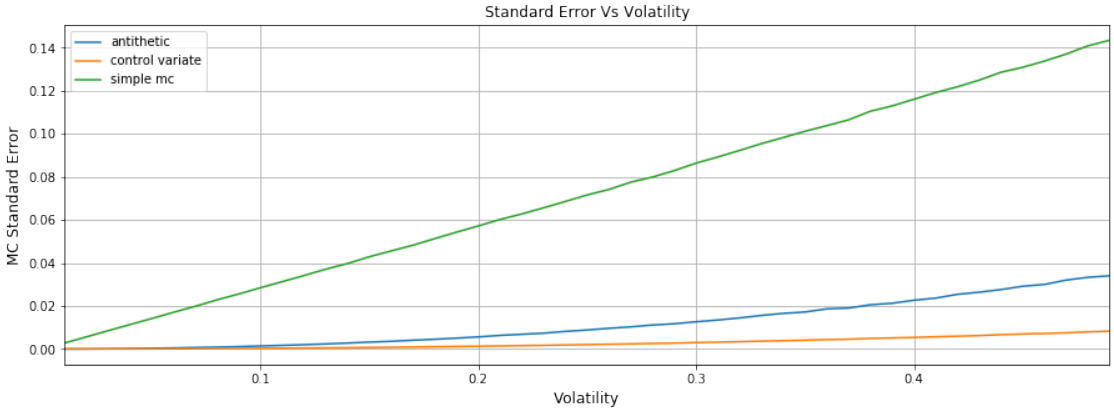
[17]:
```
navg  =   24
#
# Simulation Parameters␣
 ↪-------------------------------------------------------------
#
b     = 100000    # number of paths
M     = 25       # number of points for each path
dt    = float(T) /float( M)
df    = math.exp(-r * T)
#
#␣
 ↪-------------------------------------------------------------------------------
#
x   = []
y1  = []
y2  = []
y3  = []
```

```
for sigma in np.arange(.01, .5, .01):
    #
    # Random numbers generations. We use the function 'randn' which returns
    # a sample (or samples) from the "standard normal" distribution.
    # If positive, int_like or int-convertible arguments are provided,
    # 'randn' generates an array of shape (d0, d1, ..., dn), filled
    # with random floats sampled from a univariate "normal" (Gaussian)
    # distribution of mean 0 and variance 1
    #
    z1  = np.random.randn(M, b)
    #
    # Stock Price Paths. The function 'cumsum' returns the cumulative sum
    # of the elements along a given axis, in this case we want to sum along
    # the time axis (axis=0)
    #
  ␣
↪#-------------------------------------------------------------------------------
    #
    # Simple Montecarlo without variance reduction
    #
    S1 = S0 * np.exp(np.cumsum((r - 0.5 * sigma ** 2) * dt
        + sigma * math.sqrt(dt) * z1, axis=0))
    S1 = np.insert(S1, 0, S0, axis=0)

    avg  = np.average(S1[-navg:-1,:], axis=0)
    gavg = gmean     (S1[-navg:-1,:], axis=0)

    opt_asian_1      = np.maximum(avg   - K, 0) * df
    std_dev_1        = np.std(opt_asian_1)/math.sqrt(float(b))
    opt_asian_1      = np.average(opt_asian_1)
    #
  ␣
↪#-------------------------------------------------------------------------------
    #
    # Antithetic Variable
    #
    z2  = np.random.randn(M, int(b/2))
    #
    S21 = S0 * np.exp(np.cumsum((r - 0.5 * sigma ** 2) * dt
          + sigma * math.sqrt(dt) * z2, axis=0))
    S22 = S0 * np.exp(np.cumsum((r - 0.5 * sigma ** 2) * dt
          + sigma * math.sqrt(dt) * (-z2), axis=0))
    S21 = np.insert(S21, 0, S0, axis=0)
    S22 = np.insert(S22, 0, S0, axis=0)

    avg1  = np.average(S21[-navg:-1,:], axis=0)
    avg2  = np.average(S22[-navg:-1,:], axis=0)
```

```
    opt_asian_21 = np.maximum(avg1 - K, 0) * df
    opt_asian_22 = np.maximum(avg2 - K, 0) * df

    opt_asian_2 = 0.5 * (opt_asian_21 + opt_asian_22)
    std_dev_2   = np.std(opt_asian_2)/math.sqrt(float(b))
    opt_asian_2 = np.average(opt_asian_2)
    #
    ␣
↪#-------------------------------------------------------------------------------
    #
    # Control Variate
    #
    opt_estimate    = np.maximum(avg   - K, 0) * df
    control_variate = np.maximum(gavg  - K, 0) * df

    covariance      = np.cov(control_variate, opt_estimate)
    variance        = np.var(control_variate)

    beta            = covariance[0][1]/variance

    opt_asian_g_teo = AsianGeometric(1, S0, K, r, delta, sigma, T, navg)
    opt_asian_3     = opt_estimate - beta*(control_variate - opt_asian_g_teo)
    std_dev_3       = np.std(opt_asian_3)/math.sqrt(float(b))
    opt_asian_3     = np.average(opt_asian_3)

    x.append(sigma)
    y1.append(std_dev_1)
    y2.append(std_dev_2)
    y3.append(std_dev_3)

    sys.stdout.write("\r" + 'Sigma : ' + str(round(sigma,3)))
    sys.stdout.flush()
#
#␣
↪-------------------------------------------------------------------------------
#
if len(x) > 1:

    df1 = pd.DataFrame({'simple mc':y1, 'antithetic':y2, 'control variate':y3},␣
↪index=x)
    ax = df1.plot.line()
    ax.set_xlim((min(x), max(x)))

    plt.title('Standard Error Vs Volatility', fontsize=12)
    plt.xlabel('Volatility',              fontsize=12)
    plt.ylabel('MC Standard Error',       fontsize=12)
```

```
    plt.grid(True)
    plt.show()
```

```
Sigma : 0.49
```



1.5 Moment Matching

The moment matching method try to reduce the estimator variance through the adjustment of the samples extracted from a standardized normal distribution in order to ensure the equality between the sample moments (generally the first and the second) and the corresponding moments of the probabilistic distribution.

If we indicate with Z_i a sample of random variables, to ensure the equality of the first two moments, we calculate the sample mean m and the sample standard deviation s. So we define the adjusted samples as follows

$$Z_i' = \frac{Z_i - m}{s}$$

1.6 References

Paul Glasserman, **Monte Carlo Methods in Financial Engineering**, Springer (2004)