# FinancialMathematics

Nick Syring

2023-02-22

# Contents

# Chapter 1

# Introduction

# Chapter 2

# Introduction

# Chapter 3

# The risk-neutral framework

A key concept to understanding this section is the existence of a risk-free interest rate $r$ at which one may invest cash to accumulate into the future with no chance of loss/default. In practice, US treasury rates or LIBOR may be used as near-riskless rates.

## 3.1 Forwards

A forward is a promise now to buy an asset (think stock) at a prescribed future time $T$. What should be the price of the forward, which is paid at time $T$?

Someone with experience in probability or statistics might come up with the following answer. Suppose we have a perfect model of the future price, say, $\log S_T/S_0 = X$ where $X \sim N(\mu, \sigma^2)$; that is, the log price change is a normal fluctuation up or down. The price of the forward, say $F$, is paid at time $T$, so that the payoff of the forward is $S_T - F$. The value of the payoff at time 0 is $e^{-rT}(S_T - F)$ and the expected value of the payoff at time 0 is

$$E(e^{-rT}(S_T - F)) = e^{-rT}\left[S_0 e^{\mu+\sigma^2/2} - F\right].$$

It would seem that either the holder or the seller of the forward contract would be disadvantaged if $F$ is anything other than $S_0 e^{\mu+\sigma^2/2}$, but this is incorrect!

Here's why. Pretend you are the seller/writer of the forward contract. Borrow $S_0$ at time $t = 0$ and buy one share. At time $T$ deliver the stock for $K$ and repay the loan of $S_0 e^{rT}$. If $K > S_0 e^{rT}$ you have made riskless profit, i.e., *arbitrage*. On the other hand, suppose you are the buyer/holder of the forward contract. Short a share at time 0 and recieve $S_0$, investing it at the risk free rate $r$. At time $T$ you have $S_0 e^{rT}$, you buy the stock for $F$, and you close your short position. If $F < S_0 e^{rT}$ you make riskless profit.

The pricing lesson is that the forward price $F$ must simply be the current asset price accumulated at the risk-free rate; any other price results in arbitrage. The bigger picture lesson is that the probabilities of asset value changes were irrelevant—only the initial asset value and the risk free rate (information known at time 0) played a role in the arbitrage-free pricing of the forward contract. The phenomenon that arbitrage-free prices are agnostic towards probabilities of future asset value moves is known as *risk-neutrality*.

Forwards are simple financial instruments, but it turns out that the above argument for risk-neutral pricing applies to arbitrary derivatives, as we will see next.

## 3.2 Risk-neutral pricing of derivatives

Consider a derivative that provides a payoff based on the value of an underlying asset, for example, a stock. By modeling the stock's future price moves we can replicate the derivative using a portfolio consisting of $(\phi, \psi)$ units of the stock and a risk-free bond.

Our model consists of probabilities of prescribed up or down moves at discrete time points. For example, the stock that has value $S_0$ at time 0 either moves up to value $S_2$ at time 1 with probability $p$ or down to value $S_1$ at time 1 with probability $1-p$. The portfolio has value either $\phi S_2 + \psi B_0 e^r$ or $\phi S_1 + \psi B_0 e^r$ at time 1. And, the derivative has value either $f(S_2)$ or $f(S_1)$ at time 1. Set the potential portfolio values equal to the potential derivative values and solve for $\phi, \psi$, yielding

$$\phi = \frac{f(S_2) - f(S_1)}{S_2 - S_1} \quad \text{and} \quad \psi = B_0^{-1} e^{-r}[f(S_2) - \phi S_2].$$

The portfolio value at time 0 is

$$\phi S_0 + \psi B_0 = S_0 \frac{f(S_2) - f(S_1)}{S_2 - S_1} + e^{-r} \left\{ f(S_2) - \frac{f(S_2) - f(S_1)}{S_2 - S_1} S_2 \right\}.$$

This is an enforceable price for the derivative because any deviation up or down (in excess of bid-ask spreads) creates an arbitrage opportunity using the replicating portfolio and the derivative.

Let $q := (S_0 e^r - S_1)/(S_2 - S_1)$ and note that the portfolio value may be rewritten as

$$e^{-r} \left\{ q f(S_2) + (1-q) f(S_1) \right\}.$$

This appears to be a discounted expected value. And, indeed, if $q < 0$ the future stock price is definitely higher than the bond's future value, while if $q > 1$ the bond is certainly more valuable than the stock. Therefore $q \in (0, 1)$ behaves like a probability. We call the probabilities $(q, 1-q)$ the *risk-neutral probabilities*

or risk-neutral measure because they do not depend on the underlying true probabilities $(p, 1-p)$ of up and down stock moves; rather they are the result ofthe arbitrage-free market constraint.

## 3.3 Binomial derivative pricing model

The previous argument for derivative pricing using a replicating portfolio based on up and down asset moves suggests a practical pricing model. In real life the asset may move anywhere in a large, continuous range from time point to time point, and may experience substantial volatility, so a model only allowing one up and one down move is not very realistic. But, if we simply increase the number of time points within a time interval and allow for an up or down jump at each time point, we can build a risch model of asset price over time.

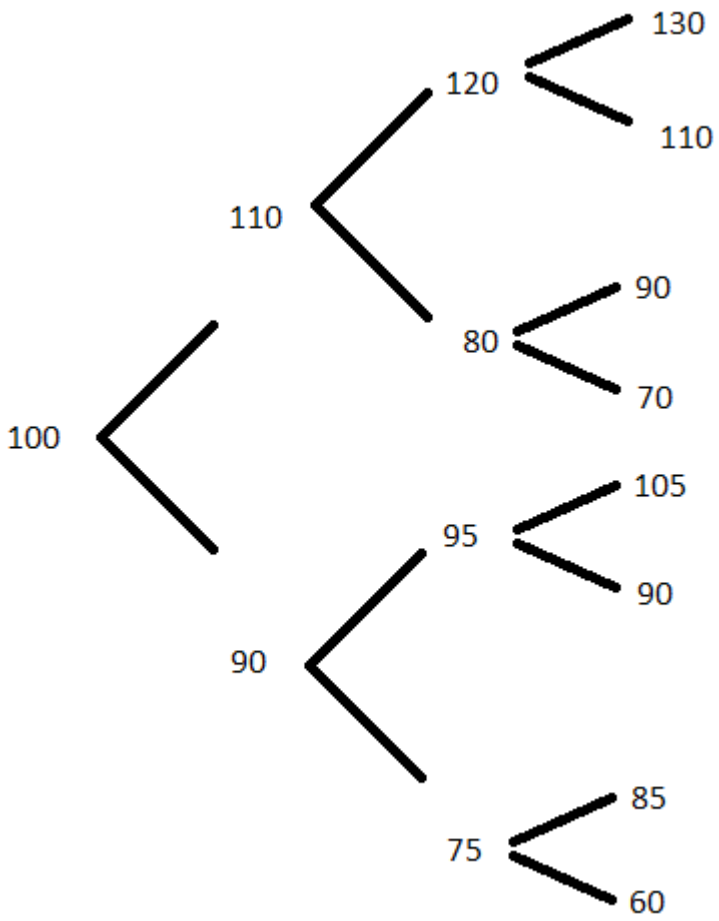
Let $\Delta_t$ denote the size of the time step. At time point $i$ the stock takes one of $2^i$ values, and, at time $i+1$, $2^{i+1}$ values. Suppose we are at time $i$, and value $S_j^i$ of $2^i$. Then,

$$f(S_j^i) = e^{-r\Delta_t}\left\{q_j^i f(S_{j,u}^i) + (1-q_j^i) f(S_{j,d}^i)\right\}$$

where $(S_{j,u}^i, S_{j,d}^i)$ are the up and down stock moves from $S_j^i$, and

$$q_j^i = \frac{S_j^i e^{r\Delta_t} - S_{j,d}^i}{S_{j,u}^i - S_{j,d}^i}.$$

Notice that the derivative prices can now be computed recursively from the top of the tree (the end of the time interval) to the bottom (the start of the time interval). For example, consider the following four-period tree of stock prices (with jump probabilities omitted):




Suppose the derivative we are pricing is a European call with strike 95. At time 3, from top to bottom, the payoffs are 35, 15, 0, 0, 10, 0, 0, and 0. Suppose $r = 3\%$ and the timestep is $\Delta_t = 1$ for simplicity. Then, at time 2 at node 120, we have $q = (120e^{0.03} - 110)/20 = 0.6827$ and $1 - q = 0.3173$. The price of the derivative at node 120 is $e^{-0.03}(0.6827 \times 35 + 0.3173 \times 15) = 27.8077$. The price at the 80 node is 0.

Similarly, the price of the derivative at the 95 node is $10e^{-0.03} \cdot q$ where $q = (95e^{0.03} - 90)/15 = 0.5262$, yielding a price of 5.1066.

Backtracking to the time 1 nodes, we have derivative prices of 22.50 at node 110 and 4.3959 at node 90. Finally, at time 0 (and node 100) we find the price is 15.72579.

# Chapter 4

# The geometric Brownian motion model of asset value and Monte Carlo simulation

## 4.1 Asset values as random variables

Let $\{S_t, t \geq 0\}$ denote the values of an asset at times $t$. Ideally, we think of time as a continuum, so that $S_t$ is a *continuous process*. Practically, however, we will approximate the continuous process by a discrete one by performing computations associated with the process at a grid of $t$ values in some interval $[0, T]$ with mesh-size $\Delta t$.

We model the process $S_t$ as a random or stochastic sequence; essentially, it is just a sequence of realizations of random variables.

Here is a plot of Apple's (AAPL) close-of-day stock price and logarithm of return $\log(S_t/S_{t-1})$ from its initial listing to the present day. Stock prices are available much more often than daily, so the data visualized below already represent a discretization of the underlying process (with mesh-size 1 day).

```
import numpy as np
import numpy.random as npr
import matplotlib as mpl
from matplotlib.pylab import plt
import math

plt.style.use('seaborn')
mpl.rcParams['font.family'] = 'serif'
```

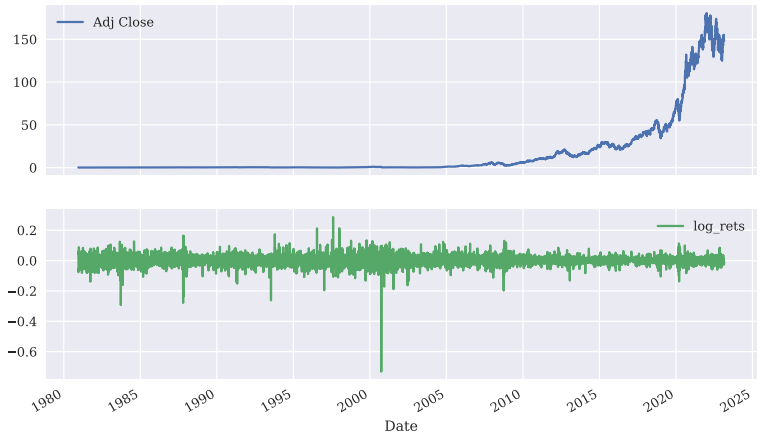
```
import pandas as pd
import yfinance as yf
from yahoofinancials import YahooFinancials

aapl_df = yf.download('AAPL')
```

```
## [**********************100%***********************]  1 of 1 completed
```

```
close_price = aapl_df['Adj Close']
log_rets = np.log(close_price / close_price.shift(1))
aapl_df['log_rets'] = log_rets
aapl_df[['Adj Close', 'log_rets']].plot(subplots=True, figsize=(10, 6))
```

```
## array([<AxesSubplot:xlabel='Date'>, <AxesSubplot:xlabel='Date'>],
##       dtype=object)
```

```
plt.show();
```




## 4.2 Geometric Brownian motion model of asset prices over time

The mathematics of continuous stochastic processes is advanced. Rather than presenting a full account here we focus on an intermediate level of understanding

of one of the most common models used for asset prices. Later we will modify the model to account for several real-life phenomena.

It is folk-wisdom that the log returns of at asset $\log \frac{S_t}{S_{t-1}}$ are normally-distributed, or, rather, are modeled as such. This comes as a consequence of modeling the sequence of asset prices as a *geometric Brownian motion*.

A Brownian motion (which is also called a Wiener process) is essentially a sequence of normal random variables. Specifically, the process $W_t$ satisfies - $W_0 = 0$ - $W_t$ is continuous a.s. - $W_t$ has independent increments - $W_t - W_s \sim N(0, t-s)$ for $0 \leq s \leq t$ Additionally, the sequence is measurable with respect to a filtration, an ordered family of sigma-fields (for details see, for example, Resnick's A Probability Path Chapter 10).

The geometric Brownian motion (gBm) model says changes in asset price from time $t$ to time $t+s$ are determined by a Brownian motion and a drift. This is typically written in the following fashion as a stochastic differential equation (SDE) with respect to instantaneous price change:

$$dS_t = rS_t dt + \sigma S_t dW_t$$

where $r$ is a risk-free interest rate, $dt$ is an instantaneous change in time, $\sigma$ is a volatility (standard deviation) parameter, and $W_t$ is a Brownian motion.

It is important to keep in mind the SDE doesn't really mean anything—rather, it is simply notation used to express a stochastic integral in a concise manner. A more meaningful expression of the geometric Brownian motion model is given by the difference equation

$$S_{t+s} = S_t = \int_t^{t+s} rS_u \, du + \int_t^{t+s} \sigma S_u \, dW_u.$$

The primary challenge to overcome is how to define integration with respect to the Brownian motion $W_u$. For various technical reasons, such an integration cannot behave exactly in the same way integration works for real-valued functions, i.e., Riemann integration. A new theory of integration, Ito integration, is needed.

Rather than giving a thorough treatment of Ito's calculus, we will simply provide an informal derivation of Ito's Lemma, which is enough to provide a "solution" to the geometric Brownian motion. Let $f(t, S_t)$ be a function of time and asset price at time $t$. Take a two term Taylor expansion of $f$ and use the geometric Brownian motion SDE in the chain rule as follows:

$$\begin{aligned} df &= \frac{\partial f}{\partial t} dt + \frac{\partial f}{\partial S_t} dS_t + \frac{1}{2} \frac{\partial^2 f}{\partial S_t^2} dS_t^2 + \cdots \\ &= \frac{\partial f}{\partial t} dt + \frac{\partial f}{\partial S_t} (rS_t dt + \sigma S_t dW_t) + \frac{1}{2} \frac{\partial^2 f}{\partial S_t^2} (r^2 S_t^2 dt^2 + 2r\sigma S_t^2 dt dW_t + \sigma^2 S_t^2 dW_t^2) + \cdots . \end{aligned}$$

Then, Ito's Lemma says $dW_t^2 = O(dt)$ and the substitution $dW_t^2 = dt$ is justified, while the terms $dt^2$ and $dt\,dW_t$ are ignorable and may be substituted by zero. The Taylor expansion simplifies, according to Ito, to

$$df = \left( \frac{\partial f}{\partial t} + rS_t \frac{\partial f}{\partial S_t} + \frac{\sigma^2}{2} S_t^2 \frac{\partial^2 f}{\partial S_t^2} \right) dt + \sigma \frac{\partial f}{\partial S_t} S_t \, dW_t.$$

Now, let $f(t, S_t) := \log(S_t)$, the log asset price at time $t$. In that case, we have the following derivatives:

$$\partial f/\partial t = 0 \quad \partial f/\partial S_t = 1/S_t \quad \partial^2 f/\partial S_t^2 = -1/S_t^2.$$

Substituting these into the SDE above, we get

$$d \log S_t = r\,dt - \frac{\sigma^2}{2S_t^2} S_t^2\,dt + \frac{\sigma}{S_t} S_t\,dW_t.$$

Next integrate both sides:

$$\log S_t = \log(S_0) + \left( r - \frac{\sigma^2}{2} \right) t + \sigma\, W_t.$$

Exponentiate to obtain

$$S_t = S_0 \exp\left( rt - \frac{\sigma^2}{2} t + \sigma\, W_t \right).$$

Recall that $W_t - W_0 := W_t$ has variance $t$ and see that $S_t/S_0$ is log-normally distributed with parameters $(r - \sigma^2/2)t$ and $\sigma\, t^{1/2}$, which means it is right-skewed with mean $\exp(rt)$ and variance $(\exp(\sigma^2 t) - 1) \cdot \exp(2rt)$.

Armed with Ito's Lemma, we have confirmed the folk-wisdom that log-returns from time $t$ to $t + s$ are (modeled as) normal random variables with mean $(r - \frac{\sigma^2}{2})s$ and variance $\sigma^2 s$. One last minor point: if $S_t/S_0$ has a lognormal distribution with density $f$, then the density of $S_t$ is simply

$$g(s) = S_0^{-1} f(s/S_0),$$

a scaled log-normal. This is helpful, e.g., for comparing the exact distribution of $S_t$ to MC samples values of $S_t$, as we do below.

## 4.3 Monte Carlo simulation of the gBm model

As we showed above, there is an exact solution to the geometric Brownian motion SDE—namely, a continuous time random process characterized by independent, normal log-returns over disjoint time periods. Equivalently, given

the asset value $S_0$ at time zero we know $S_t/S_0$ is log-normally distributed with the parameters given above.

Later, we will modify (complicate) the gBm model to take into account several real-life phenomena including, e.g., time-varying volatility. The augmented models do not necessarily have explicit solutions like the gBm model. Alternatively, we can simulate many times from the model to compute approximate solutions—this is called Monte Carlo. It's not needed for the gBm model, but we will illustrate it here in order to take advantage of the simplified setting of gBm before moving on to more complicated models.

Below we compare three methods for computing $S_T$ at time $T$, given $S_0$, $\sigma$, and $r$: the explicit solution due to Ito, a Monte Carlo (MC) simulation from the lognormal distribution, and a MC simulation of *paths* of asset values $S_s$ for a discretization $s \in \{s_0 = 0, s_1, \ldots, s_{M+1} = T\}$. All three provide the same answer (in a distributional sense) but the MC procedures contain some additional MC variability (noise) that decreases as the number of simulations increases.

```
import numpy as np
import numpy.random as npr
import matplotlib as mpl
from matplotlib.pylab import plt
import math
from scipy.stats import lognorm
plt.style.use('seaborn')
mpl.rcParams['font.family'] = 'serif'


S0 = 100
r = 0.05
sigma = 0.25
T = 2.0

# Exact density of ST based on gBm model
mu = np.exp((r - sigma**2/2)*T)
s = sigma * math.sqrt(T)
x = np.linspace(lognorm.ppf(0.001, s, scale = mu), lognorm.ppf(0.999, s, scale = mu), 1000)


# MC sampling of density of ST
I=10000
ST = S0 * np.exp((r-0.5*sigma**2)*T + sigma*math.sqrt(T)*npr.standard_normal(I))


# MC sampling of asset price path S0 to ST at 50 equally-spaced timepoints
M=50
dt = T / M
```

```
S = np.zeros((M+1, I))
S[0] = S0
for t in range(1,M+1):
    S[t]=S[t-1]*np.exp((r - 0.5*sigma**2)*dt + sigma*math.sqrt(dt)*npr.standard_normal
```

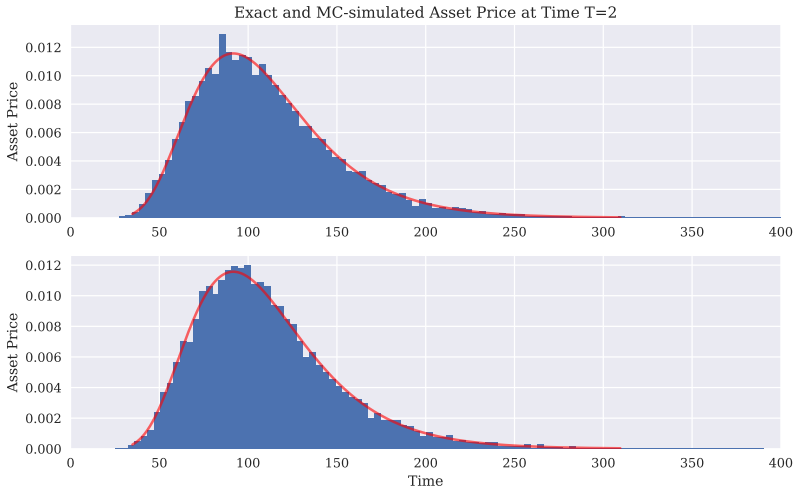
```
# Plots of asset Price distribution at T = 2
plt.figure(figsize = (10,6))
plt.subplot(211)
# histogram of MC samples from scaled log-normal
hist1 = plt.hist(ST,100,density = True)
# scaled log-normal density
dens = plt.plot(x*S0, (1/S0)*lognorm.pdf(x, s, scale = mu),
       'r-', lw=2, alpha=0.6)
plt.ylabel('Asset Price')
plt.title('Exact and MC-simulated Asset Price at Time T=2')
plt.xlim([0,400])
#plt.ylim([0,0.0014])
```

```
## (0.0, 400.0)
```

```
plt.subplot(212)
# histogram of path-wise MC samples from scaled log-normal over grid of 50 times
hist2 = plt.hist(S[-1],100,density = True)
dens = plt.plot(x*S0, (1/S0)*lognorm.pdf(x, s, scale = mu),
       'r-', lw=2, alpha=0.6)
plt.ylabel('Asset Price')
plt.xlabel('Time')
plt.xlim([0,400])
#plt.ylim([0,0.0014])
```

```
## (0.0, 400.0)
```

```
plt.show()
```




## 4.4 Off on a tangent: reducing MC variability

The difference between the theoretic (exact) values for the mean and standard deviation of $S_T$ and the corresponding MC approximations (shown below) is due to MC error. The Law of Large Numbers (LLN) implies MC error declines to zero as the number of MC samples increases to infinity. In practice, using many more MC samples in order to reduce MC error has the trade-off of increasing computation time (and memory usage if storing those random variates in a vector). On the other hand, there are a few ways to reduce MC error by making more clever choices of MC samples.

```
import scipy.stats  as scs

def print_stats(a2,a3):
    stat2 = scs.describe(a2)
    stat3 = scs.describe(a3)
    print('%14s %14s %14s %14s' % ('statistic', 'data set 1', 'data set 2', 'data set 3'))
    print(45 * '-')
    print('%14s %14s %14.3f %14.3f' % ('size', 'NA', stat2[0], stat3[0]))
    print('%14s %14s %14.3f %14.3f' % ('min', 'NA', stat2[1][0], stat3[1][0] ))
    print('%14s %14s %14.3f %14.3f' % ('max', 'NA', stat2[1][1], stat3[1][1]))
    print('%14s %14.3f %14.3f %14.3f' % ('mean', S0*math.exp(r*T), stat2[2], stat3[2]))
    print('%14s %14.3f %14.3f %14.3f' % ('stdev', S0*math.sqrt((math.exp(sigma**2 * T)-1)*math.ex


print_stats(ST, S[-1])
```

```
##      statistic     data set 1     data set 2     data set 3
## ---------------------------------------------------------
##           size             NA      10000.000      10000.000
##            min             NA         27.282         24.947
##            max             NA        402.374        390.602
##           mean        110.517        110.065        110.384
##          stdev         40.327         40.454         39.955
```

The simplest way to reduce MC variability (besides increasing the number of samples) is to use *anti-thetical variates*. When sampling standard normal random variates this amounts to sampling $z$ and using both $(z, -z)$ as samples. To get $2I$ samples one only needs to actually compute $I$ samples, the remaining $I$ are the same in magnitude but with the opposite signs. This accomplishes exact mean-matching, i.e., $(2I)^{-1} \sum_{i=1}^{2I} z_i = 0$, exactly the standard normal distribution mean.

Another method for reducing MC variability is second-moment matching. Suppose we generate antithetical samples $z = (z_1, \ldots, z_I)$. Let $z_i^\star := z_i / s_z$ for $i = 1, \ldots, I$ where $s_z$ is the sample standard deviation of $z$. Now, $z^\star$ has sample mean exactly zero and sample standard deviation exactly 1.

Further, a Box-Cox transformation may be used if the samples $z$ have positive skew. (If they have negative skew, then the samples may be reflected about the maximum value to produce a set of positively-skewed values starting from zero.) After applying the Box-cox transformation, the resulting values $y$ should be close to symmetric (closer to a normal distribution than the original variates) and a further standardization $z_i^\star = (y_i - \overline{y})/s_y$ may be used to transform the values to approximately standard normal.

The following simulation supports the use of standardized antithetic variates. By design, these had exactly zero mean and unit variance, but were slightly more likely to be skewed compared to vanilly MC samples. Box-Cox transformed variates did not perform better, and were more likely to display excess kurtosis than even vanilla MC variates. It is worth pointing out that antithetic variates may be produced sequentially while standardized variates requires storing all variates in memory, which may be a problem in some applications.

```
import scipy.stats  as scs
import numpy  as np
import numpy.random  as npr

def bc_standardize(z):
  M = np.max(z)
  zM = M-z+0.0000001
  y = scs.boxcox(zM)
  m = np.mean(y[0])
  s = np.std(y[0], ddof=1)
```

```
  z_star = (y[0]-m)/s
  return z_star

def at_standardize(z):
  s = np.std(z, ddof=1)
  z_star = z/s
  return z_star

kurtosis_vals = np.zeros((1000,8))
skew_vals = np.zeros_like(kurtosis_vals)
mean_vals = np.zeros_like(kurtosis_vals)
std_vals = np.zeros_like(kurtosis_vals)

def print_stats2():
  for i in range(1,1001):
    X = npr.standard_normal(10000)
    Z1 = npr.standard_normal(5000)
    Z2 = npr.standard_normal(5000)
    Z3 = -Z1
    Z4 = -Z2
    Z = np.concatenate((Z1,Z2))
    Y = np.concatenate((Z1,Z3))
    W = np.concatenate((Z2,Z4))
    Ystar = bc_standardize(Y)
    Wstar = bc_standardize(W)
    Yst = at_standardize(Y)
    Wst = at_standardize(W)
    stat1 = scs.describe(X, ddof = 1)
    stat2 = scs.describe(Z, ddof = 1)
    stat3 = scs.describe(Y, ddof = 1)
    stat4 = scs.describe(W, ddof = 1)
    stat5 = scs.describe(Ystar, ddof = 1)
    stat6 = scs.describe(Wstar, ddof = 1)
    stat7 = scs.describe(Yst, ddof = 1)
    stat8 = scs.describe(Wst, ddof = 1)
    mean_vals[i-1,:] = np.array([stat1[2], stat2[2], stat3[2], stat4[2], stat5[2], stat6[2], stat
    std_vals[i-1,:] = np.array([stat1[3], stat2[3], stat3[3], stat4[3], stat5[3], stat6[3], stat7
    kurtosis_vals[i-1,:] = np.array([stat1[4], stat2[4], stat3[4], stat4[4], stat5[4], stat6[4],
    skew_vals[i-1,:] = np.array([stat1[5], stat2[5], stat3[5], stat4[5], stat5[5], stat6[5], stat

  print('%14s %14s %14s %14s %14s %14s %14s %14s %14s' % ('statistic', 'data set 1', 'data set 2'
  print('%14s %14.3f %14.3f %14.3f %14.3f %14.3f %14.3f %14.3f %14.3f' % ('mean mean', np.mean(me
  print('%14s %14.3f %14.3f %14.3f %14.3f %14.3f %14.3f %14.3f %14.3f' % ('std mean', np.std(mean
  print('%14s %14.3f %14.3f %14.3f %14.3f %14.3f %14.3f %14.3f %14.3f' % ('mean std', np.mean(std
  print('%14s %14.3f %14.3f %14.3f %14.3f %14.3f %14.3f %14.3f %14.3f' % ('std std', np.std(std_v
```

```
  print('%14s %14.3f %14.3f %14.3f %14.3f %14.3f %14.3f %14.3f %14.3f' % ('mean kurtosi
  print('%14s %14.3f %14.3f %14.3f %14.3f %14.3f %14.3f %14.3f %14.3f' % ('std kurtosis
  print('%14s %14.3f %14.3f %14.3f %14.3f %14.3f %14.3f %14.3f %14.3f' % ('mean skew',
  print('%14s %14.3f %14.3f %14.3f %14.3f %14.3f %14.3f %14.3f %14.3f' % ('std skew', r

print_stats2()
```

```
##      statistic     data set 1     data set 2     data set 3     data set 4     data
##      mean mean          0.000         -0.000          0.000         -0.000
##       std mean          0.009          0.010          0.000          0.000
##       mean std          1.000          0.999          1.000          0.999
##        std std          0.014          0.014          0.020          0.020
##  mean kurtosis         -0.001          0.001          0.000          0.000
##   std kurtosis          0.024          0.024          0.000          0.000
##      mean skew         -0.001          0.001          0.001         -0.001
##       std skew          0.048          0.049          0.068          0.070
```

# Chapter 5

# A flexible derivative pricing model

The model we consider in this section is the Bakshi-Cao- (BCC) model from "Empirical Performance of Alternative Option Pricing Models" in **The Journal of Finance**, Vol. 52, No. 5 (Dec., 1997), pp. 2003-2049.

The BCC model is characterized by a jump-diffusion model of asset price, with a square-root diffusion process for stochastic volatility, and a Cox-Ingersoll-Ross (CIR) for the stochastic short-term interest rate. Formally, the model may be expressed as follows:

$$\begin{aligned} dS_t &= (r_t - r_J)S_t dt + \sqrt{v_t} S_t dZ_t^1 + J_t S_t dN_t \\ dv_t &= \kappa_v(\theta_v - v_t)dt + \sigma_v \sqrt{v_t} dZ_t^2 \\ dr_t &= \kappa_r(\theta_r - r_t)dt + \sigma_r \sqrt{r_t} dZ_t^3. \end{aligned}$$

The parameters in the above model have the following meanings:

- $S_t$ is the asset (stock/index) price at time $t$
- $r_t$ is the risk-free short term rate at time $t$
- $r_J := \lambda(\exp\{\mu_J + \delta^2/2\} - 1)$ is the drift correction for the jump process
- $v_t$ is the variance at time $t$
- $\kappa_v$ is the speed of adjustment of $v_t$ to its long term average $\theta_t$
- $\theta_t$ is long-term average variance, a mean-reversion
- $\sigma_v$ is the colatility coefficient of the asset price variance

- $Z_t^i$ for $i = 1, 2, 3$ is a Brownian motion with correlations $dZ_t^1 dZ_t^2 = \rho dt$ and zero for the other two pairs

- $N_t$ is a Poisson jump process with intensity $\lambda$

- $J_t$ is a lognormal jump at time $t$, i.e., $\log(1+J_t) \sim N(\log(1+\mu_J) - \delta^2/2, \delta^2)$

- $\kappa_r$ is the speed of adjustment of $r_t$ to its long term average $\theta_r$

- $\theta_r$ is long-term average rate, a mean-reversion

- $\sigma_r$ is the volatility coefficient of the rate

This model is inspired by several real-world observations, including the following:

* Asset price volatility implied by the Black-Scholes model (that is, the volatility value solved for given the observed prices) exhibits the following propoerties in real life

    - varies over time
    - negative correlation with returns
    - varies for different option strike prices
    - varies for different option maturities

* Asset prices/returns exhibit occasional large jumps not modeled well by a Black-Scholes model (geometric Brownian motion) alone

* Interest rates do vary over time (e.g., adjustments by Federal Reserve have an impact) and vary for different maturities (there is a term-structure of interest rates)

## 5.1 Present-value with stochastic short-term rate

Under continuous compounding the present value at time $t$ of a dollar paid at time $T$ is

$$B_0(T) = E_t^Q \left( \exp \left\{ - \int_t^T r_u du \right\} \right).$$

An advantage of the CIR model is that the above discounting has a closed-form:

$$
\begin{aligned}
B_t(T) &= b_1(T)e^{-b_2(T)r_0} \\
b_1(T) &= \left(\frac{2\gamma \exp((\kappa_r+\gamma)T/2)}{2\gamma+(\kappa_r+\gamma)(e^{\gamma T}-1)}\right)^{\frac{2\kappa_r\theta_r}{\sigma_r^2}} \\
b_2(T) &= \frac{2(e^{\gamma T}-1)}{2\gamma+(\kappa_r+\gamma)(e^{\gamma T}-1)} \\
\gamma &= \sqrt{\kappa_r^2+2\sigma_r^2}
\end{aligned}
$$

The following python codes implement discounting under the CIR model.

```
import math
import numpy as np

kappa_r, theta_r, sigma_r, r0, T = 0.3, 0.04, 0.1, 0.04, 1.0

def gamma(kappa_r, sigma_r):
  ''' Help Function. '''
  return math.sqrt(kappa_r ** 2 + 2 * sigma_r ** 2)

def b1(alpha):
  ''' Help Function. '''
  kappa_r, theta_r, sigma_r, r0, T = alpha
  g = gamma(kappa_r, sigma_r)
  return (((2 * g * math.exp((kappa_r + g) * T / 2)) /(2 * g + (kappa_r + g) * (math.exp(g * T) -


def b2(alpha):
  ''' Help Function. '''
  kappa_r, theta_r, sigma_r, r0, T = alpha
  g = gamma(kappa_r, sigma_r)
  return ((2 * (math.exp(g * T) - 1)) /(2 * g + (kappa_r + g) * (math.exp(g * T) - 1)))

def B(alpha):
  ''' Function to value unit zero-coupon bonds in Cox-Ingersoll-Ross (1985)
  model.
  Parameters
  ==========
  r0: float
  initial short rate
  kappa_r: float
  mean-reversion factor
  theta_r: float
  long-run mean of short rate
```

```
  sigma_r: float
  volatility of short rate
  T: float
  time horizon/interval
  Returns
  =======
  zcb_value: float
  zero-coupon bond present value
  '''
  b_1 = b1(alpha)
  b_2 = b2(alpha)
  kappa_r, theta_r, sigma_r, r0, T = alpha
  return b_1 * math.exp(-b_2 * r0)

if __name__ == '__main__':
  #
  # Example Valuation
  #
  B0T = B([kappa_r, theta_r, sigma_r, r0, T])
  # discount factor, ZCB value
  print("ZCB Value %10.4f" % B0T)
```

```
## ZCB Value     0.9608
```

## 5.2 Evaluating a European option via Fourier transform

```
import numpy as np
from scipy.integrate import quad
import warnings
warnings.simplefilter('ignore')

#
# Example Parameters B96 Model
#
## H93 Parameters
kappa_v = 1.5
theta_v = 0.02
sigma_v = 0.15
rho = 0.1
v0 = 0.01
## M76 Parameters
```

```
lamb = 0.25
mu = -0.2
delta = 0.1
sigma = np.sqrt(v0)
## General Parameters
S0 = 100.0
K = 100.0
T = 1.0
r = 0.05

def BCC_call_value(S0, K, T, r, kappa_v, theta_v, sigma_v, rho, v0, lamb, mu, delta):
  ''' Valuation of European call option in B96 Model via Lewis (2001)
  Fourier-based approach.
  Parameters
  ==========
  S0: float
  initial stock/index level
  K: float
  strike price
  T: float
  time-to-maturity (for t=0)
  r: float
  constant risk-free short rate
  kappa_v: float
  mean-reversion factor
  theta_v: float
  long-run mean of variance
  sigma_v: float
  volatility of variance
  rho: float
  correlation between variance and stock/index level
  v0: float
  initial level of variance
  lamb: float
  jump intensity
  mu: float
  expected jump size
  delta: float
  standard deviation of jump
  Returns
  =======
  call_value: float
  present value of European call option
  '''
  int_value = quad(lambda u: BCC_int_func(u, S0, K, T, r, kappa_v, theta_v,sigma_v, rho, v0, lamb
```

```
    call_value = max(0, S0 - np.exp(-r * T) * np.sqrt(S0 * K)/ np.pi * int_value)
    return call_value

def H93_call_value(S0, K, T, r, kappa_v, theta_v, sigma_v, rho, v0):
    ''' Valuation of European call option in H93 model via Lewis (2001)
    Fourier-based approach.
    Parameters
    ==========
    S0: float
    initial stock/index level
    K: float
    strike price
    T: float
    time-to-maturity (for t=0)
    r: float
    constant risk-free short rate
    kappa_v: float
    mean-reversion factor
    theta_v: float
    long-run mean of variance
    sigma_v: float
    volatility of variance
    rho: float
    correlation between variance and stock/index level
    v0: float
    initial level of variance
    Returns
    =======
    call_value: float
    present value of European call option
    '''
    int_value = quad(lambda u: H93_int_func(u, S0, K, T, r, kappa_v, theta_v, sigma_v, rh
    call_value = max(0, S0 - np.exp(-r * T) * np.sqrt(S0 * K)/ np.pi * int_value)
    return call_value

def M76_call_value(S0, K, T, r, v0, lamb, mu, delta):
    ''' Valuation of European call option in M76 model via Lewis (2001)
    Fourier-based approach.
    Parameters
    ==========
    S0: float
    initial stock/index level
    K: float
    strike price
    T: float
```

```
    time-to-maturity (for t=0)
    r: float
    constant risk-free short rate
    lamb: float
    jump intensity
    mu: float
    expected jump size
    delta: float
    standard deviation of jump
    Returns
    =======
    call_value: float
    present value of European call option
    '''
    sigma = np.sqrt(v0)
    int_value = quad(lambda u: M76_int_func_sa(u, S0, K, T, r,
    sigma, lamb, mu, delta), 0, np.inf, limit=250)[0]
    call_value = max(0, S0 - np.exp(-r * T) * np.sqrt(S0 * K)/ np.pi * int_value)
    return call_value

def BCC_int_func(u, S0, K, T, r, kappa_v, theta_v, sigma_v, rho, v0,lamb, mu, delta):
    ''' Valuation of European call option in BCC97 model via Lewis (2001)
    Fourier-based approach: integration function.
    Parameter definitions see function BCC_call_value.'''
    char_func_value = BCC_char_func(u - 1j * 0.5, T, r, kappa_v, theta_v, sigma_v, rho, v0, lamb, m
    int_func_value = 1 / (u ** 2 + 0.25) * (np.exp(1j * u * np.log(S0 / K)) * char_func_value).real
    return int_func_value

def H93_int_func(u, S0, K, T, r, kappa_v, theta_v, sigma_v, rho, v0):
    ''' Valuation of European call option in H93 model via Lewis (2001)
    Fourier-based approach: integration function.
    Parameter definitions see function H93_call_value.'''
    char_func_value = H93_char_func(u - 1j * 0.5, T, r, kappa_v, theta_v, sigma_v, rho, v0)
    int_func_value = 1 / (u ** 2 + 0.25) * (np.exp(1j * u * np.log(S0 / K)) * char_func_value).real
    return int_func_value


def M76_int_func_sa(u, S0, K, T, r, sigma, lamb, mu, delta):
    ''' Valuation of European call option in M76 model via Lewis (2001)
    Fourier-based approach: integration function.
    Parameter definitions see function M76_call_value.'''
    char_func_value = M76_char_func_sa(u - 0.5 * 1j, T, r, sigma, lamb, mu, delta)
    int_func_value = 1 / (u ** 2 + 0.25) * (np.exp(1j * u * np.log(S0 / K)) * char_func_value).real
    return int_func_value
```

```
def BCC_char_func(u, T, r, kappa_v, theta_v, sigma_v, rho, v0,lamb, mu, delta):
  ''' Valuation of European call option in BCC97 model via Lewis (2001)
  Fourier-based approach: characteristic function.
  Parameter definitions see function BCC_call_value.'''
  BCC1 = H93_char_func(u, T, r, kappa_v, theta_v, sigma_v, rho, v0)
  BCC2 = M76_char_func(u, T, lamb, mu, delta)
  return BCC1 * BCC2

def H93_char_func(u, T, r, kappa_v, theta_v, sigma_v, rho, v0):
  ''' Valuation of European call option in H93 model via Lewis (2001)
  Fourier-based approach: characteristic function.
  Parameter definitions see function BCC_call_value.'''
  c1 = kappa_v * theta_v
  c2 = -np.sqrt((rho * sigma_v * u * 1j - kappa_v)** 2 - sigma_v ** 2 * (-u * 1j - u **
  c3 = (kappa_v - rho * sigma_v * u * 1j + c2) / (kappa_v - rho * sigma_v * u * 1j - c2
  H1 = (r * u * 1j * T + (c1 / sigma_v ** 2)* ((kappa_v - rho * sigma_v * u * 1j + c2)
  H2 = ((kappa_v - rho * sigma_v * u * 1j + c2) / sigma_v ** 2* ((1 - np.exp(c2 * T)) /
  char_func_value = np.exp(H1 + H2 * v0)
  return char_func_value

def M76_char_func(u, T, lamb, mu, delta):
  ''' Valuation of European call option in M76 model via Lewis (2001)
  Fourier-based approach: characteristic function.
  Parameter definitions see function M76_call_value.'''
  omega = -lamb * (np.exp(mu + 0.5 * delta ** 2) - 1)
  char_func_value = np.exp((1j * u * omega + lamb * (np.exp(1j * u * mu - u ** 2 * delt
  return char_func_value

def M76_char_func_sa(u, T, r, sigma, lamb, mu, delta):
  ''' Valuation of European call option in M76 model via Lewis (2001)
  Fourier-based approach: characteristic function "jump component".
  Parameter definitions see function M76_call_value.'''
  omega = r - 0.5 * sigma ** 2 - lamb * (np.exp(mu + 0.5 * delta ** 2) - 1)
  char_func_value = np.exp((1j * u * omega - 0.5 * u ** 2 * sigma ** 2+ lamb * (np.exp
  return char_func_value

if __name__ == '__main__':
  #
  # Example Parameters CIR85 Model
  #
  kappa_r, theta_r, sigma_r, r0, T = 0.3, 0.04, 0.1, 0.04, T
  B0T = B([kappa_r, theta_r, sigma_r, r0, T]) # discount factor
  r = -np.log(B0T) / T
  #
  # Example Values
```

```
  #
  print("M76 Value %10.4f" % M76_call_value(S0, K, T, r, v0, lamb, mu, delta))
  print("H93 Value %10.4f" % H93_call_value(S0, K, T, r, kappa_v, theta_v, sigma_v, rho, v0))
  print("BCC97 Value %10.4f" % BCC_call_value(S0, K, T, r, kappa_v, theta_v,sigma_v, rho, v0, lam
```

```
## M76 Value     7.7611
## H93 Value     6.8672
## BCC97 Value     8.2942
```