Chapter 12. Stochastics

Predictability is not how things will go, but how they can go.

—Raheel Farooq

Nowadays, stochastics is one of the most important mathematical and numerical disciplines in finance. In the beginning of the modern era of finance, mainly in the 1970s and 1980s, the major goal of financial research was to come up with closed-form solutions for, e.g., option prices given a specific financial model. The requirements have drastically changed in recent years in that not only is the correct valuation of single financial instruments important to participants in the financial markets, but also the consistent valuation of whole derivatives books, for example. Similarly, to come up with consistent risk measures across a whole financial institution, like value-at-risk and credit valuation adjustments, one needs to take into account the whole book of the institution and all its counterparties. Such daunting tasks can only be tackled by flexible and efficient numerical methods. Therefore, stochastics in general and Monte Carlo simulation in particular have risen to prominence in the financial field.

This chapter introduces the following topics from a Python perspective:

"Random Numbers"

It all starts with pseudo-random numbers, which build the basis for all simulation efforts; although quasi-random numbers (e.g., based on Sobol sequences) have gained some popularity in finance, pseudo-random numbers still seem to be the benchmark.

"Simulation"

In finance, two simulation tasks are of particular importance: simulation of *random variables* and of *stochastic processes*.

"Valuation"

The two main disciplines when it comes to valuation are the valuation of derivatives with *European exercise* (at a specific date) and *American exercise* (over a specific time interval); there are also instruments with *Bermudan exercise*, or exercise at a finite set of specific dates.

"Risk Measures"

Simulation lends itself pretty well to the calculation of risk measures like value-at-risk, credit value-at-risk, and credit valuation adjustments.

Random Numbers

Throughout this chapter, to generate random numbers,[1] the functions provided by the `numpy.random` subpackage are used:

```
In [1]: import math
        import numpy as np
        import numpy.random as npr  ❶
        from pylab import plt, mpl

In [2]: plt.style.use('seaborn')
        mpl.rcParams['font.family'] = 'serif'
        %matplotlib inline
```

❶ Imports the random number generation subpackage from `NumPy`.

For example, the `rand()` function returns random numbers from the open interval [0,1) in the shape provided as a parameter to the function. The return object is an `ndarray` object. Such numbers can be easily transformed to cover other intervals of the real line. For instance, if one wants to generate random numbers from the interval $[a,b)=[5,10)$, one can transform the returned numbers from `npr.rand()` as in the next example—this also works in multiple dimensions due to `NumPy` broadcasting:

```
In [3]: npr.seed(100)  ❶
        np.set_printoptions(precision=4) ❶

In [4]: npr.rand(10)  ❷
Out[4]: array([0.5434, 0.2784, 0.4245, 0.8448, 0.0047, 0.1216, 0.6707, 0.8259,
               0.1367, 0.5751])

In [5]: npr.rand(5, 5)  ❸
Out[5]: array([[0.8913, 0.2092, 0.1853, 0.1084, 0.2197],
               [0.9786, 0.8117, 0.1719, 0.8162, 0.2741],
               [0.4317, 0.94  , 0.8176, 0.3361, 0.1754],
               [0.3728, 0.0057, 0.2524, 0.7957, 0.0153],
               [0.5988, 0.6038, 0.1051, 0.3819, 0.0365]])

In [6]: a = 5.  ❹
        b = 10.  ❺
        npr.rand(10) * (b - a) + a  ❻
Out[6]: array([9.4521, 9.9046, 5.2997, 9.4527, 7.8845, 8.7124, 8.1509, 7.9092,
               5.1022, 6.0501])

In [7]: npr.rand(5, 5) * (b - a) + a  ❼
Out[7]: array([[7.7234, 8.8456, 6.2535, 6.4295, 9.262 ],
               [9.875 , 9.4243, 6.7975, 7.9943, 6.774 ],
               [6.701 , 5.8904, 6.1885, 5.2243, 7.5272],
               [6.8813, 7.964 , 8.1497, 5.713 , 9.6692],
               [9.7319, 8.0115, 6.9388, 6.8159, 6.0217]])
```

❶ Fixes the seed value for reproducibility and fixes the number of digits for printouts.

❷ Uniformly distributed random numbers as *one-dimensional* `ndarray` object.

❸ Uniformly distributed random numbers as *two-dimensional* `ndarray` object.

❹ Lower limit ...

❺ ... and upper limit ...

❻ ... for the transformation to another interval.

❼ The same transformation for two dimensions.

Table 12-1 lists functions to generate simple random numbers.

Table 12-1. Functions for simple random number generation

Function	Parameters	Returns/result
`rand`	`d0, d1, ..., dn`	Random values in the given shape
`randn`	`d0, d1, ..., dn`	A sample (or samples) from the standard normal distribution
`randint`	`low[, high, size]`	Random integers from `low` (inclusive) to `high` (exclusive)
`random_integers`	`low[, high, size]`	Random integers between `low` and `high`, inclusive
`random_sample`	`[size]`	Random floats in the half-open interval [0.0, 1.0)
`random`	`[size]`	Random floats in the half-open interval [0.0, 1.0)
`ranf`	`[size]`	Random floats in the half-open interval [0.0, 1.0)
`sample`	`[size]`	Random floats in the half-open interval [0.0, 1.0)
`choice`	`a[, size, replace, p]`	Random sample from a given 1D array
`bytes`	`length`	Random bytes

It is straightforward to visualize some random draws generated by selected functions from Table 12-1. Figure 12-1 shows the results graphically for two continuous distributions and two discrete ones:

```
In [8]: sample_size = 500
        rn1 = npr.rand(sample_size, 3)  ❶
        rn2 = npr.randint(0, 10, sample_size)  ❷
        rn3 = npr.sample(size=sample_size)  ❶
        a = [0, 25, 50, 75, 100]  ❸
        rn4 = npr.choice(a, size=sample_size) ❸

In [9]: fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2, ncols=2,
                                                     figsize=(10, 8))
        ax1.hist(rn1, bins=25, stacked=True)
        ax1.set_title('rand')
        ax1.set_ylabel('frequency')
        ax2.hist(rn2, bins=25)
        ax2.set_title('randint')
        ax3.hist(rn3, bins=25)
        ax3.set_title('sample')
        ax3.set_ylabel('frequency')
        ax4.hist(rn4, bins=25)
        ax4.set_title('choice');
```

❶ Uniformly distributed random numbers.

❷ Random integers for a given interval.

❸ Randomly sampled values from a finite `list` object.

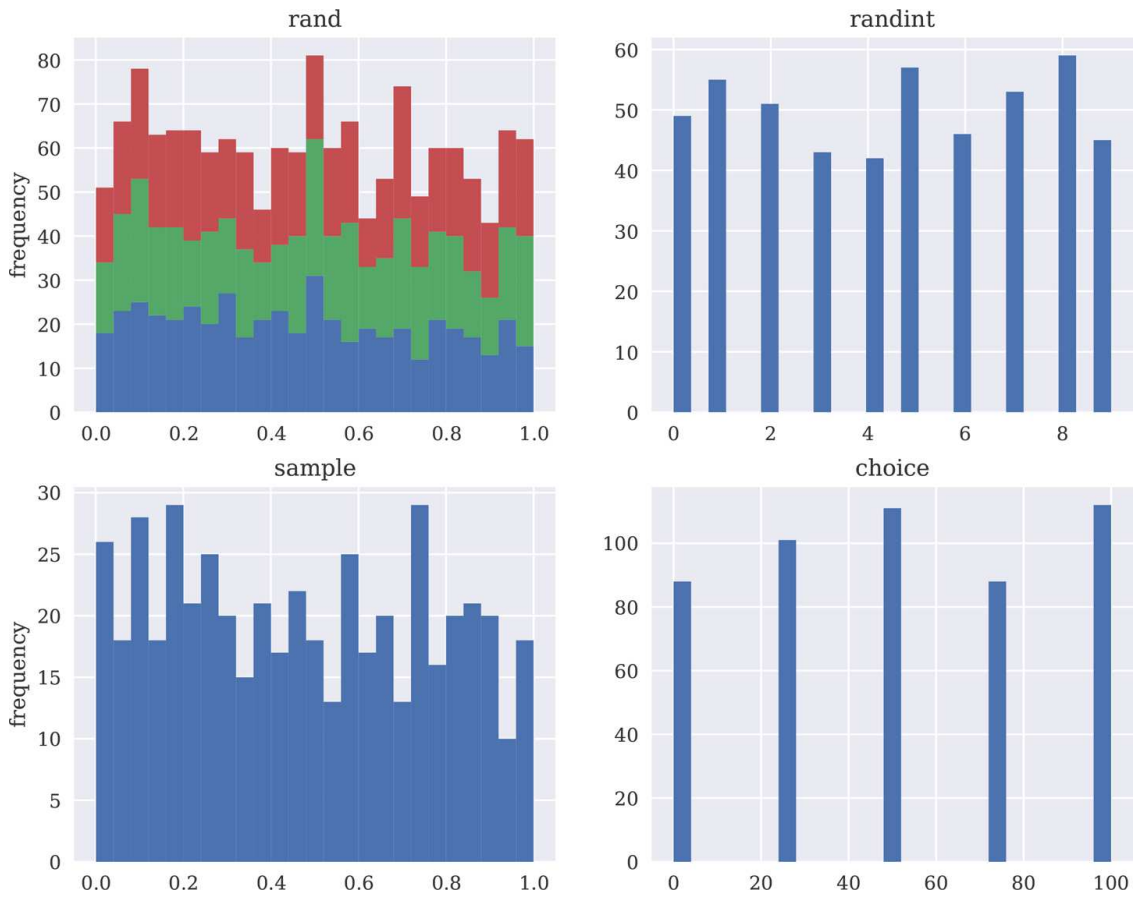

Figure 12-1. Histograms of simple random numbers

Table 12-2 lists functions for generating random numbers according to different distributions.

Table 12-2. Functions to generate random numbers according to different distribution laws

Function	Parameters	Returns/result
`beta`	`a, b [, size]`	Samples for a beta distribution over [0, 1]
`binomial`	`n, p [, size]`	Samples from a binomial distribution
`chisquare`	`df [, size]`	Samples from a chi-square distribution
`dirichlet`	`alpha [, size]`	Samples from the Dirichlet distribution
`exponential`	`[scale, size]`	Samples from the exponential distribution
`f`	`dfnum, dfden [, size]`	Samples from an F distribution
`gamma`	`shape [, scale, size]`	Samples from a gamma distribution
`geometric`	`p [, size]`	Samples from the geometric distribution
`gumbel`	`[loc, scale, size]`	Samples from a Gumbel distribution
`hypergeometric`	`ngood, nbad, nsample [, size]`	Samples from a hypergeometric distribution
`laplace`	`[loc, scale, size]`	Samples from the Laplace or double exponential distribution

Function	Parameters	Returns/result
`logistic`	`[loc, scale, size]`	Samples from a logistic distribution
`lognormal`	`[mean, sigma, size]`	Samples from a log-normal distribution
`logseries`	`p [, size]`	Samples from a logarithmic series distribution
`multinomial`	`n, pvals [, size]`	Samples from a multinomial distribution
`multivariate_normal`	`mean, cov [, size]`	Samples from a multivariate normal distribution
`negative_binomial`	`n, p [, size]`	Samples from a negative binomial distribution
`noncentral_chisquare`	`df, nonc [, size]`	Samples from a noncentral chi-square distribution
`noncentral_f`	`dfnum, dfden, nonc [, size]`	Samples from the noncentral F distribution
`normal`	`[loc, scale, size]`	Samples from a normal (Gaussian) distribution
`pareto`	`a [, size]`	Samples from a Pareto II or Lomax distribution with the specified shape
`poisson`	`[lam, size]`	Samples from a Poisson distribution

Function	Parameters	Returns/result
`power`	`a [, size ]`	Samples in [0, 1] from a power distribution with positive exponent `a – 1`
`rayleigh`	`[ scale , size ]`	Samples from a Rayleigh distribution
`standard_cauchy`	`[ size ]`	Samples from standard Cauchy distribution with mode = 0
`standard_exponential`	`[ size ]`	Samples from the standard exponential distribution
`standard_gamma`	`shape [, size ]`	Samples from a standard gamma distribution
`standard_normal`	`[ size ]`	Samples from a standard normal distribution (mean=0, stdev=1)
`standard_t`	`df [, size ]`	Samples from a Student's t distribution with `df` degrees of freedom
`triangular`	`left , mode , right [, size ]`	Samples from the triangular distribution over the interval `[ left , right ]`
`uniform`	`[ low , high , size ]`	Samples from a uniform distribution
`vonmises`	`mu , kappa [, size ]`	Samples from a von Mises distribution

Function	Parameters	Returns/result
`wald`	`mean , scale [, size ]`	Samples from a Wald, or inverse Gaussian, distribution
`weibull`	`a [, size ]`	Samples from a Weibull distribution
`zipf`	`a [, size ]`	Samples from a Zipf distribution

Although there is much criticism around the use of (standard) normal distributions in finance, they are an indispensable tool and still the most widely used type of distribution, in analytical as well as numerical applications. One reason is that many financial models directly rest in one way or another on a normal distribution or a log-normal distribution. Another reason is that many financial models that do not rest directly on a (log-)normal assumption can be discretized, and therewith approximated for simulation purposes, by the use of the normal distribution.

As an illustration, Figure 12-2 visualizes random draws from the following distributions:

- *Standard normal* with mean of 0 and standard deviation of 1
- *Normal* with mean of 100 and standard deviation of 20
- *Chi square* with 0.5 degrees of freedom
- *Poisson* with lambda of 1

Figure 12-2 shows the results for the three continuous distributions and the discrete one (Poisson). The Poisson distribution is used, for example, to simulate the arrival of (rare) external events, like a jump in the price of an instrument or an exogenic shock. Here is the code that generates it:

```
In [10]: sample_size = 500
         rn1 = npr.standard_normal(sample_size)  ❶
         rn2 = npr.normal(100, 20, sample_size)  ❷
         rn3 = npr.chisquare(df=0.5, size=sample_size)  ❸
```

```
rn4 = npr.poisson(lam=1.0, size=sample_size)  ❹
```

```
In [11]: fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2, ncols=2,
                                                      figsize=(10, 8))
         ax1.hist(rn1, bins=25)
         ax1.set_title('standard normal')
         ax1.set_ylabel('frequency')
         ax2.hist(rn2, bins=25)
         ax2.set_title('normal(100, 20)')
         ax3.hist(rn3, bins=25)
         ax3.set_title('chi square')
         ax3.set_ylabel('frequency')
         ax4.hist(rn4, bins=25)
         ax4.set_title('Poisson');
```

❶ Standard normally distributed random numbers.

❷ Normally distributed random numbers.

❸ Chi-square distributed random numbers.

❹ Poisson distributed numbers.

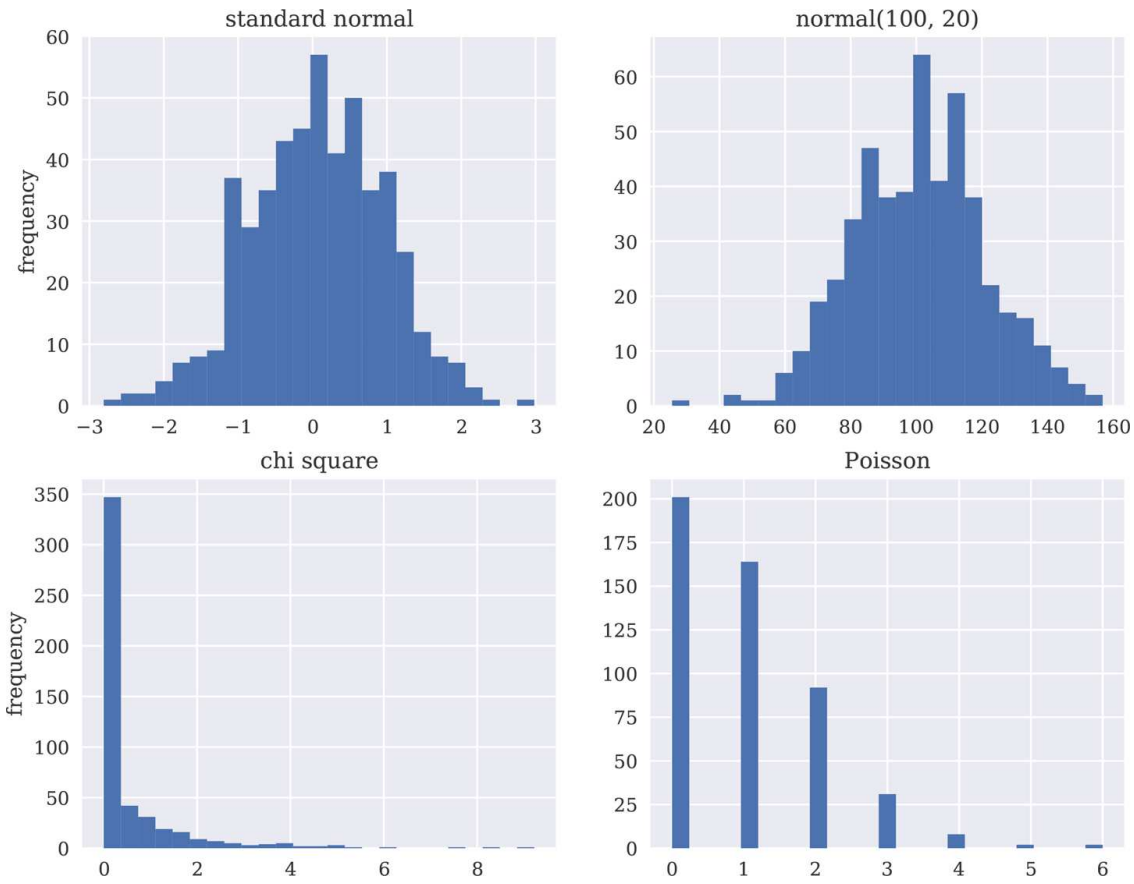

Figure 12-2. Histograms of random samples for different distributions

NUMPY AND RANDOM NUMBERS

This section shows that `NumPy` is a powerful (even indispensable) tool when generating pseudo-random numbers in Python. The creation of small or large `ndarray` objects with such numbers is not only convenient but also performant.

Simulation

Monte Carlo simulation (MCS) is among the most important numerical techniques in finance, if not *the* most important and widely used. This mainly stems from the fact that it is the most flexible numerical method when it comes to the evaluation of mathematical expressions (e.g., integrals), and specifically the valuation of financial derivatives. The flexibility comes at the cost of a relatively high computational burden, though, since often hundreds of thousands or even millions of complex computations have to be carried out to come up with a single value estimate.

Random Variables

Consider, for example, the Black-Scholes-Merton setup for option pricing. In their setup, the level of a stock index S_T at a future date T given a level S_0 as of today is given according to Equation 12-1.

Equation 12-1. Simulating future index level in Black-Scholes-Merton setup

$$S_T = S_0 \exp\left(\left(r - \frac{1}{2}\sigma^2\right)T + \sigma\sqrt{T}z\right)$$

The variables and parameters have the following meaning:

S_T
: Index level at date T

r
: Constant riskless short rate

σ

Constant volatility (= standard deviation of returns) of S

z

Standard normally distributed random variable

This financial model is parameterized and simulated as follows. The output of this simulation code is shown in Figure 12-3:

```
In [12]: S0 = 100  ❶
         r = 0.05  ❷
         sigma = 0.25  ❸
         T = 2.0  ❹
         I = 10000  ❺
         ST1 = S0 * np.exp((r - 0.5 * sigma ** 2) * T +
                 sigma * math.sqrt(T) * npr.standard_normal(I))  ❻

In [13]: plt.figure(figsize=(10, 6))
         plt.hist(ST1, bins=50)
         plt.xlabel('index level')
         plt.ylabel('frequency');
```

❶ The initial index level.

❷ The constant riskless short rate.

❸ The constant volatility factor.

❹ The horizon in year fractions.

❺ The number of simulations.

❻ The simulation itself via a vectorized expression; the discretization scheme makes use of the `npr.standard_normal()` function.

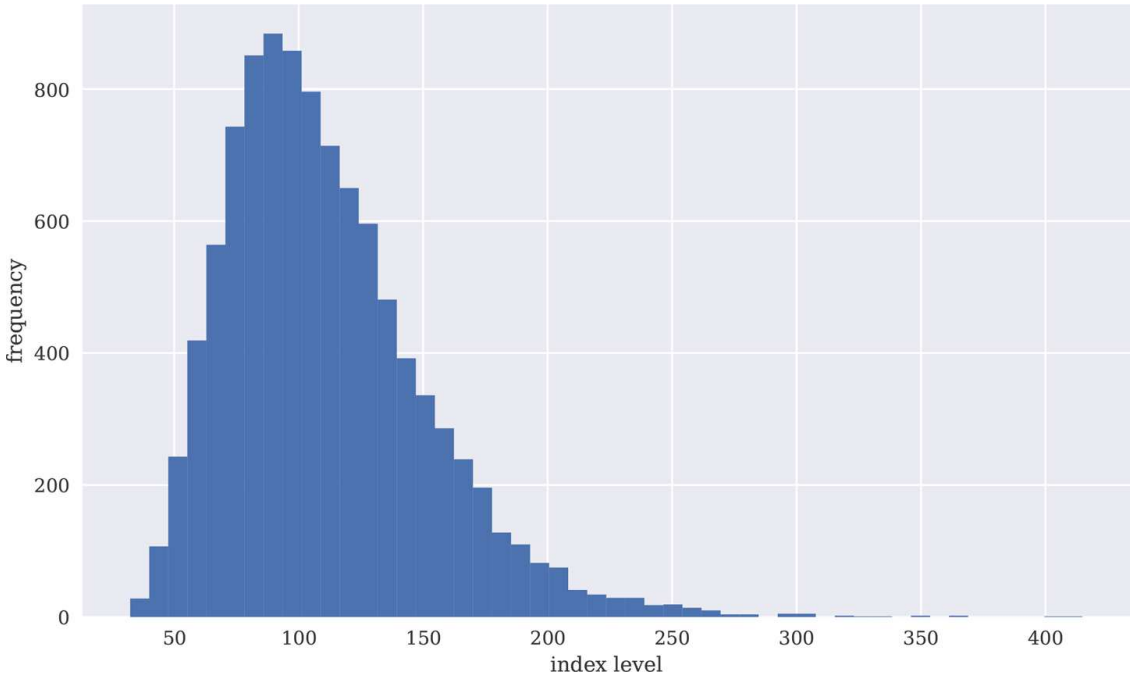

Figure 12-3. Statically simulated geometric Brownian motion (via npr.standard_normal())

Figure 12-3 suggests that the distribution of the random variable as defined in Equation 12-1 is *log-normal*. One could therefore also try to use the `npr.lognormal()` function to directly derive the values for the random variable. In that case, one has to provide the mean and the standard deviation to the function:

```
In [14]: ST2 = S0 * npr.lognormal((r - 0.5 * sigma ** 2) * T,
                                  sigma * math.sqrt(T), size=I)  ❶

In [15]: plt.figure(figsize=(10, 6))
         plt.hist(ST2, bins=50)
         plt.xlabel('index level')
         plt.ylabel('frequency');
```

❶ The simulation via a vectorized expression; the discretization scheme makes use of the `npr.lognormal()` function.

The result is shown in Figure 12-4.

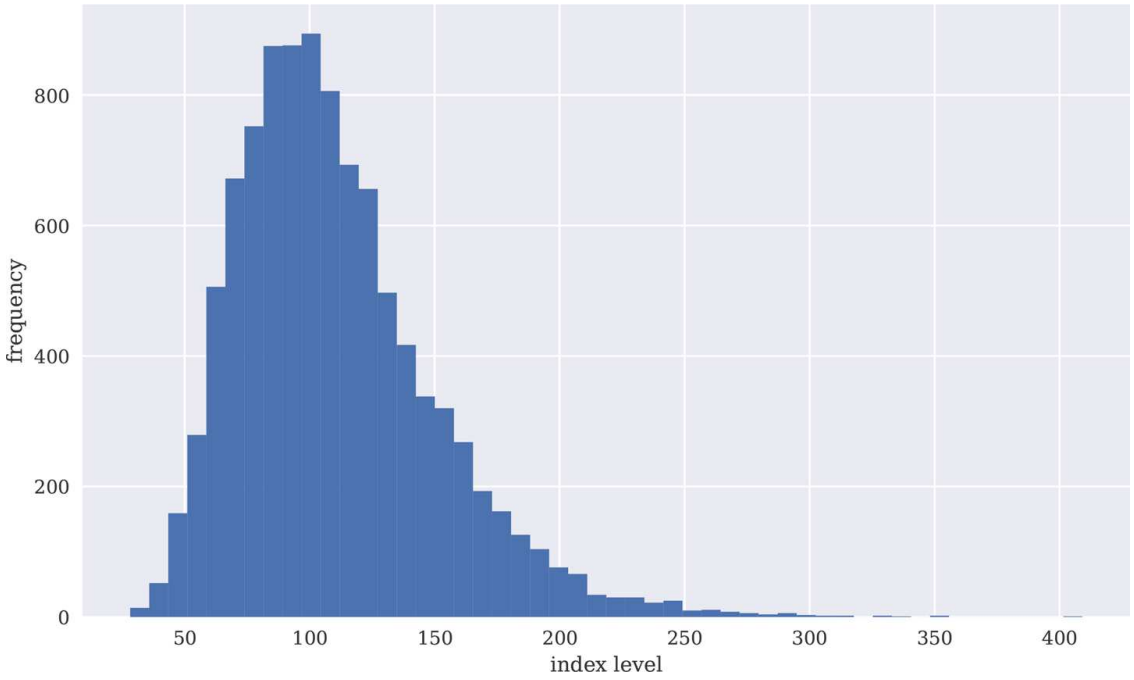

Figure 12-4. Statically simulated geometric Brownian motion (via npr.lognormal())

By visual inspection, Figures 12-3 and 12-4 indeed look pretty similar. This can be verified a bit more rigorously by comparing statistical moments of the resulting distributions. To compare the distributional characteristics of simulation results, the `scipy.stats` subpackage and the helper function `print_statistics()`, as defined here, prove useful:

```
In [16]: import scipy.stats as scs

In [17]: def print_statistics(a1, a2):
             ''' Prints selected statistics.

             Parameters
             ==========
             a1, a2: ndarray objects
                 results objects from simulation
             '''
             sta1 = scs.describe(a1)  ❶
             sta2 = scs.describe(a2)  ❶
             print('%14s %14s %14s' %
                 ('statistic', 'data set 1', 'data set 2'))
             print(45 * "-")
             print('%14s %14.3f %14.3f' % ('size', sta1[0], sta2[0]))
             print('%14s %14.3f %14.3f' % ('min', sta1[1][0], sta2[1][0]))
             print('%14s %14.3f %14.3f' % ('max', sta1[1][1], sta2[1][1]))
```

```
                print('%14s %14.3f %14.3f' % ('mean', sta1[2], sta2[2]))
                print('%14s %14.3f %14.3f' % ('std', np.sqrt(sta1[3]),
                                                  np.sqrt(sta2[3])))
                print('%14s %14.3f %14.3f' % ('skew', sta1[4], sta2[4]))
                print('%14s %14.3f %14.3f' % ('kurtosis', sta1[5], sta2[5]))

In [18]: print_statistics(ST1, ST2)
                 statistic     data set 1     data set 2
            ------------------------------------------------
                      size      10000.000      10000.000
                       min         32.327         28.230
                       max        414.825        409.110
                      mean        110.730        110.431
                       std         40.300         39.878
                      skew          1.122          1.115
                  kurtosis          2.438          2.217
```

❶ The `scs.describe()` function gives back important statistics for a data set.

Obviously, the statistics of both simulation results are quite similar. The differences are mainly due to what is called the *sampling error* in simulation. Another error can also be introduced when *discretely* simulating *continuous* stochastic processes—namely the *discretization error*, which plays no role here due to the static nature of the simulation approach.

Stochastic Processes

Roughly speaking, a *stochastic process* is a sequence of random variables. In that sense, one should expect something similar to a sequence of repeated simulations of a random variable when simulating a process. This is mainly true, apart from the fact that the draws are typically not independent but rather depend on the result(s) of the previous draw(s). In general, however, stochastic processes used in finance exhibit the *Markov property*, which mainly says that tomorrow's value of the process only depends on today's state of the process, and not any other more "historic" state or even the whole path history. The process then is also called *memoryless*.

Geometric Brownian motion

Consider now the Black-Scholes-Merton model in its dynamic form, as described by the stochastic differential equation (SDE) in Equation 12-2. Here, Z_t is a standard Brownian motion. The SDE is called a *geometric Brownian motion*. The values of S_t are log-normally distributed and the (marginal) returns $\frac{dS_t}{S_t}$ normally.

Equation 12-2. Stochastic differential equation in Black-Scholes-Merton setup

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

The SDE in Equation 12-2 can be discretized exactly by an Euler scheme. Such a scheme is presented in Equation 12-3, with Δt being the fixed discretization interval and z_t being a standard normally distributed random variable.

Equation 12-3. Simulating index levels dynamically in Black-Scholes-Merton setup

$$S_t = S_{t-\Delta t} \exp\left(\left(r - \frac{1}{2}\sigma^2\right)\Delta t + \sigma\sqrt{\Delta t} z_t\right)$$

As before, translation into Python and `NumPy` code is straightforward. The resulting end values for the index level are log-normally distributed again, as Figure 12-5 illustrates. The first four moments are also quite close to those resulting from the static simulation approach:

```
In [19]: I = 10000  ❶
         M = 50  ❷
         dt = T / M  ❸
         S = np.zeros((M + 1, I))  ❹
         S[0] = S0  ❺
         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(I))  ❻

In [20]: plt.figure(figsize=(10, 6))
         plt.hist(S[-1], bins=50)
```

```
plt.xlabel('index level')
plt.ylabel('frequency');
```

❶ The number of paths to be simulated.

❷ The number of time intervals for the discretization.

❸ The length of the time interval in year fractions.

❹ The two-dimensional `ndarray` object for the index levels.

❺ The initial values for the initial point in time $t = 0$.

❻ The simulation via semivectorized expression; the loop is over the points in time starting at $t = 1$ and ending at $t = T$.

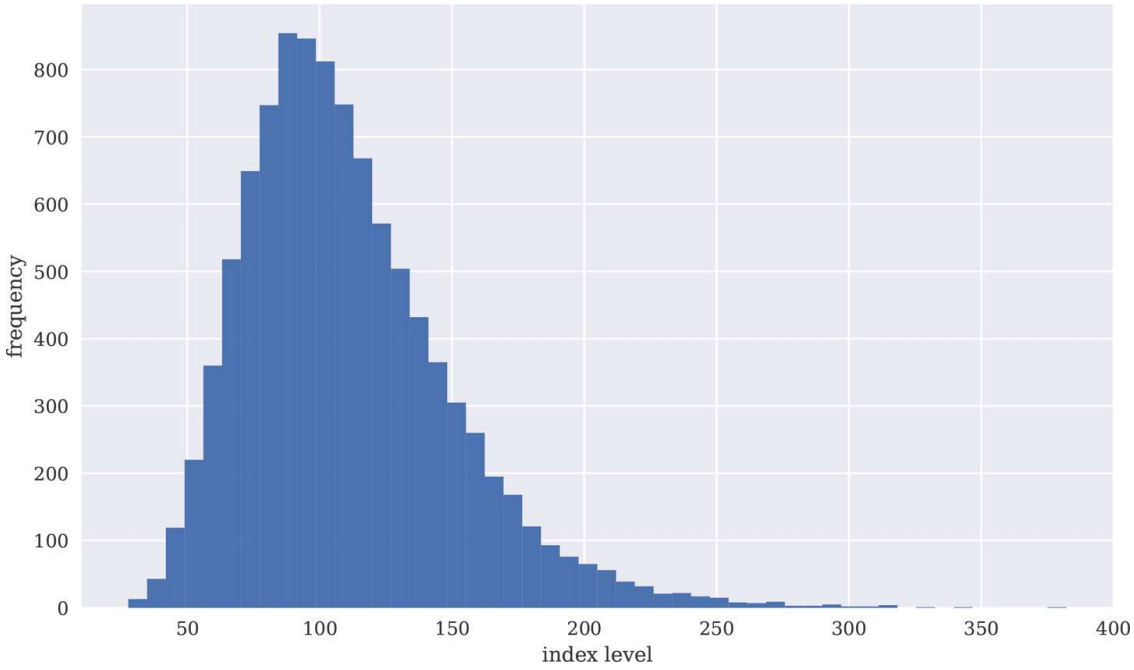

Figure 12-5. Dynamically simulated geometric Brownian motion at maturity

Following is a comparison of the statistics resulting from the dynamic simulation as well as from the static simulation. Figure 12-6 shows the first 10 simulated paths:

```
In [21]: print_statistics(S[-1], ST2)
             statistic      data set 1      data set 2
        ----------------------------------------------
                  size       10000.000       10000.000
```

min	27.746	28.230
max	382.096	409.110
mean	110.423	110.431
std	39.179	39.878
skew	1.069	1.115
kurtosis	2.028	2.217

```
In [22]: plt.figure(figsize=(10, 6))
         plt.plot(S[:, :10], lw=1.5)
         plt.xlabel('time')
         plt.ylabel('index level');
```

Figure 12-6. Dynamically simulated geometric Brownian motion paths

Using the dynamic simulation approach not only allows us to visualize paths as displayed in Figure 12-6, but also to value options with American/Bermudan exercise or options whose payoff is path-dependent. One gets the full dynamic picture over time, so to say.

Square-root diffusion

Another important class of financial processes is *mean-reverting processes*, which are used to model short rates or volatility processes, for example. A popular and widely used model is the *square-root diffusion*, as proposed by Cox, Ingersoll, and Ross (1985). Equation 12-4 provides the respective SDE.

Equation 12-4. Stochastic differential equation for square-root diffusion

$$dx_t = \kappa\left(\theta - x_t\right)dt + \sigma\sqrt{x_t}dZ_t$$

The variables and parameters have the following meaning:

x_t
: Process level at date t

κ
: Mean-reversion factor

θ
: Long-term mean of the process

σ
: Constant volatility parameter

Z_t
: Standard Brownian motion

It is well known that the values of x_t are chi-squared distributed. However, as stated before, many financial models can be discretized and approximated by using the normal distribution (i.e., a so-called Euler discretization scheme). While the Euler scheme is exact for the geometric Brownian motion, it is biased for the majority of other stochastic processes. Even if there is an exact scheme available—one for the square-root diffusion will be presented later—the use of an Euler scheme might be desirable for numerical and/or computational reasons. Defining $s = t - \Delta t$ and $x^+ \equiv \max\left(x, 0\right)$, Equation 12-5 presents such an Euler scheme. This particular one is generally called *full truncation* in the literature (see Hilpisch (2015) for more details and other schemes).

Equation 12-5. Euler discretization for square-root diffusion

$$\begin{aligned}\tilde{x}_t &= \tilde{x}_s + \kappa\left(\theta - \tilde{x}_s^+\right)\Delta t + \sigma\sqrt{\tilde{x}_s^+}\sqrt{\Delta t}z_t \\ x_t &= \tilde{x}_t^+\end{aligned}$$

The square-root diffusion has the convenient and realistic characteristic that the values of x_t remain strictly positive. When discretizing it by an Euler scheme, negative values cannot be excluded. That is the reason why one works always with the positive version of the originally simulated process. In the simulation code, one therefore needs two `ndarray` objects instead of only one. Figure 12-7 shows the result of the simulation graphically as a histogram:

```
In [23]: x0 = 0.05  ❶
         kappa = 3.0  ❷
         theta = 0.02  ❸
         sigma = 0.1  ❹
         I = 10000
         M = 50
         dt = T / M

In [24]: def srd_euler():
             xh = np.zeros((M + 1, I))
             x = np.zeros_like(xh)
             xh[0] = x0
             x[0] = x0
             for t in range(1, M + 1):
                 xh[t] = (xh[t - 1] +
                          kappa * (theta - np.maximum(xh[t - 1], 0)) * dt +
                          sigma * np.sqrt(np.maximum(xh[t - 1], 0)) *
                          math.sqrt(dt) * npr.standard_normal(I))  ❺
             x = np.maximum(xh, 0)
             return x
         x1 = srd_euler()

In [25]: plt.figure(figsize=(10, 6))
         plt.hist(x1[-1], bins=50)
         plt.xlabel('value')
         plt.ylabel('frequency');
```

❶ The initial value (e.g., for a short rate).

❷ The mean reversion factor.

❸ The long-term mean value.

❹ The volatility factor.

❺ The simulation based on an Euler scheme.

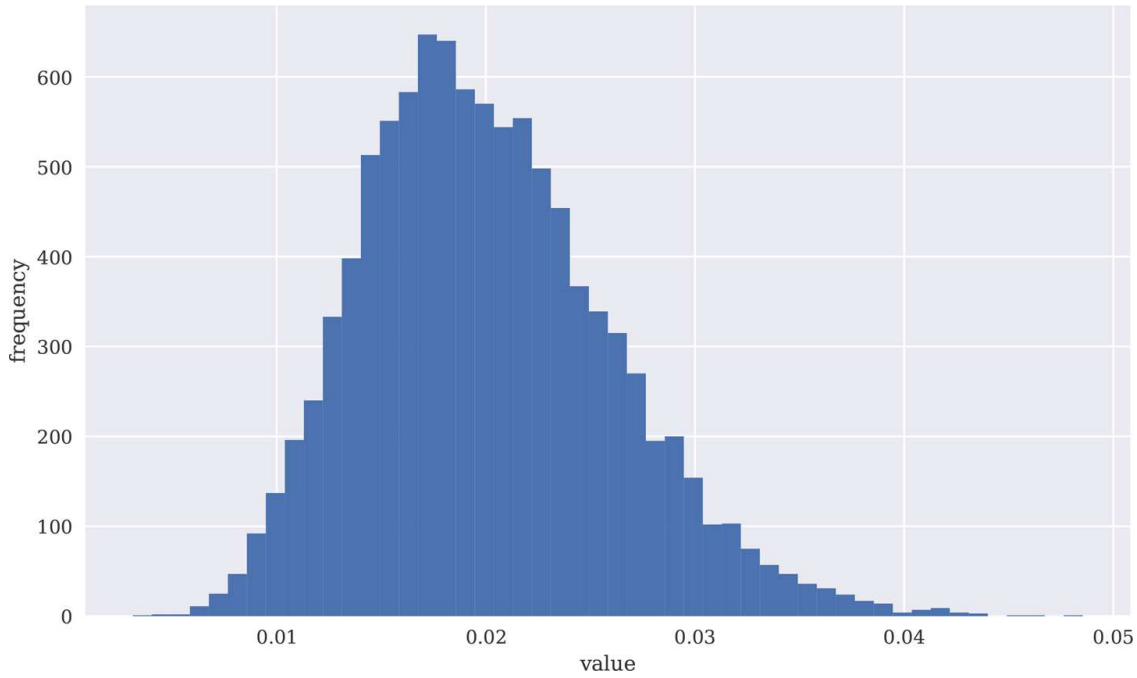

Figure 12-7. Dynamically simulated square-root diffusion at maturity (Euler scheme)

Figure 12-8 then shows the first 10 simulated paths, illustrating the resulting negative average drift (due to $x_0 > \theta$) and the convergence to $\theta = 0.02$:

```
In [26]: plt.figure(figsize=(10, 6))
         plt.plot(x1[:, :10], lw=1.5)
         plt.xlabel('time')
         plt.ylabel('index level');
```

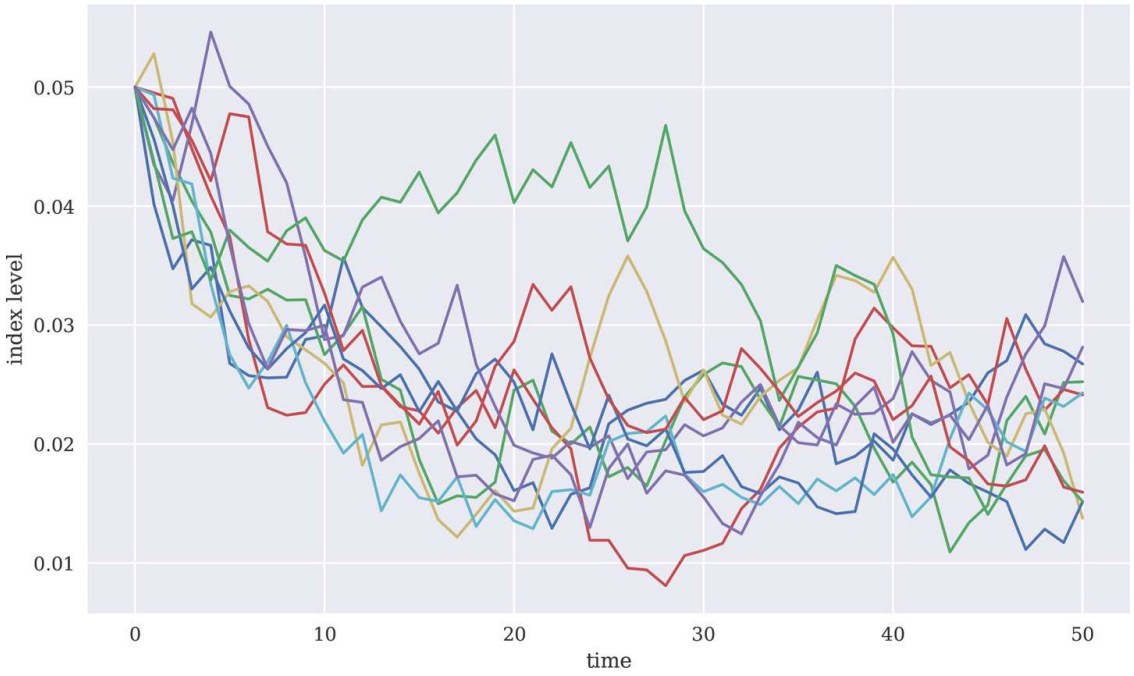

Figure 12-8. Dynamically simulated square-root diffusion paths (Euler scheme)

Equation 12-6 presents the exact discretization scheme for the square-root diffusion based on the noncentral chi-square distribution χ'^2_d with

$$df = \frac{4\theta\kappa}{\sigma^2}$$

degrees of freedom and noncentrality parameter

$$nc = \frac{4\kappa e^{-\kappa\Delta t}}{\sigma^2\left(1-e^{-\kappa\Delta t}\right)} x_s$$

Equation 12-6. Exact discretization for square-root diffusion

$$x_t = \frac{\sigma^2\left(1-e^{-\kappa\Delta t}\right)}{4\kappa}\chi'^2_d\left(\frac{4\kappa e^{-\kappa\Delta t}}{\sigma^2\left(1-e^{-\kappa\Delta t}\right)} x_s\right)$$

The Python implementation of this discretization scheme is a bit more involved but still quite concise. Figure 12-9 shows the output at maturity of the simulation with the exact scheme as a histogram:

```
In [27]: def srd_exact():
             x = np.zeros((M + 1, I))
             x[0] = x0
             for t in range(1, M + 1):
                 df = 4 * theta * kappa / sigma ** 2  ❶
                 c = (sigma ** 2 * (1 - np.exp(-kappa * dt))) / (4 * kappa)  ❶
```

```
                nc = np.exp(-kappa * dt) / c * x[t - 1]  ❶
                x[t] = c * npr.noncentral_chisquare(df, nc, size=I)  ❶
            return x
        x2 = srd_exact()
```

```
In [28]: plt.figure(figsize=(10, 6))
         plt.hist(x2[-1], bins=50)
         plt.xlabel('value')
         plt.ylabel('frequency');
```

❶ Exact discretization scheme, making use of `npr.noncentral_chisquare()`.

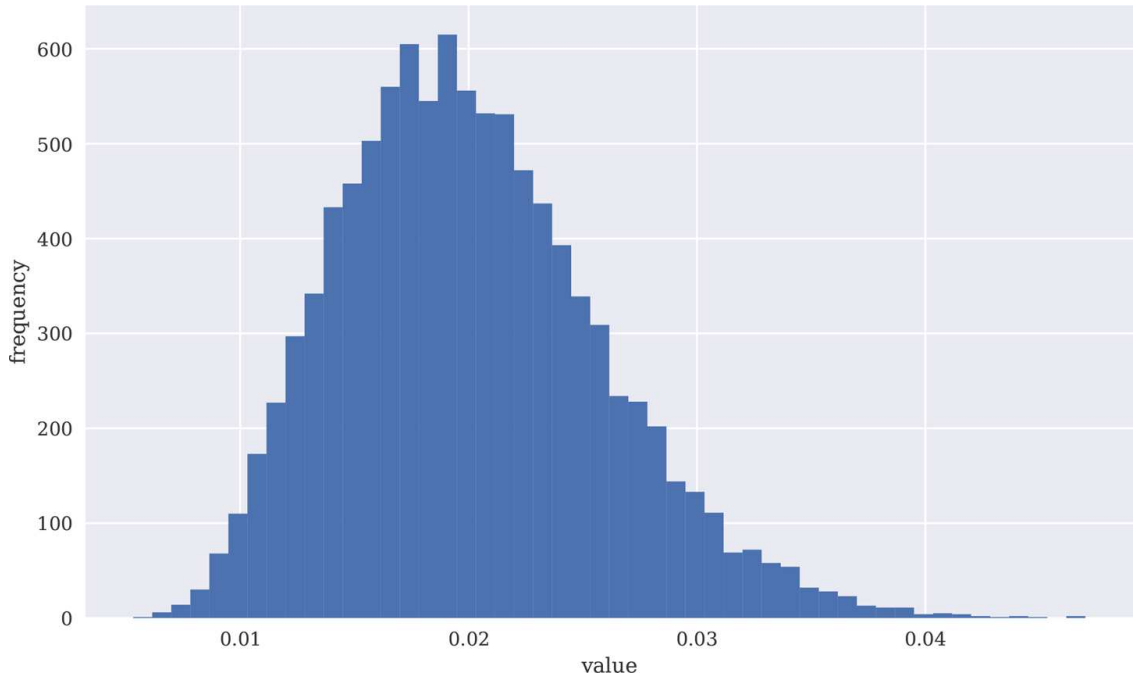

Figure 12-9. Dynamically simulated square-root diffusion at maturity (exact scheme)

Figure 12-10 presents as before the first 10 simulated paths, again displaying the negative average drift and the convergence to θ:

```
In [29]: plt.figure(figsize=(10, 6))
         plt.plot(x2[:, :10], lw=1.5)
         plt.xlabel('time')
         plt.ylabel('index level');
```

Figure 12-10. Dynamically simulated square-root diffusion paths (exact scheme)

Comparing the main statistics from the different approaches reveals that the biased Euler scheme indeed performs quite well when it comes to the desired statistical properties:

```
In [30]: print_statistics(x1[-1], x2[-1])
             statistic     data set 1     data set 2
         ----------------------------------------------
                  size      10000.000      10000.000
                   min          0.003          0.005
                   max          0.049          0.047
                  mean          0.020          0.020
                   std          0.006          0.006
                  skew          0.529          0.532
              kurtosis          0.289          0.273

In [31]: I = 250000
         %time x1 = srd_euler()
         CPU times: user 1.62 s, sys: 184 ms, total: 1.81 s
         Wall time: 1.08 s

In [32]: %time x2 = srd_exact()
         CPU times: user 3.29 s, sys: 39.8 ms, total: 3.33 s
         Wall time: 1.98 s

In [33]: print_statistics(x1[-1], x2[-1])
```

```
x1 = 0.0; x2 = 0.0
      statistic      data set 1      data set 2
-------------------------------------------------
           size      250000.000      250000.000
            min           0.002           0.003
            max           0.071           0.055
           mean           0.020           0.020
            std           0.006           0.006
           skew           0.563           0.579
       kurtosis           0.492           0.520
```

However, a major difference can be observed in terms of execution speed, since sampling from the noncentral chi-square distribution is more computationally demanding than from the standard normal distribution. The exact scheme takes roughly twice as much time for virtually the same results as with the Euler scheme.

Stochastic volatility

One of the major simplifying assumptions of the Black-Scholes-Merton model is the *constant* volatility. However, volatility in general is neither constant nor deterministic—it is *stochastic*. Therefore, a major advancement with regard to financial modeling was achieved in the early 1990s with the introduction of so-called *stochastic volatility models*. One of the most popular models that fall into that category is that of Heston (1993), which is presented in Equation 12-7.

Equation 12-7. Stochastic differential equations for Heston stochastic volatility model

$$\begin{aligned} dS_t &= rS_t dt + \sqrt{v_t} S_t dZ_t^1 \\ dv_t &= \kappa_v (\theta_v - v_t) dt + \sigma_v \sqrt{v_t} dZ_t^2 \\ dZ_t^1 dZ_t^2 &= \rho \end{aligned}$$

The meaning of the variables and parameters can now be inferred easily from the discussion of the geometric Brownian motion and the square-root diffusion. The parameter ρ represents the instantaneous correlation between the two standard Brownian motions Z_t^1, Z_t^2. This allows us to account for a stylized fact called the *leverage effect*, which in essence

states that volatility goes up in times of stress (declining markets) and goes down in times of a bull market (rising markets).

Consider the following parameterization of the model. To account for the correlation between the two stochastic processes, one needs to determine the Cholesky decomposition of the correlation matrix:

```
In [34]: S0 = 100.
         r = 0.05
         v0 = 0.1  ❶
         kappa = 3.0
         theta = 0.25
         sigma = 0.1
         rho = 0.6  ❷
         T = 1.0

In [35]: corr_mat = np.zeros((2, 2))
         corr_mat[0, :] = [1.0, rho]
         corr_mat[1, :] = [rho, 1.0]
         cho_mat = np.linalg.cholesky(corr_mat)  ❸

In [36]: cho_mat  ❸
Out[36]: array([[1. , 0. ],
                [0.6, 0.8]])
```

❶ Initial (instantaneous) volatility value.

❷ Fixed correlation between the two Brownian motions.

❸ Cholesky decomposition and resulting matrix.

Before the start of the simulation of the stochastic processes the whole set of random numbers for both processes is generated, looking to use set 0 for the index process and set 1 for the volatility process. For the volatility process modeled by a square-root diffusion, the Euler scheme is chosen, taking into account the correlation via the Cholesky matrix:

```
In [37]: M = 50
         I = 10000
         dt = T / M
```

```
In [38]: ran_num = npr.standard_normal((2, M + 1, I))  ❶

In [39]: v = np.zeros_like(ran_num[0])
         vh = np.zeros_like(v)

In [40]: v[0] = v0
         vh[0] = v0

In [41]: for t in range(1, M + 1):
             ran = np.dot(cho_mat, ran_num[:, t, :])  ❷
             vh[t] = (vh[t - 1] +
                      kappa * (theta - np.maximum(vh[t - 1], 0)) * dt +
                      sigma * np.sqrt(np.maximum(vh[t - 1], 0)) *
                      math.sqrt(dt) * ran[1])  ❸

In [42]: v = np.maximum(vh, 0)
```

❶ Generates the three-dimensional random number data set.

❷ Picks out the relevant random number subset and transforms it via the Cholesky matrix.

❸ Simulates the paths based on an Euler scheme.

The simulation of the index level process also takes into account the correlation and uses the (in this case) exact Euler scheme for the geometric Brownian motion. Figure 12-11 shows the simulation results at maturity as a histogram for both the index level process and the volatility process:

```
In [43]: S = np.zeros_like(ran_num[0])
         S[0] = S0
         for t in range(1, M + 1):
             ran = np.dot(cho_mat, ran_num[:, t, :])
             S[t] = S[t - 1] * np.exp((r - 0.5 * v[t]) * dt +
                                      np.sqrt(v[t]) * ran[0] * np.sqrt(dt))

In [44]: fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 6))
         ax1.hist(S[-1], bins=50)
         ax1.set_xlabel('index level')
         ax1.set_ylabel('frequency')
```

```
ax2.hist(v[-1], bins=50)
ax2.set_xlabel('volatility');
```

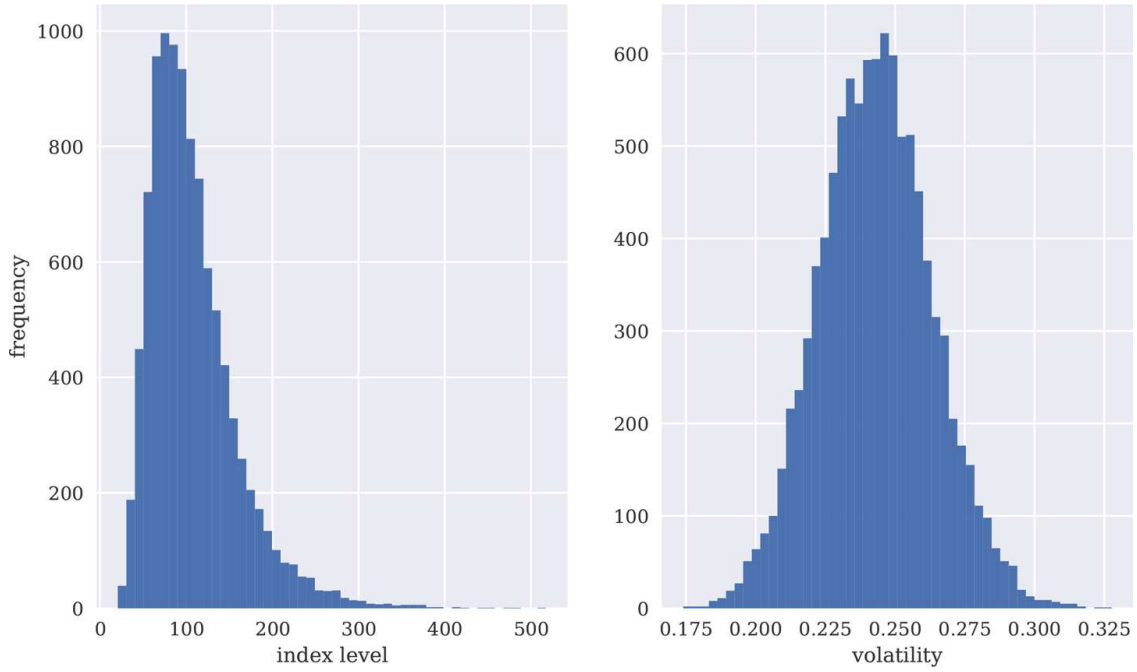

Figure 12-11. Dynamically simulated stochastic volatility process at maturity

This illustrates another advantage of working with the Euler scheme for the square-root diffusion: *correlation is easily and consistently accounted for* since one only draws standard normally distributed random numbers. There is no simple way of achieving the same with a mixed approach (i.e., using Euler for the index and the noncentral chi-square-based exact approach for the volatility process).

An inspection of the first 10 simulated paths of each process (see Figure 12-12) shows that the volatility process is drifting positively on average and that it, as expected, converges to $\theta = 0.25$:

```
In [45]: print_statistics(S[-1], v[-1])
             statistic      data set 1      data set 2
        ----------------------------------------------
                  size       10000.000       10000.000
                   min          20.556           0.174
                   max         517.798           0.328
                  mean         107.843           0.243
                   std          51.341           0.020
                  skew           1.577           0.124
              kurtosis           4.306           0.048
```

```
In [46]: fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True,
                                        figsize=(10, 6))
         ax1.plot(S[:, :10], lw=1.5)
         ax1.set_ylabel('index level')
         ax2.plot(v[:, :10], lw=1.5)
         ax2.set_xlabel('time')
         ax2.set_ylabel('volatility');
```

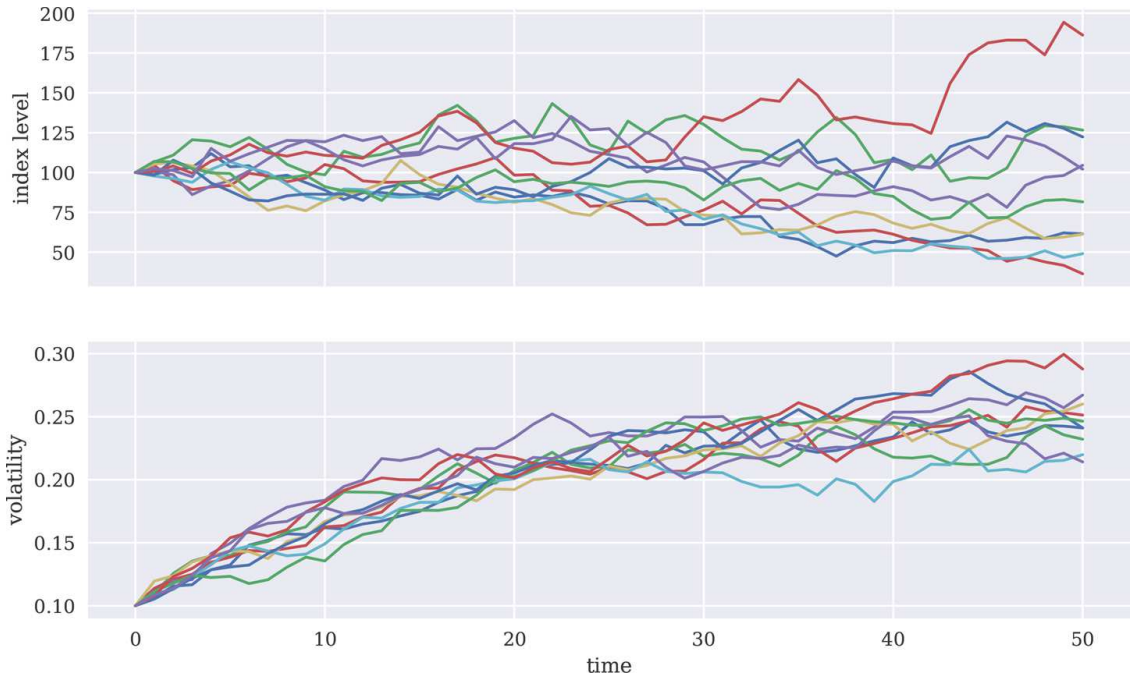

Figure 12-12. Dynamically simulated stochastic volatility process paths

Having a brief look at the statistics for the maturity date for both data sets reveals a pretty high maximum value for the index level process. In fact, this is much higher than a geometric Brownian motion with constant volatility could ever climb, *ceteris paribus*.

Jump diffusion

Stochastic volatility and the leverage effect are stylized (empirical) facts found in a number of markets. Another important stylized fact is the existence of *jumps* in asset prices and, for example, volatility. In 1976, Merton published his jump diffusion model, enhancing the Black-Scholes-Merton setup through a model component generating jumps with log-normal distribution. The risk-neutral SDE is presented in Equation 12-8.

Equation 12-8. Stochastic differential equation for Merton jump diffusion model

$$dS_t = (r - r_j)S_t dt + \sigma S_t dZ_t + J_t S_t dN_t$$

For completeness, here is an overview of the variables' and parameters' meaning:

S_t

: Index level at date t

r

: Constant riskless short rate

$r_J \equiv \lambda \cdot \left(e^{\mu_J + \delta^2/2} - 1 \right)$

: Drift correction for jump to maintain risk neutrality

σ

: Constant volatility of S

Z_t

: Standard Brownian motion

J_t

: Jump at date t with distribution ...

- ... $\log(1 + J_t) \approx \mathbf{N}\left(\log(1 + \mu_J) - \frac{\delta^2}{2}, \delta^2 \right)$ with ...
- ... $\mathbf{N}$ as the cumulative distribution function of a standard normal random variable

N_t

: Poisson process with intensity λ

Equation 12-9 presents an Euler discretization for the jump diffusion where the z_t^n are standard normally distributed and the y_t are Poisson distributed with intensity λ.

Equation 12-9. Euler discretization for Merton jump diffusion model

$$S_t = S_{t-\Delta t}\left(e^{(r-r_J-\sigma^2/2)\Delta t+\sigma\sqrt{\Delta t}z_t^1} + \left(e^{\mu_J+\delta z_t^2}-1\right)y_t\right)$$

Given the discretization scheme, consider the following numerical parameterization:

```
In [47]: S0 = 100.
         r = 0.05
         sigma = 0.2
         lamb = 0.75  ❶
         mu = -0.6  ❷
         delta = 0.25  ❸
         rj = lamb * (math.exp(mu + 0.5 * delta ** 2) - 1)  ❹

In [48]: T = 1.0
         M = 50
         I = 10000
         dt = T / M
```

❶ The jump intensity.

❷ The mean jump size.

❸ The jump volatility.

❹ The drift correction.

This time, three sets of random numbers are needed. Notice in Figure 12-13 the second peak (bimodal frequency distribution), which is due to the jumps:

```
In [49]: S = np.zeros((M + 1, I))
         S[0] = S0
         sn1 = npr.standard_normal((M + 1, I))  ❶
         sn2 = npr.standard_normal((M + 1, I))  ❶
         poi = npr.poisson(lamb * dt, (M + 1, I))  ❷
         for t in range(1, M + 1, 1):
             S[t] = S[t - 1] * (np.exp((r - rj - 0.5 * sigma ** 2) * dt +
                                       sigma * math.sqrt(dt) * sn1[t]) +
                                       (np.exp(mu + delta * sn2[t]) - 1) *
```

```
                                    poi[t])  ❸
            S[t] = np.maximum(S[t], 0)
```

```
In [50]: plt.figure(figsize=(10, 6))
         plt.hist(S[-1], bins=50)
         plt.xlabel('value')
         plt.ylabel('frequency');
```

❶ Standard normally distributed random numbers.

❷ Poisson distributed random numbers.

❸ Simulation based on the exact Euler scheme.

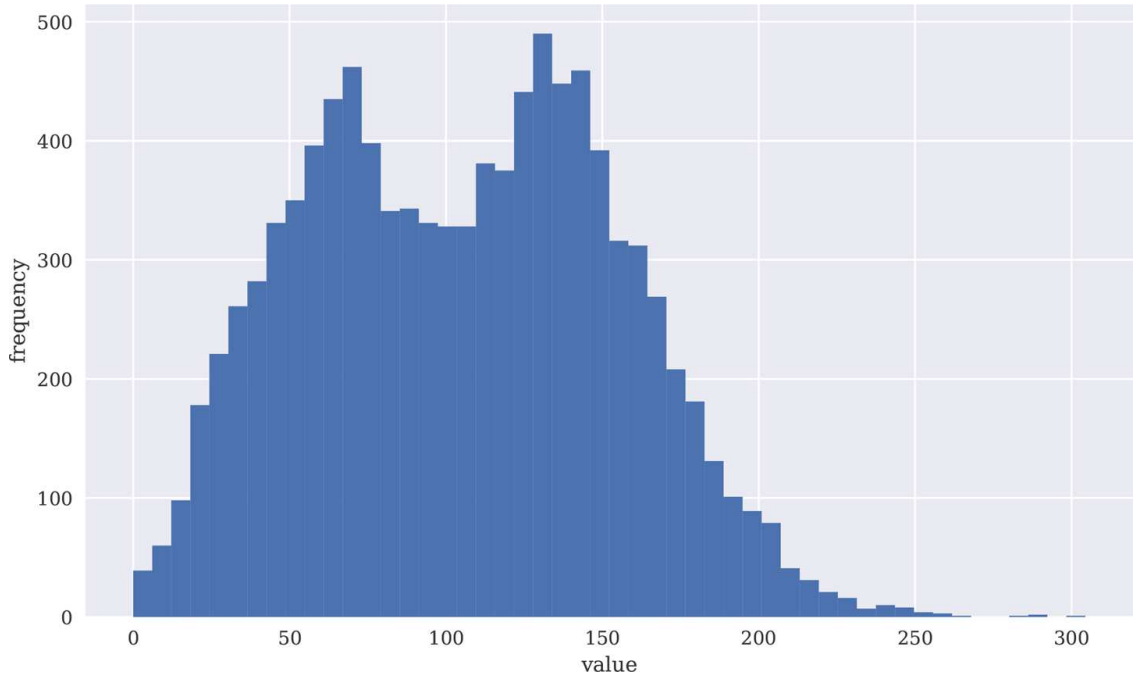

Figure 12-13. Dynamically simulated jump diffusion process at maturity

The negative jumps can also be spotted in the first 10 simulated index level paths, as presented in Figure 12-14:

```
In [51]: plt.figure(figsize=(10, 6))
         plt.plot(S[:, :10], lw=1.5)
         plt.xlabel('time')
         plt.ylabel('index level');
```

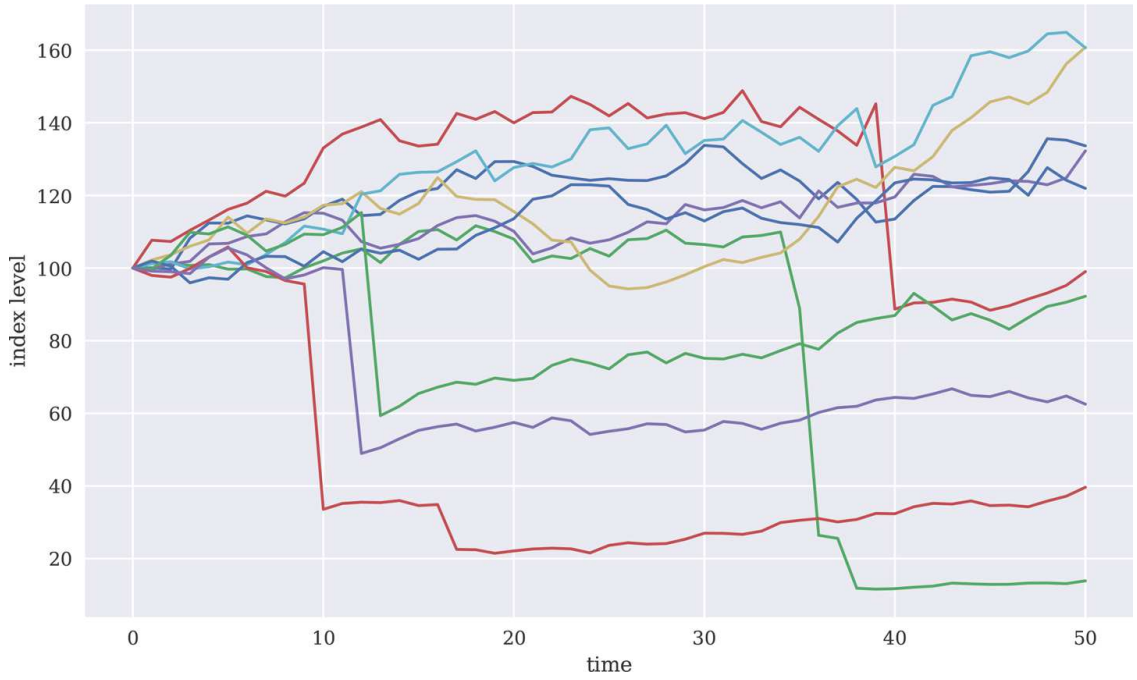

Figure 12-14. Dynamically simulated jump diffusion process paths

Variance Reduction

Because the Python functions used so far generate *pseudo-random* numbers and due to the varying sizes of the samples drawn, the resulting sets of numbers might not exhibit statistics close enough to the expected or desired ones. For example, one would expect a set of standard normally distributed random numbers to show a mean of 0 and a standard deviation of 1. Let us check what statistics different sets of random numbers exhibit. To achieve a realistic comparison, the seed value for the random number generator is fixed:

```
In [52]: print('%15s %15s' % ('Mean', 'Std. Deviation'))
         print(31 * '-')
         for i in range(1, 31, 2):
             npr.seed(100)
             sn = npr.standard_normal(i ** 2 * 10000)
             print('%15.12f %15.12f' % (sn.mean(), sn.std()))
                    Mean   Std. Deviation
         -------------------------------
          0.001150944833   1.006296354600
          0.002841204001   0.995987967146
          0.001998082016   0.997701714233
          0.001322322067   0.997771186968
          0.000592711311   0.998388962646
```

```
-0.000339730751  0.998399891450
-0.000228109010  0.998657429396
 0.000295768719  0.998877333340
 0.000257107789  0.999284894532
-0.000357870642  0.999456401088
-0.000528443742  0.999617831131
-0.000300171536  0.999445228838
-0.000162924037  0.999516059328
 0.000135778889  0.999611052522
 0.000182006048  0.999619405229
```

```
In [53]: i ** 2 * 10000
Out[53]: 8410000
```

The results show that the statistics "somehow" get better the larger the number of draws becomes.[2] But they still do not match the desired ones, even in our largest sample with more than 8,000,000 random numbers.

Fortunately, there are easy-to-implement, generic variance reduction techniques available to improve the matching of the first two moments of the (standard) normal distribution. The first technique is to use *antithetic variates*. This approach simply draws only half the desired number of random draws, and adds the same set of random numbers with the opposite sign afterward.[3] For example, if the random number generator (i.e., the respective Python function) draws 0.5, then another number with value –0.5 is added to the set. By construction, the mean value of such a data set must equal zero.

With `NumPy` this is concisely implemented by using the function `np.concatenate()`. The following repeats the exercise from before, this time using antithetic variates:

```
In [54]: sn = npr.standard_normal(int(10000 / 2))
         sn = np.concatenate((sn, -sn))  ❶

In [55]: np.shape(sn)  ❷
Out[55]: (10000,)

In [56]: sn.mean()  ❸
Out[56]: 2.842170943040401e-18
```

```
In [57]: print('%15s %15s' % ('Mean', 'Std. Deviation'))
         print(31 * "-")
         for i in range(1, 31, 2):
             npr.seed(1000)
             sn = npr.standard_normal(i ** 2 * int(10000 / 2))
             sn = np.concatenate((sn, -sn))
             print("%15.12f %15.12f" % (sn.mean(), sn.std()))
                    Mean   Std. Deviation
         ------------------------------
          0.000000000000  1.009653753942
         -0.000000000000  1.000413716783
          0.000000000000  1.002925061201
         -0.000000000000  1.000755212673
          0.000000000000  1.001636910076
         -0.000000000000  1.000726758438
         -0.000000000000  1.001621265149
          0.000000000000  1.001203722778
         -0.000000000000  1.000556669784
         -0.000000000000  1.000113464185
         -0.000000000000  0.999435175324
         -0.000000000000  0.999356961431
         -0.000000000000  0.999641436845
         -0.000000000000  0.999642768905
         -0.000000000000  0.999638303451
```

❶ This concatenates the two `ndarray` objects ...

❷ ... to arrive at the desired number of random numbers.

❸ The resulting mean value is zero (within standard floating-point arithmetic errors).

As immediately noticed, this approach corrects the first moment perfectly—which should not come as a surprise due to the very construction of the data set. However, this approach does not have any influence on the second moment, the standard deviation. Using another variance reduction technique, called *moment matching*, helps correct in one step both the first and second moments:

```
In [58]: sn = npr.standard_normal(10000)

In [59]: sn.mean()
Out[59]: -0.001165998295162494

In [60]: sn.std()
Out[60]: 0.991255920204605

In [61]: sn_new = (sn - sn.mean()) / sn.std()  ❶

In [62]: sn_new.mean()
Out[62]: -2.3803181647963357e-17

In [63]: sn_new.std()
Out[63]: 0.9999999999999999
```

❶ Corrects both the first and second moment in a single step.

By subtracting the mean from every single random number and dividing every single number by the standard deviation, this technique ensures that the set of random numbers matches the desired first and second moments of the standard normal distribution (almost) perfectly.

The following function utilizes the insight with regard to variance reduction techniques and generates standard normal random numbers for process simulation using either two, one, or no variance reduction technique(s):

```
In [64]: def gen_sn(M, I, anti_paths=True, mo_match=True):
             ''' Function to generate random numbers for simulation.

             Parameters
             ==========
             M: int
                 number of time intervals for discretization
             I: int
                 number of paths to be simulated
             anti_paths: boolean
                 use of antithetic variates
             mo_math: boolean
                 use of moment matching
```

```
    '''
    if anti_paths is True:
        sn = npr.standard_normal((M + 1, int(I / 2)))
        sn = np.concatenate((sn, -sn), axis=1)
    else:
        sn = npr.standard_normal((M + 1, I))
    if mo_match is True:
        sn = (sn - sn.mean()) / sn.std()
    return sn
```

VECTORIZATION AND SIMULATION

Vectorization with `NumPy` is a natural, concise, and efficient approach to implementing Monte Carlo simulation algorithms in Python. However, using `NumPy` vectorization comes with a larger memory footprint in general. For alternatives that might be equally fast, see Chapter 10.

Valuation

One of the most important applications of Monte Carlo simulation is the *valuation of contingent claims* (options, derivatives, hybrid instruments, etc.). Simply stated, in a risk-neutral world, the value of a contingent claim is the discounted expected payoff under the risk-neutral (martingale) measure. This is the probability measure that makes all risk factors (stocks, indices, etc.) drift at the riskless short rate, making the discounted processes martingales. According to the Fundamental Theorem of Asset Pricing, the existence of such a probability measure is equivalent to the absence of arbitrage.

A financial option embodies the right to buy (*call option*) or sell (*put option*) a specified financial instrument at a given maturity date (*European option*), or over a specified period of time (*American option*), at a given price (*strike price*). Let us first consider the simpler case of European options in terms of valuation.

European Options

The payoff of a European call option on an index at maturity is given by $h(S_T) \equiv \max(S_T - K, 0)$, where S_T is the index level at maturity date T and K is the strike price. Given a, or in complete markets *the*, risk-neutral measure for the relevant stochastic process (e.g., geometric Brownian motion), the price of such an option is given by the formula in Equation 12-10.

Equation 12-10. Pricing by risk-neutral expectation

$$C_0 = e^{-rT} \mathbf{E}_0^Q (h(S_T)) = e^{-rT} \int_0^\infty h(s) q(s) ds$$

Chapter 11 sketches how to numerically evaluate an integral by Monte Carlo simulation. This approach is used in the following and applied to Equation 12-10. Equation 12-11 provides the respective Monte Carlo estimator for the European option, where $\tilde{S}_T^i$ is the Tth simulated index level at maturity.

Equation 12-11. Risk-neutral Monte Carlo estimator

$$\widetilde{C_0} = e^{-rT} \frac{1}{I} \sum_{i=1}^{I} h\left(\tilde{S}_T^i\right)$$

Consider now the following parameterization for the geometric Brownian motion and the valuation function `gbm_mcs_stat()`, taking as a parameter only the strike price. Here, only the index level at maturity is simulated. As a reference, consider the case with a strike price of K = 105:

```
In [65]: S0 = 100.
         r = 0.05
         sigma = 0.25
         T = 1.0
         I = 50000

In [66]: def gbm_mcs_stat(K):
             ''' Valuation of European call option in Black-Scholes-Merton
             by Monte Carlo simulation (of index level at maturity)

             Parameters
```

```
                ==========
                K: float
                    (positive) strike price of the option

                Returns
                =======
                C0: float
                    estimated present value of European call option
                '''
                sn = gen_sn(1, I)
                # simulate index level at maturity
                ST = S0 * np.exp((r - 0.5 * sigma ** 2) * T
                             + sigma * math.sqrt(T) * sn[1])
                # calculate payoff at maturity
                hT = np.maximum(ST - K, 0)
                # calculate MCS estimator
                C0 = math.exp(-r * T) * np.mean(hT)
                return C0

In [67]: gbm_mcs_stat(K=105.)  ❶
Out[67]: 10.044221852841922
```

❶ The Monte Carlo estimator value for the European call option.

Next, consider the dynamic simulation approach and allow for European put options in addition to the call option. The function `gbm_mcs_dyna()` implements the algorithm. The code also compares option price estimates for a call and a put stroke at the same level:

```
In [68]: M = 50  ❶

In [69]: def gbm_mcs_dyna(K, option='call'):
             ''' Valuation of European options in Black-Scholes-Merton
             by Monte Carlo simulation (of index level paths)

             Parameters
             ==========
             K: float
                 (positive) strike price of the option
             option : string
                 type of the option to be valued ('call', 'put')
```

```
    Returns
    =======
    C0: float
        estimated present value of European call option
    '''
    dt = T / M
    # simulation of index level paths
    S = np.zeros((M + 1, I))
    S[0] = S0
    sn = gen_sn(M, I)
    for t in range(1, M + 1):
        S[t] = S[t - 1] * np.exp((r - 0.5 * sigma ** 2) * dt
                + sigma * math.sqrt(dt) * sn[t])
    # case-based calculation of payoff
    if option == 'call':
        hT = np.maximum(S[-1] - K, 0)
    else:
        hT = np.maximum(K - S[-1], 0)
    # calculation of MCS estimator
    C0 = math.exp(-r * T) * np.mean(hT)
    return C0
```

```
In [70]: gbm_mcs_dyna(K=110., option='call')  ❷
Out[70]: 7.950008525028434

In [71]: gbm_mcs_dyna(K=110., option='put')  ❸
Out[71]: 12.629934942682004
```

❶ The number of time intervals for the discretization.

❷ The Monte Carlo estimator value for the European *call* option.

❸ The Monte Carlo estimator value for the European *put* option.

The question is how well these simulation-based valuation approaches perform relative to the benchmark value from the Black-Scholes-Merton valuation formula. To find out, the following code generates respective option values/estimates for a range of strike prices, using the analytical option pricing formula for European calls found in the module `bsm_functions.py` (see “Python Script”).

First, we compare the results from the static simulation approach with precise analytical values:

```
In [72]: from bsm_functions import bsm_call_value

In [73]: stat_res = []  ❶
         dyna_res = []  ❶
         anal_res = []  ❶
         k_list = np.arange(80., 120.1, 5.)  ❷
         np.random.seed(100)

In [74]: for K in k_list:
             stat_res.append(gbm_mcs_stat(K))  ❸
             dyna_res.append(gbm_mcs_dyna(K))  ❸
             anal_res.append(bsm_call_value(S0, K, T, r, sigma))  ❸

In [75]: stat_res = np.array(stat_res)  ❹
         dyna_res = np.array(dyna_res)  ❹
         anal_res = np.array(anal_res)  ❹
```

❶ Instantiates empty `list` objects to collect the results.

❷ Creates an `ndarray` object containing the range of strike prices.

❸ Simulates/calculates and collects the option values for all strike prices.

❹ Transforms the `list` objects to `ndarray` objects.

Figure 12-15 shows the results. All valuation differences are smaller than 1% absolutely. There are both negative and positive value differences:

```
In [76]: plt.figure(figsize=(10, 6))
         fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True, figsize=(10, 6))
         ax1.plot(k_list, anal_res, 'b', label='analytical')
         ax1.plot(k_list, stat_res, 'ro', label='static')
         ax1.set_ylabel('European call option value')
         ax1.legend(loc=0)
         ax1.set_ylim(bottom=0)
         wi = 1.0
```

```
ax2.bar(k_list - wi / 2, (anal_res - stat_res) / anal_res * 100, wi)
ax2.set_xlabel('strike')
ax2.set_ylabel('difference in %')
ax2.set_xlim(left=75, right=125);
```

```
Out[76]: <Figure size 720x432 with 0 Axes>
```

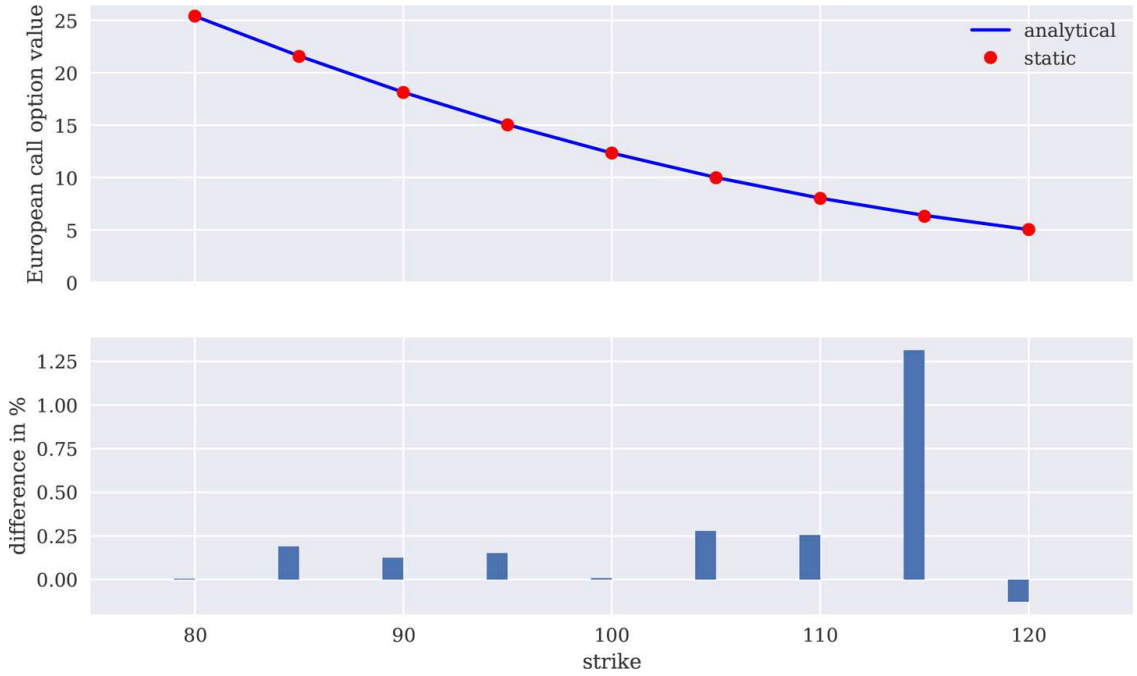

Figure 12-15. Analytical option values vs. Monte Carlo estimators (static simulation)

A similar picture emerges for the dynamic simulation and valuation approach, whose results are reported in Figure 12-16. Again, all valuation differences are smaller than 1% absolutely, with both positive and negative deviations. As a general rule, the quality of the Monte Carlo estimator can be controlled for by adjusting the number of time intervals M used and/or the number of paths I simulated:

```
In [77]: fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True, figsize=(10, 6))
         ax1.plot(k_list, anal_res, 'b', label='analytical')
         ax1.plot(k_list, dyna_res, 'ro', label='dynamic')
         ax1.set_ylabel('European call option value')
         ax1.legend(loc=0)
         ax1.set_ylim(bottom=0)
         wi = 1.0
         ax2.bar(k_list - wi / 2, (anal_res - dyna_res) / anal_res * 100, wi)
         ax2.set_xlabel('strike')
         ax2.set_ylabel('difference in %')
         ax2.set_xlim(left=75, right=125);
```

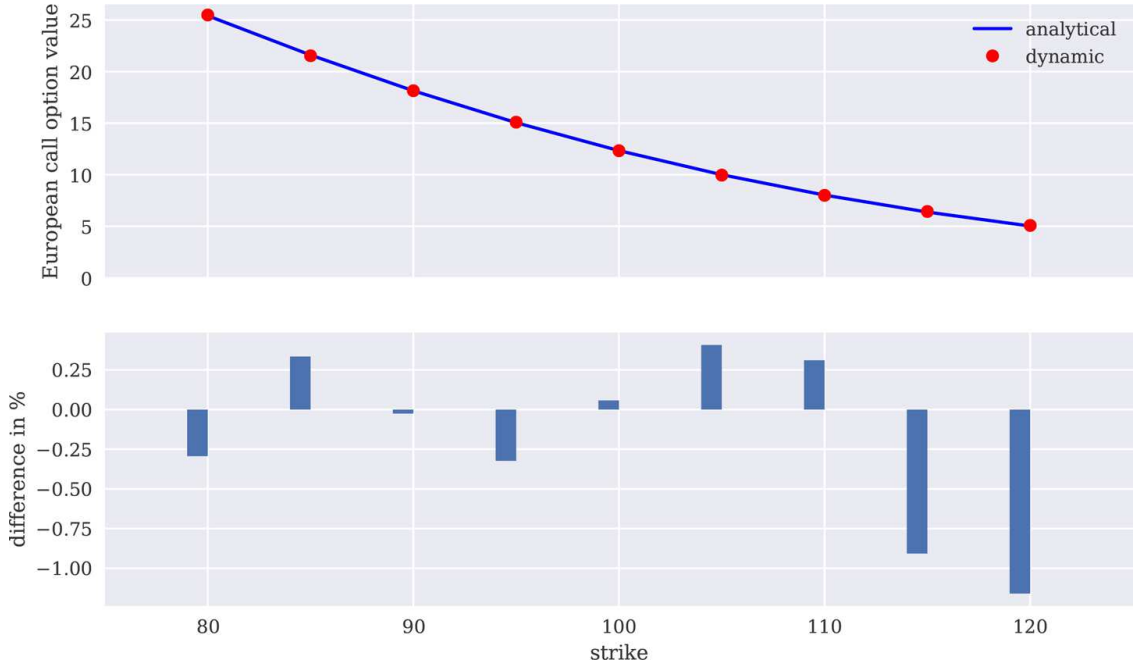

Figure 12-16. Analytical option values vs. Monte Carlo estimators (dynamic simulation)

American Options

The valuation of American options is more involved compared to European options. In this case, an *optimal stopping* problem has to be solved to come up with a fair value of the option. Equation 12-12 formulates the valuation of an American option as such a problem. The problem formulation is already based on a discrete time grid for use with numerical simulation. In a sense, it is therefore more correct to speak of an option value given *Bermudan* exercise. For the time interval converging to zero length, the value of the Bermudan option converges to the one of the American option.

Equation 12-12. American option prices as optimal stopping problem

$$V_0 = \sup_{\tau \in \{0, \Delta t, 2\Delta t, \ldots, T\}} e^{-rT} \mathbf{E}_0^Q \left(h_\tau \left(S_\tau \right) \right)$$

The algorithm described in the following is called *Least-Squares Monte Carlo* (LSM) and is from the paper by Longstaff and Schwartz (2001). It can be shown that the value of an American (Bermudan) option at any given date t is given as $V_t(s) = \max\left(h_t(s), C_t(s)\right)$, where $C_t(s) = \mathbf{E}_t^Q\left(e^{-r\Delta t} V_{t+\Delta t}\left(S_{t+\Delta t}\right) \middle| S_t = s\right)$ is the so-called *continuation value* of the option given an index level of S_t=s.

Consider now that we have simulated I paths of the index level over M time intervals of equal size Δt. Define $Y_{t,i} \equiv e^{-r\Delta t} V_{t+\Delta t,i}$ to be the simulated continuation value for path i at time t. We cannot use this number directly because it would imply perfect foresight. However, we can use the cross section of all such simulated continuation values to estimate the (expected) continuation value by least-squares regression.

Given a set of basis functions $b_d, d = 1, \cdots, D$, the continuation value is then given by the regression estimate $\hat{C}_{t,i} = \sum_{d=1}^{D} \alpha^*_{d,t} \cdot b_d(S_{t,i})$, where the optimal regression parameters α^* are the solution of the least-squares problem stated in Equation 12-13.

Equation 12-13. Least-squares regression for American option valuation

$$\min_{\alpha_{1,t},\ldots,\alpha_{D,t}} \frac{1}{I} \sum_{i=1}^{I} \left(Y_{t,i} - \sum_{d=1}^{D} \alpha_{d,t} \cdot b_d(S_{t,i}) \right)^2$$

The function `gbm_mcs_amer()` implements the LSM algorithm for both American call and put options:[4]

```
In [78]: def gbm_mcs_amer(K, option='call'):
             ''' Valuation of American option in Black-Scholes-Merton
             by Monte Carlo simulation by LSM algorithm

             Parameters
             ==========
             K: float
                 (positive) strike price of the option
             option: string
                 type of the option to be valued ('call', 'put')

             Returns
             =======
             C0: float
                 estimated present value of American call option
             '''
             dt = T / M
             df = math.exp(-r * dt)
             # simulation of index levels
             S = np.zeros((M + 1, I))
```

```
                S[0] = S0
                sn = gen_sn(M, I)
                for t in range(1, M + 1):
                    S[t] = S[t - 1] * np.exp((r - 0.5 * sigma ** 2) * dt
                            + sigma * math.sqrt(dt) * sn[t])
                # case based calculation of payoff
                if option == 'call':
                    h = np.maximum(S - K, 0)
                else:
                    h = np.maximum(K - S, 0)
                # LSM algorithm
                V = np.copy(h)
                for t in range(M - 1, 0, -1):
                    reg = np.polyfit(S[t], V[t + 1] * df, 7)
                    C = np.polyval(reg, S[t])
                    V[t] = np.where(C > h[t], V[t + 1] * df, h[t])
                # MCS estimator
                C0 = df * np.mean(V[1])
                return C0
```

```
In [79]: gbm_mcs_amer(110., option='call')
Out[79]: 7.721705606305352

In [80]: gbm_mcs_amer(110., option='put')
Out[80]: 13.609997625418051
```

The European value of an option represents a lower bound to the American option's value. The difference is generally called the *early exercise premium*. What follows compares European and American option values for the same range of strikes as before to estimate the early exercise premium, this time with puts:[5]

```
In [81]: euro_res = []
         amer_res = []

In [82]: k_list = np.arange(80., 120.1, 5.)

In [83]: for K in k_list:
             euro_res.append(gbm_mcs_dyna(K, 'put'))
             amer_res.append(gbm_mcs_amer(K, 'put'))
```

```
In [84]: euro_res = np.array(euro_res)
         amer_res = np.array(amer_res)
```

Figure 12-17 shows that for the range of strikes chosen the early exercise premium can rise to up to 10%:

```
In [85]: fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True, figsize=(10, 6))
         ax1.plot(k_list, euro_res, 'b', label='European put')
         ax1.plot(k_list, amer_res, 'ro', label='American put')
         ax1.set_ylabel('call option value')
         ax1.legend(loc=0)
         wi = 1.0
         ax2.bar(k_list - wi / 2, (amer_res - euro_res) / euro_res * 100, wi)
         ax2.set_xlabel('strike')
         ax2.set_ylabel('early exercise premium in %')
         ax2.set_xlim(left=75, right=125);
```

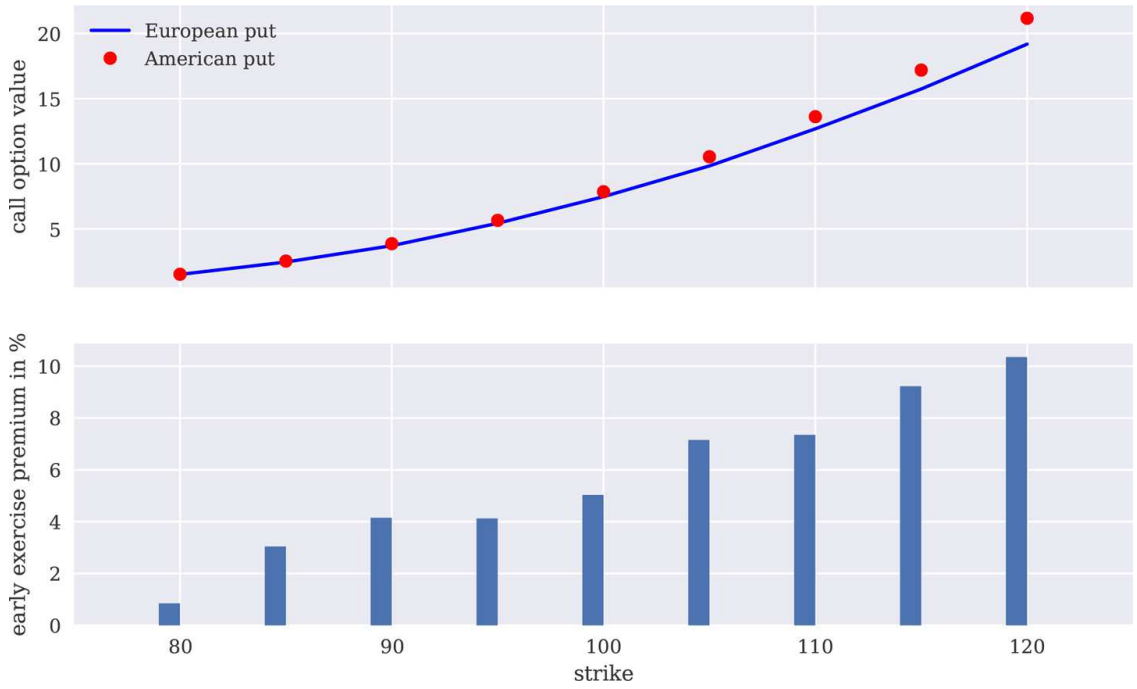

Figure 12-17. European vs. American Monte Carlo estimators

Risk Measures

In addition to valuation, *risk management* is another important application area of stochastic methods and simulation. This section illustrates the calculation/estimation of two of the most common risk measures applied today in the finance industry.

Value-at-Risk

Value-at-risk (VaR) is one of the most widely used risk measures, and a much debated one. Loved by practitioners for its intuitive appeal, it is widely discussed and criticized by many—mainly on theoretical grounds, with regard to its limited ability to capture what is called *tail risk* (more on this shortly). In words, VaR is a number denoted in currency units (e.g., USD, EUR, JPY) indicating a loss (of a portfolio, a single position, etc.) that is not exceeded with some confidence level (probability) over a given period of time.

Consider a stock position, worth 1 million USD today, that has a VaR of 50,000 USD at a confidence level of 99% over a time period of 30 days (one month). This VaR figure says that with a probability of 99% (i.e., in 99 out of 100 cases), the loss to be expected over a period of 30 days will *not exceed* 50,000 USD. However, it does not say anything about the size of the loss once a loss beyond 50,000 USD occurs—i.e., if the maximum loss is 100,000 or 500,000 USD what the probability of such a specific "higher than VaR loss" is. All it says is that there is a 1% probability that a loss of a *minimum of 50,000 USD or higher* will occur.

Assume the Black-Scholes-Merton setup and consider the following parameterization and simulation of index levels at a future date $T = 30/365$ (a period of 30 days). The estimation of VaR figures requires the simulated absolute profits and losses relative to the value of the position today in a sorted manner, i.e., from the severest loss to the largest profit. Figure 12-18 shows the histogram of the simulated absolute performance values:

```
In [86]: S0 = 100
         r = 0.05
         sigma = 0.25
         T = 30 / 365.
         I = 10000

In [87]: ST = S0 * np.exp((r - 0.5 * sigma ** 2) * T +
                          sigma * np.sqrt(T) * npr.standard_normal(I))  ❶

In [88]: R_gbm = np.sort(ST - S0)  ❷
```

```
In [89]: plt.figure(figsize=(10, 6))
         plt.hist(R_gbm, bins=50)
         plt.xlabel('absolute return')
         plt.ylabel('frequency');
```

❶ Simulates end-of-period values for the geometric Brownian motion.

❷ Calculates the absolute profits and losses per simulation run and sorts the values.

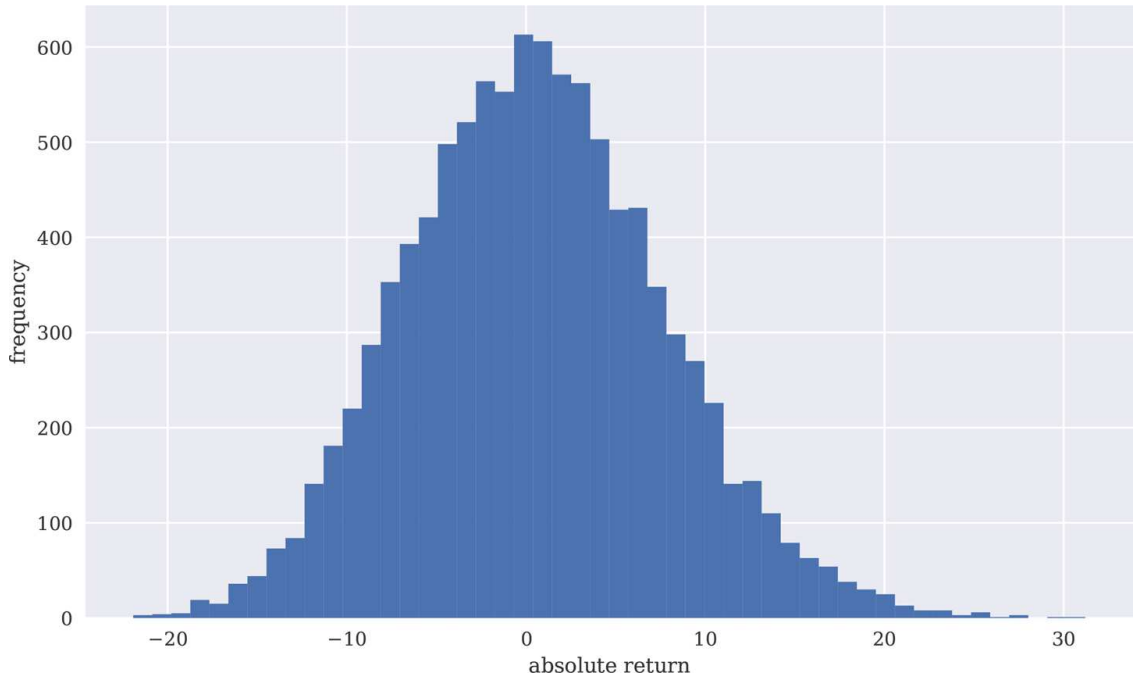

Figure 12-18. Absolute profits and losses from simulation (geometric Brownian motion)

Having the `ndarray` object with the sorted results, the `scs.scoreatpercentile()` function already does the trick. All one has to do is to define the percentiles of interest (in percent values). In the `list` object `percs`, `0.1` translates into a confidence level of 100% – 0.1% = 99.9%. The 30-day VaR given a confidence level of 99.9% in this case is 18.8 currency units, while it is 8.5 at the 90% confidence level:

```
In [91]: percs = [0.01, 0.1, 1., 2.5, 5.0, 10.0]
         var = scs.scoreatpercentile(R_gbm, percs)
         print('%16s %16s' % ('Confidence Level', 'Value-at-Risk'))
         print(33 * '-')
         for pair in zip(percs, var):
             print('%16.2f %16.3f' % (100 - pair[0], -pair[1]))
         Confidence Level    Value-at-Risk
```

```
--------------------------------
            99.99           21.814
            99.90           18.837
            99.00           15.230
            97.50           12.816
            95.00           10.824
            90.00            8.504
```

As a second example, recall the jump diffusion setup from Merton, which is simulated dynamically. In this case, with the jump component having a negative mean, one sees something like a bimodal distribution for the simulated profits/losses in Figure 12-19. From a normal distribution point of view, one sees a pronounced left *fat tail*:

```
In [92]: dt = 30. / 365 / M
         rj = lamb * (math.exp(mu + 0.5 * delta ** 2) - 1)

In [93]: S = np.zeros((M + 1, I))
         S[0] = S0
         sn1 = npr.standard_normal((M + 1, I))
         sn2 = npr.standard_normal((M + 1, I))
         poi = npr.poisson(lamb * dt, (M + 1, I))
         for t in range(1, M + 1, 1):
             S[t] = S[t - 1] * (np.exp((r - rj - 0.5 * sigma ** 2) * dt
                                + sigma * math.sqrt(dt) * sn1[t])
                                + (np.exp(mu + delta * sn2[t]) - 1)
                                * poi[t])
             S[t] = np.maximum(S[t], 0)

In [94]: R_jd = np.sort(S[-1] - S0)

In [95]: plt.figure(figsize=(10, 6))
         plt.hist(R_jd, bins=50)
         plt.xlabel('absolute return')
         plt.ylabel('frequency');
```

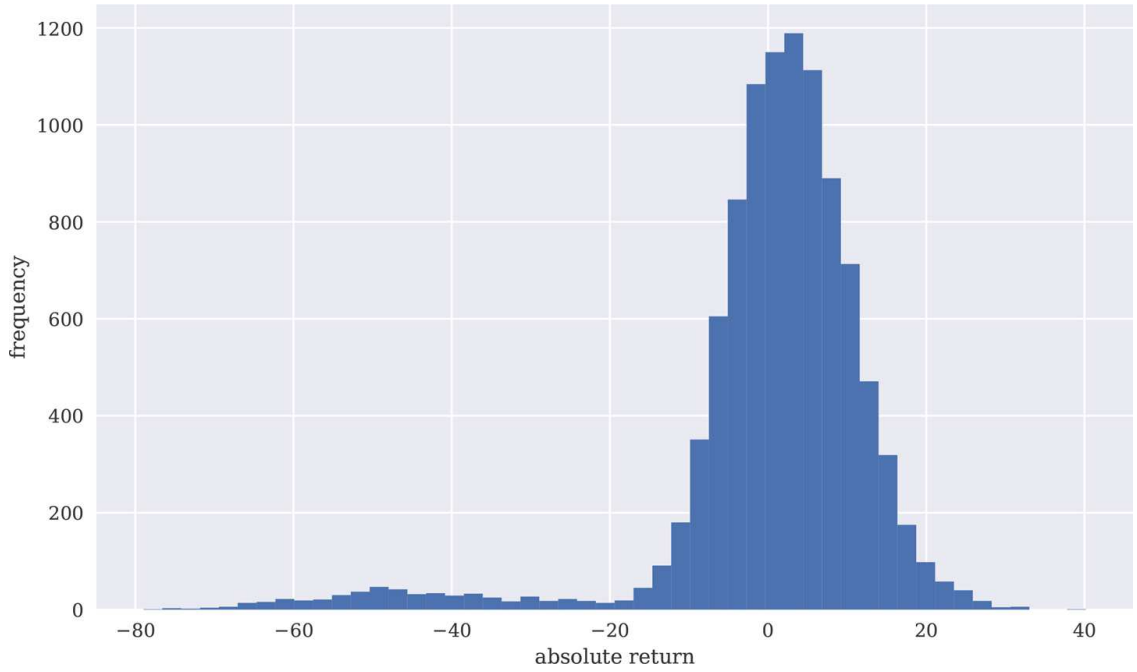

Figure 12-19. Absolute profits and losses from simulation (jump diffusion)

For this process and parameterization, the VaR over 30 days at the 90% level is almost identical as with the geometric Brownian motion, while it is more than *three times* as high at the 99.9% level (70 vs. 18.8 currency units):

```
In [96]: percs = [0.01, 0.1, 1., 2.5, 5.0, 10.0]
         var = scs.scoreatpercentile(R_jd, percs)
         print('%16s %16s' % ('Confidence Level', 'Value-at-Risk'))
         print(33 * '-')
         for pair in zip(percs, var):
             print('%16.2f %16.3f' % (100 - pair[0], -pair[1]))
         Confidence Level    Value-at-Risk
         ---------------------------------
                    99.99           76.520
                    99.90           69.396
                    99.00           55.974
                    97.50           46.405
                    95.00           24.198
                    90.00            8.836
```

This illustrates the problem of capturing the tail risk so often encountered in financial markets by the standard VaR measure.

To further illustrate the point, Figure 12-20 lastly shows the VaR measures for both cases in direct comparison graphically. As the plot reveals, the VaR measures behave completely differently given a range of typical confidence levels:

```
In [97]: percs = list(np.arange(0.0, 10.1, 0.1))
         gbm_var = scs.scoreatpercentile(R_gbm, percs)
         jd_var = scs.scoreatpercentile(R_jd, percs)

In [98]: plt.figure(figsize=(10, 6))
         plt.plot(percs, gbm_var, 'b', lw=1.5, label='GBM')
         plt.plot(percs, jd_var, 'r', lw=1.5, label='JD')
         plt.legend(loc=4)
         plt.xlabel('100 - confidence level [%]')
         plt.ylabel('value-at-risk')
         plt.ylim(ymax=0.0);
```

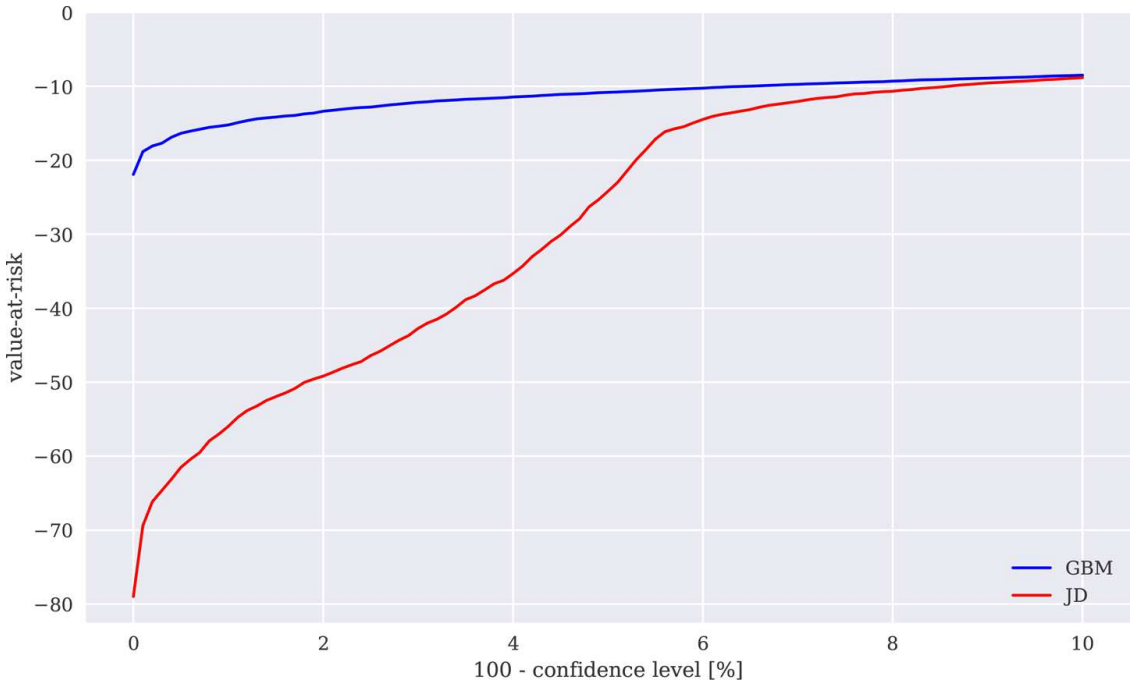

Figure 12-20. Value-at-risk for geometric Brownian motion and jump diffusion

Credit Valuation Adjustments

Other important risk measures are the credit value-at-risk (CVaR) and the credit valuation adjustment (CVA), which is derived from the CVaR. Roughly speaking, CVaR is a measure for the risk resulting from the possibility that a counterparty might not be able to honor its obligations—for example, if the counterparty goes bankrupt. In such a case there are two

main assumptions to be made: the *probability of default* and the (average) *loss level*.

To make it specific, consider again the benchmark setup of Black-Scholes-Merton with the parameterization in the following code. In the simplest case, one considers a fixed (average) loss level L and a fixed probability p of default (per year) of a counterparty. Using the Poisson distribution, default scenarios are generated as follows, taking into account that a default can only occur once:

```
In [99]: S0 = 100.
         r = 0.05
         sigma = 0.2
         T = 1.
         I = 100000

In [100]: ST = S0 * np.exp((r - 0.5 * sigma ** 2) * T
                           + sigma * np.sqrt(T) * npr.standard_normal(I))

In [101]: L = 0.5  ❶

In [102]: p = 0.01  ❷

In [103]: D = npr.poisson(p * T, I)  ❸

In [104]: D = np.where(D > 1, 1, D)  ❹
```

❶ Defines the loss level.

❷ Defines the probability of default.

❸ Simulates default events.

❹ Limits defaults to one such event.

Without default, the risk-neutral value of the future index level should be equal to the current value of the asset today (up to differences resulting from numerical errors). The CVaR and the present value of the asset, adjusted for the credit risk, are given as follows:

```
In [105]: math.exp(-r * T) * np.mean(ST)  ❶
Out[105]: 99.94767178982691

In [106]: CVaR = math.exp(-r * T) * np.mean(L * D * ST)  ❷
          CVaR  ❷
Out[106]: 0.4883560258963962

In [107]: S0_CVA = math.exp(-r * T) * np.mean((1 - L * D) * ST)  ❸
          S0_CVA  ❸
Out[107]: 99.45931576393053

In [108]: S0_adj = S0 - CVaR  ❹
          S0_adj  ❹
Out[108]: 99.5116439741036
```

❶ Discounted average simulated value of the asset at T.

❷ CVaR as the discounted average of the future losses in the case of a default.

❸ Discounted average simulated value of the asset at T, adjusted for the simulated losses from default.

❹ Current price of the asset adjusted by the simulated CVaR.

In this particular simulation example, one observes roughly 1,000 losses due to credit risk, which is to be expected given the assumed default probability of 1% and 100,000 simulated paths. Figure 12-21 shows the complete frequency distribution of the losses due to a default. Of course, in the large majority of cases (i.e., in about 99,000 of the 100,000 cases) there is no loss to observe:

```
In [109]: np.count_nonzero(L * D * ST)  ❶
Out[109]: 978

In [110]: plt.figure(figsize=(10, 6))
          plt.hist(L * D * ST, bins=50)
          plt.xlabel('loss')
```

```
plt.ylabel('frequency')
plt.ylim(ymax=175);
```

❶ Number of default events and therewith loss events.

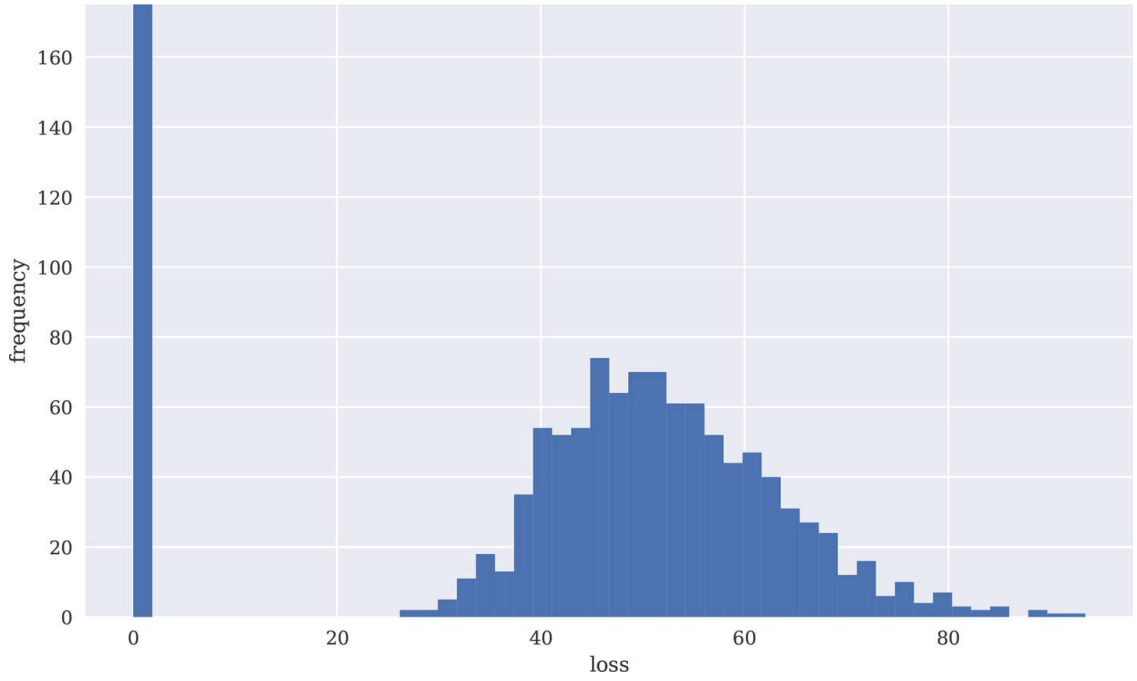

Figure 12-21. Losses due to risk-neutrally expected default (stock)

Consider now the case of a European call option. Its value is about 10.4 currency units at a strike of 100. The CVaR is about 5 cents given the same assumptions with regard to probability of default and loss level:

```
In [111]: K = 100.
          hT = np.maximum(ST - K, 0)

In [112]: C0 = math.exp(-r * T) * np.mean(hT)  ❶
          C0  ❶
Out[112]: 10.396916492839354

In [113]: CVaR = math.exp(-r * T) * np.mean(L * D * hT)  ❷
          CVaR  ❷
Out[113]: 0.05159099858923533

In [114]: C0_CVA = math.exp(-r * T) * np.mean((1 - L * D) * hT)  ❸
          C0_CVA  ❸
Out[114]: 10.34532549425012
```

❶ The Monte Carlo estimator value for the European call option.

❷ The CVaR as the discounted average of the future losses in the case of a default.

❸ The Monte Carlo estimator value for the European call option, adjusted for the simulated losses from default.

Compared to the case of a regular asset, the option case has somewhat different characteristics. One only sees a little more than 500 losses due to a default, although there are again 1,000 defaults in total. This results from the fact that the payoff of the option at maturity has a high probability of being zero. Figure 12-22 shows that the CVaR for the option has quite a different frequency distribution compared to the regular asset case:

```
In [115]: np.count_nonzero(L * D * hT)  ❶
Out[115]: 538

In [116]: np.count_nonzero(D)  ❷
Out[116]: 978

In [117]: I - np.count_nonzero(hT)  ❸
Out[117]: 44123

In [118]: plt.figure(figsize=(10, 6))
          plt.hist(L * D * hT, bins=50)
          plt.xlabel('loss')
          plt.ylabel('frequency')
          plt.ylim(ymax=350);
```

❶ The number of losses due to default.

❷ The number of defaults.

❸ The number of cases for which the option expires worthless.

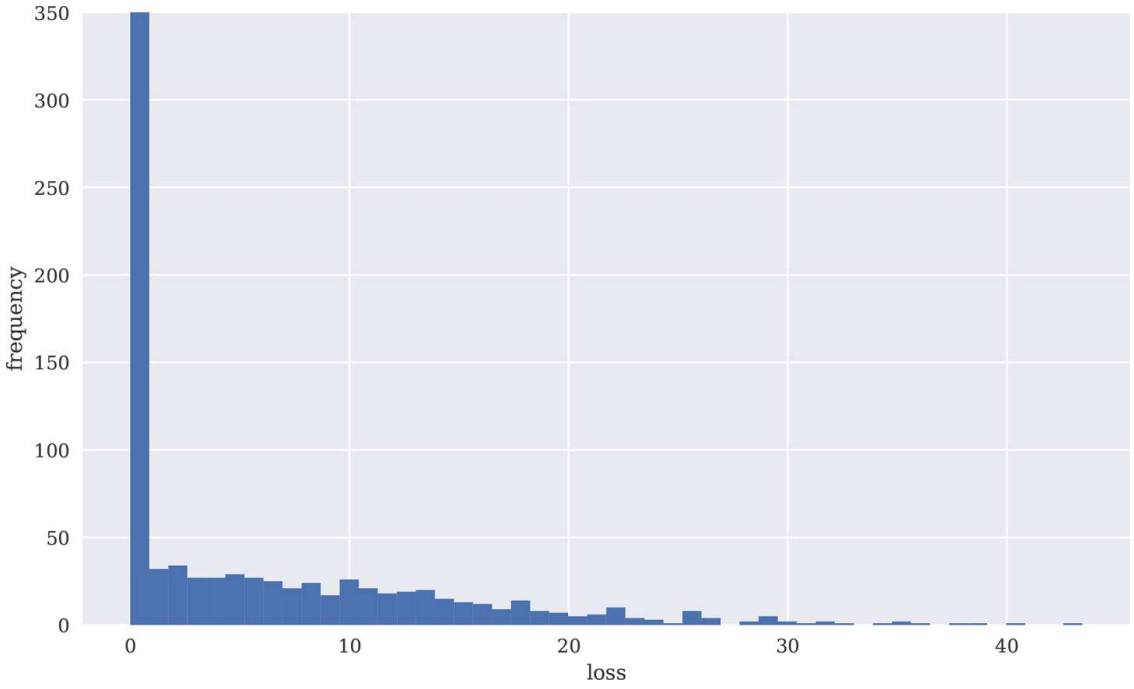

Figure 12-22. Losses due to risk-neutrally expected default (call option)

Python Script

The following presents an implementation of central functions related to the Black-Scholes-Merton model for the analytical pricing of European (call) options. For details of the model, see Black and Scholes (1973) as well as Merton (1973). See Appendix B for an alternative implementation based on a Python class.

```
#
# Valuation of European call options
# in Black-Scholes-Merton model
# incl. vega function and implied volatility estimation
# bsm_functions.py
#
# (c) Dr. Yves J. Hilpisch
# Python for Finance, 2nd ed.
#


def bsm_call_value(S0, K, T, r, sigma):
    ''' Valuation of European call option in BSM model.
    Analytical formula.
```

```
    Parameters
    ==========
    S0: float
        initial stock/index level
    K: float
        strike price
    T: float
        maturity date (in year fractions)
    r: float
        constant risk-free short rate
    sigma: float
        volatility factor in diffusion term

    Returns
    =======
    value: float
        present value of the European call option
    '''
    from math import log, sqrt, exp
    from scipy import stats

    S0 = float(S0)
    d1 = (log(S0 / K) + (r + 0.5 * sigma ** 2) * T) / (sigma * sqrt(T))
    d2 = (log(S0 / K) + (r - 0.5 * sigma ** 2) * T) / (sigma * sqrt(T))
    # stats.norm.cdf --> cumulative distribution function
    #                    for normal distribution
    value = (S0 * stats.norm.cdf(d1, 0.0, 1.0) -
             K * exp(-r * T) * stats.norm.cdf(d2, 0.0, 1.0))
    return value


def bsm_vega(S0, K, T, r, sigma):
    ''' Vega of European option in BSM model.

    Parameters
    ==========
    S0: float
        initial stock/index level
    K: float
        strike price
    T: float
        maturity date (in year fractions)
    r: float
        constant risk-free short rate
```

```
    sigma: float
        volatility factor in diffusion term

    Returns
    =======
    vega: float
        partial derivative of BSM formula with respect
        to sigma, i.e. vega

    '''
    from math import log, sqrt
    from scipy import stats

    S0 = float(S0)
    d1 = (log(S0 / K) + (r + 0.5 * sigma ** 2) * T) / (sigma * sqrt(T))
    vega = S0 * stats.norm.pdf(d1, 0.0, 1.0) * sqrt(T)
    return vega

# Implied volatility function


def bsm_call_imp_vol(S0, K, T, r, C0, sigma_est, it=100):
    ''' Implied volatility of European call option in BSM model.

    Parameters
    ==========
    S0: float
        initial stock/index level
    K: float
        strike price
    T: float
        maturity date (in year fractions)
    r: float
        constant risk-free short rate
    sigma_est: float
        estimate of impl. volatility
    it: integer
        number of iterations

    Returns
    =======
    simga_est: float
        numerically estimated implied volatility
    '''
```

```
    for i in range(it):
        sigma_est -= ((bsm_call_value(S0, K, T, r, sigma_est) - C0) /
                      bsm_vega(S0, K, T, r, sigma_est))
    return sigma_est
```

Conclusion

This chapter deals with methods and techniques important to the application of Monte Carlo simulation in finance. In particular, it first shows how to generate pseudo-random numbers based on different distribution laws. It proceeds with the simulation of random variables and stochastic processes, which is important in many financial areas. Two application areas are discussed in some depth in this chapter: valuation of options with European and American exercise and the estimation of risk measures like value-at-risk and credit valuation adjustments.

The chapter illustrates that Python in combination with `NumPy` is well suited to implementing even such computationally demanding tasks as the valuation of American options by Monte Carlo simulation. This is mainly due to the fact that the majority of functions and classes of `NumPy` are implemented in C, which leads to considerable speed advantages in general over pure Python code. A further benefit is the compactness and readability of the resulting code due to vectorized operations.

Further Resources

The original article introducing Monte Carlo simulation to finance is:

- Boyle, Phelim (1977). "Options: A Monte Carlo Approach." *Journal of Financial Economics*, Vol. 4, No. 4, pp. 322–338.

Other original papers cited in this chapter are (see also Chapter 18):

- Black, Fischer, and Myron Scholes (1973). "The Pricing of Options and Corporate Liabilities." *Journal of Political Economy*, Vol. 81, No. 3, pp. 638–659.

- Cox, John, Jonathan Ingersoll, and Stephen Ross (1985). “A Theory of the Term Structure of Interest Rates.” *Econometrica*, Vol. 53, No. 2, pp. 385–407.
- Heston, Steven (1993). “A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options.” *The Review of Financial Studies*, Vol. 6, No. 2, 327–343.
- Merton, Robert (1973). “Theory of Rational Option Pricing.” *Bell Journal of Economics and Management Science*, Vol. 4, pp. 141–183.
- Merton, Robert (1976). “Option Pricing When the Underlying Stock Returns Are Discontinuous.” *Journal of Financial Economics*, Vol. 3, No. 3, pp. 125–144.

The following books cover the topics of this chapter in more depth (however, the first one does not cover technical implementation details):

- Glasserman, Paul (2004). *Monte Carlo Methods in Financial Engineering*. New York: Springer.
- Hilpisch, Yves (2015). *Derivatives Analytics with Python*. Chichester, England: Wiley Finance.

It took until the turn of the century for an efficient method to value American options by Monte Carlo simulation to finally be published:

- Longstaff, Francis, and Eduardo Schwartz (2001). “Valuing American Options by Simulation: A Simple Least Squares Approach.” *Review of Financial Studies*, Vol. 14, No. 1, pp. 113–147.

A broad and in-depth treatment of credit risk is provided in:

- Duffie, Darrell, and Kenneth Singleton (2003). *Credit Risk—Pricing, Measurement, and Management*. Princeton, NJ: Princeton University Press.

1 For simplicity, we will speak of *random numbers* knowing that all numbers used will be *pseudo-random*.

2 The approach here is inspired by the Law of Large Numbers.

3 The described method works for symmetric median 0 random variables only, like standard normally distributed random variables, which are almost exclusively used throughout.

4 For algorithmic details, refer to Hilpisch (2015).

5 Since no dividend payments are assumed (having an index in mind), there generally is no early exercise premium for call options (i.e., no incentive to exercise the option early).