Berlin School of Economics and Law
Prof. Dr. Natalie Packham

Summer term 2021
LV-Nr. 425093

Computational Finance and FinTech

Probabilities and Monte Carlo simulation

Contents

7 Probabilities and Monte Carlo simulation . . . 1
7.1 Random numbers . . . 1
7.2 Monte Carlo Simulation . . . 4
7.3 Option pricing . . . 10
7.4 Path-dependent options . . . 12

7 Probabilities and Monte Carlo simulation

- Further reading: **Py4Fi, Chapter 12**
- This session also covers material not in **Py4Fi**.
- Uncertainty about outcomes lies at the heart of many financial applications.
- By quantifying the uncertainty in terms of probabilities we can price derivatives and measure risk.

Some general imports

- `numpy.random` is the random number generation subpackage of `numpy`.

[1]:
```
import math
import numpy as np
import numpy.random as npr
from pylab import plt, mpl
import scipy.stats as scs
```

[2]:
```
plt.style.use('seaborn')
%matplotlib inline
```

7.1 Random numbers

- A random number generator generates so-called **pseudo-random numbers**.
- Despite being deterministic, they resemble random numbers by replicating their statistical properties.
- The `seed` is the starting point from which a sequence of random numbers are generated.
- Fixing the seed allows to reproduce a sequence of random numbers, which is useful for development and debugging.
- If no seed is fixed, then this is usually set internally to a number derived from the current timestamp.

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

Random numbers

- `rand` generates random numbers of the uniform distribution spanning the $[0, 1]$ interval.

[4]:
```
npr.rand(10)
```

[4]: array([0.5434, 0.2784, 0.4245, 0.8448, 0.0047, 0.1216, 0.6707, 0.8259,
       0.1367, 0.5751])

```
[5]: npr.rand(5, 5)
```

[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]])

Random numbers

- Functions for simple random number generations:

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

Random number generation

Source: Python for Finance, 2nd ed.

Histogram

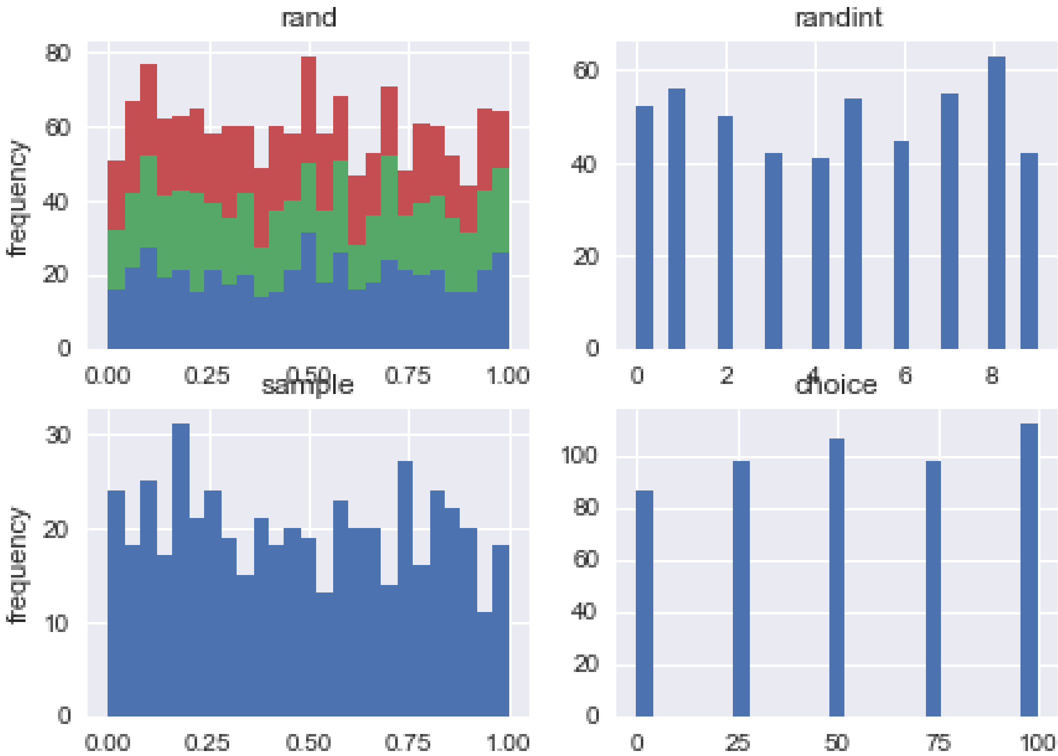
- Use `hist` to plot histograms:

```
[6]: sample_size = 500
rn1 = npr.rand(sample_size, 3)  # 3-dimensional array of 500 uniform random numbers␣
 ↪each
rn2 = npr.randint(0, 10, sample_size)  # random integers
rn3 = npr.sample(size=sample_size)  # another way of drawing uniforms
a = [0, 25, 50, 75, 100]
rn4 = npr.choice(a, size=sample_size) # drawing from a given distribution
```

Histogram

```
[7]: fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2, ncols=2,
                                             figsize=(7, 5))
ax1.hist(rn1, bins=25, stacked=True) # create a histogram with 25 bins
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');
```



Distributions

Function	Description
`binomial(n,p[,size])`	Draw samples from a binomial distribution
`exponential([scale, size])`	Draw samples from an exponential distribution
`lognormal([mean, sigma, size])`	Draw samples from a log-normal distribution
`multivariate_normal(mean, cov[, size, ...)`	Draw random samples from a multivariate normal distribution
`normal([loc, scale, size])`	Draw random samples from a normal (Gaussian) distribution
`standard_normal([size])`	Draw samples from a standard Normal distribution (mean=0, stdev=1)
`standard_t(df[, size])`	Draw samples from a standard Student's t distribution with df degrees of freedom

- More functions and distributions are documented scipy.org.

Distributions

- Generate random numbers from various distributions and plot their histograms.

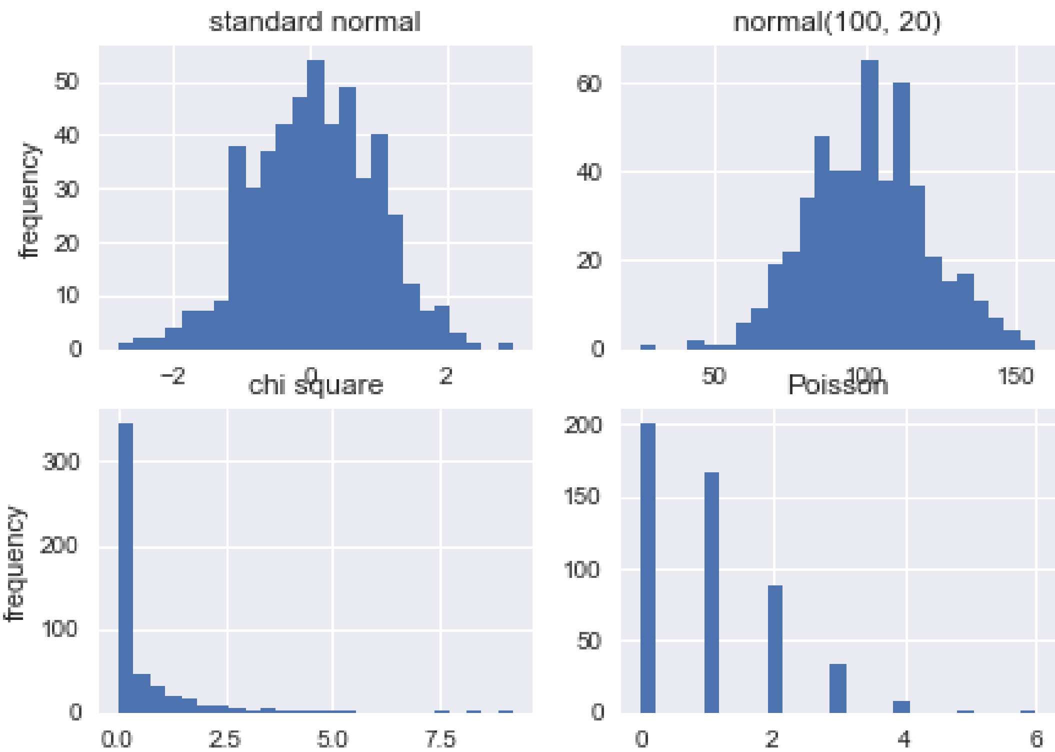
[8]:

```
sample_size = 500
rn1 = npr.standard_normal(sample_size)
rn2 = npr.normal(100, 20, sample_size)
rn3 = npr.chisquare(df=0.5, size=sample_size)
rn4 = npr.poisson(lam=1.0, size=sample_size)
```

Distributions

[9]:

```
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2, ncols=2,
                                              figsize=(7, 5))
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');
```



7.2 Monte Carlo Simulation

- **Monte Carlo simulation** is a powerful tool for the numerical computation of expectations and other statistics.
- Monte Carlo simulation refers to the simulation of (independent) samples of a random variable Z using computer-generated random numbers.

- Once sufficiently many random numbers have been drawn, these can be used to produce an **estimate** of some quantity that depends on the distribution of Z.
- The quality of an estimate can be quantified by a **confidence interval** around the estimate.
- A comprehensive resource is

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

Monte Carlo simulation

- To fix ideas consider the problem of estimating the integral of a function f over the unit interval:

$$\alpha = \int_0^1 f(x)\, dx.$$

- We may write

$$\alpha = \mathbb{E}[f(U)],$$

  with U uniformly distributed between 0 and 1.

Monte Carlo simulation

- Drawing points $u_1, u_2, \ldots, u_n$ independently and uniformly from $[0,1]$, the Monte Carlo estimate is given by

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

- If f is integrable over $[0,1]$ then, by the **strong law of large numbers**,

$$\hat{\alpha}_n \to \alpha \text{ with probability } 1, \text{ as } n \to \infty.$$

Monte Carlo simulation

- If f is square integrable, and setting,

$$\sigma_f^2 = \int_0^1 (f(x) - \alpha)^2\, dx,$$

  then, by the **central limit theorem**, the error $\hat{\alpha}_n - \alpha$ is approximately normally distributed with mean 0 and standard deviation $\sigma_f/\sqrt{n}$.
- σ_f is typically unknown, but can be estimated by the sample standard deviation

$$s_f = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(f(u_i) - \hat{\alpha}_n)^2}.$$

Monte Carlo simulation

- Thus, an (asymptotically) valid $1-\delta$ confidence interval for α is given by

$$\left[\hat{\alpha}_n - \mathrm{N}_{1-\delta/2}\frac{s_f}{\sqrt{n}}\ ,\ \hat{\alpha}_n + \mathrm{N}_{1-\delta/2}\frac{s_f}{\sqrt{n}}\right],$$

  where $\mathrm{N}_{1-\delta/2}$ denotes the $1-\delta/2$ quantile of the standard normal distribution.
- For example, for $1-\delta = 0.95$: $\mathrm{N}_{1-\delta/2} = \mathrm{N}_{0.975} \approx 1.96$.

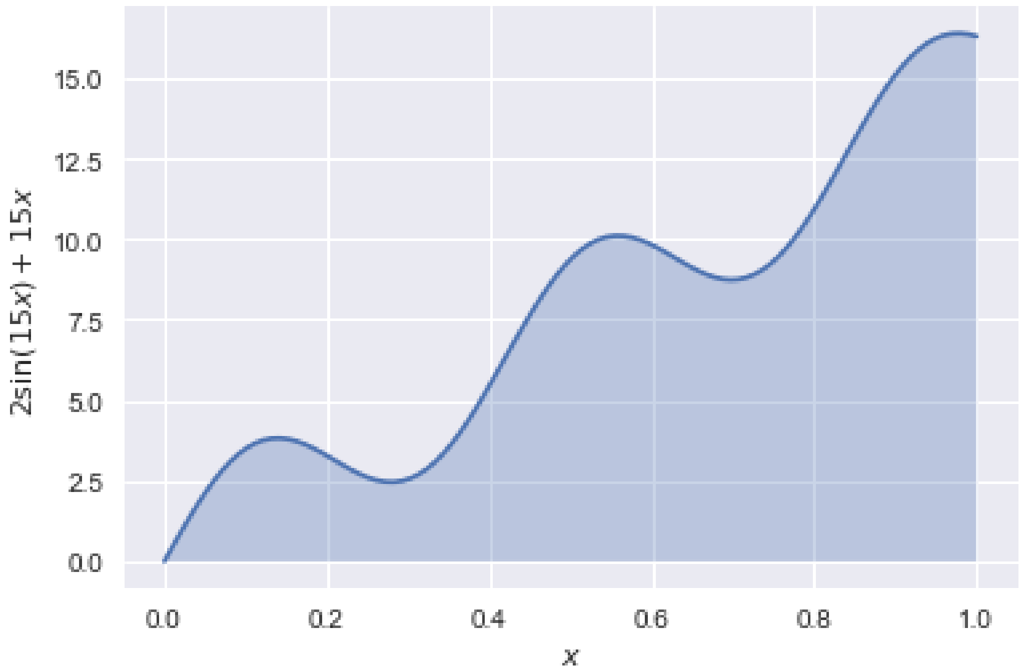
Monte Carlo simulation

- Thus, from the function value $f(u_1), \ldots, f(u_n)$ we obtain
  - an estimate of the integral α,
  - and a measure of the error of the estimate.
- The form of the standard error $\sigma_f/\sqrt{n}$ implies:
  - to cut the error in half requires increasing the sample size by four;
  - adding one decimal point of precision requires 100 times as many points.
- Monte Carlo simulation it particularly well suited for high-dimensional applications, that is, when integrating over $[0,1]^d$, $d \geq 1$.

Monte Carlo simulation - Example 1

- We use Monte Carlo simulation to estimate the integral $\int_0^1 (2\sin(15x) + 15x)\,dx$.
- The solution, calculated analytically, is $\frac{1}{30}(229 - 4\cos(15)) = 7.7346$.

[10]:
```
x = np.linspace(0,1,100)
y = 2 * np.sin(15*x) + 15*x
plt.plot(x,y)
plt.fill_between(x, y, alpha=0.3);
plt.xlabel('$x$');
plt.ylabel('$2 \sin(15x) + 15x$');
```



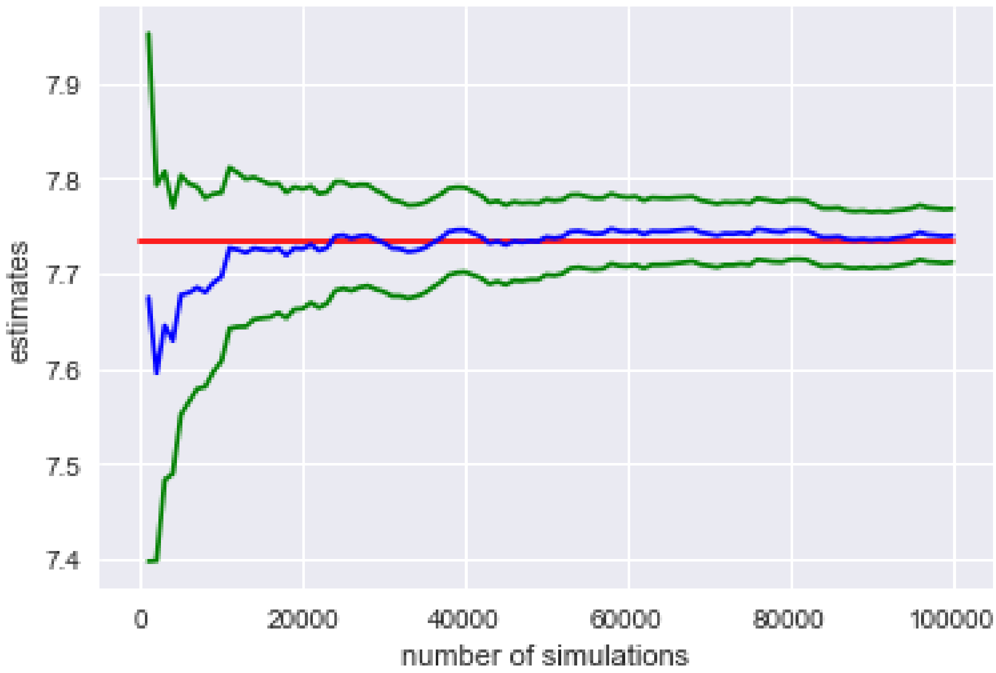
Monte Carlo simulation - Example 1

[11]:
```
n = 100000
z = npr.uniform(0, 1, n)
y = 2 * np.sin(15 * z) + 15 * z
```

[12]:
```
y_m = []
y_cfl = []
y_cfu = []
for i in range(1000, n+1, 1000):
    y_m.append(np.mean(y[:i]))
    y_cfl.append(np.mean(y[:i] - 1.96 * np.std(y[:i])/np.sqrt(i)))
    y_cfu.append(np.mean(y[:i] + 1.96 * np.std(y[:i])/np.sqrt(i)))
```

Monte Carlo simulation - Example 1

[13]:
```
plt.plot( [0,n], [7.73463,7.73463], 'r', range(1000,n+1,1000), y_m, 'b', \
         range(1000,n+1,1000), y_cfl, 'g', range(1000,n+1,1000), y_cfu, 'g');
plt.xlabel('number of simulations');
plt.ylabel('estimates');
```



Monte Carlo simulation - Example 2

- We calculate the area of a circle, given by

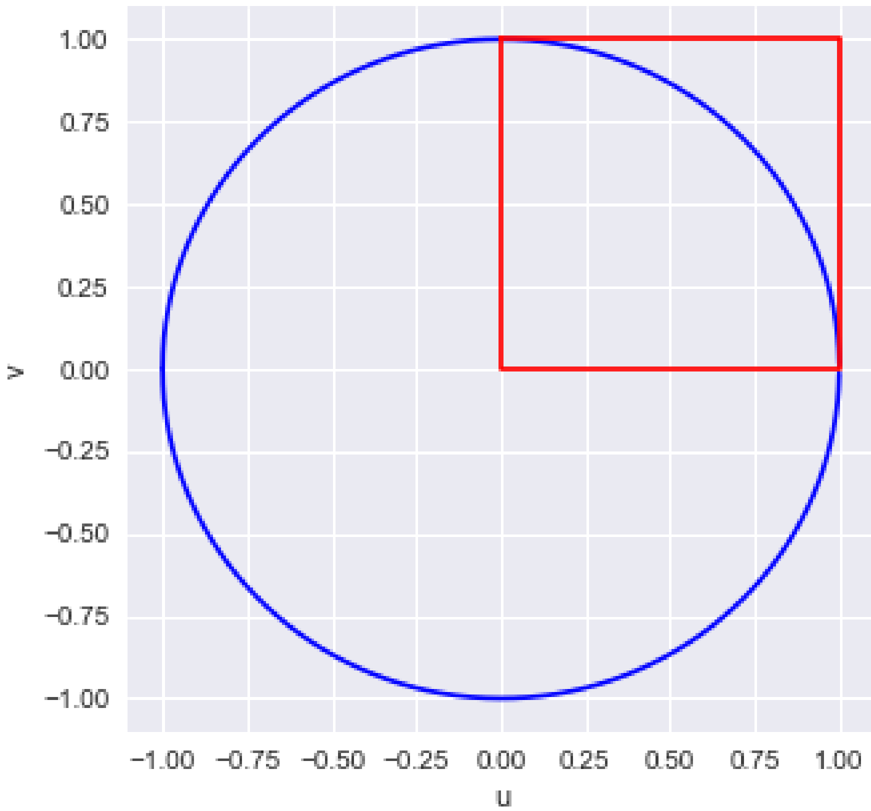
$$\iint_{[-1,1]^2} \mathbf{1}_{\{u^2+v^2\leq 1\}}\, du\, dv = \pi = 3.14159.$$

- In a Monte Carlo simulation, we sample random numbers $u_1, \ldots, u_n$ and $v_1, \ldots, v_n$, each uniformly from $[-1, 1]$.
- Then, we count the number of samples that fulfill $u_k^2 + v_k^2 \leq 1$, $k = 1, \ldots, n$.
- The fraction of such samples will be near $\pi/4$.
- Given that $[-1, 1]^2$ has area 4, we obtain a result near π.

Monte Carlo simulation - Example 2

- The plot below shows the unit circle (a circle with radius 1) and the "pie" of the cirle on the $[0, 1]$ interval:

[14]:
```
plt.figure(figsize=(5,5))
u = np.arange(-1,1,0.0001)
v = np.sqrt(1-u*u)
plt.plot(u,v, 'b')
plt.plot(u,-v,'b')
plt.plot([0,0,1,1,0], [1,0,0,1,1], 'r')
plt.xlabel('u')
plt.ylabel('v');
```



Monte Carlo simulation - Example 2

- The following code performs the Monte Carlo simulation on the "quarter" $[0, 1]$.
- The simulation below tests if a pair of independent uniformly distributed random numbers on $[0, 1]$ lies in the "pie"
- Multiplying the fraction of theses pairs by 4 gives an estimate of the area of the cirle:

[15]:
```
n = 100000
x = npr.uniform(0, 1, (n,2))
y = list(map(lambda u: (4 if u[0]**2 + u[1]**2 <1 else 0), x))
```

- The code contains two short-hand constructions that we look at below.

Lambda functions

- The code contains two short-hand constructions:
  - `lambda arguments: expression` is short-hand for so-called lambda functions.
  - Functions that consist of only one expression can be defined in this short-hand way.
  - For example, the two function definitions below are equivalent:

[16]:
```
x = lambda a : a + 10
print(x(5))
```

```
15
```

[17]:
```
def x(a):
    return a + 10

print(x(5))
```

```
15
```

Short-hand for `if... else...`

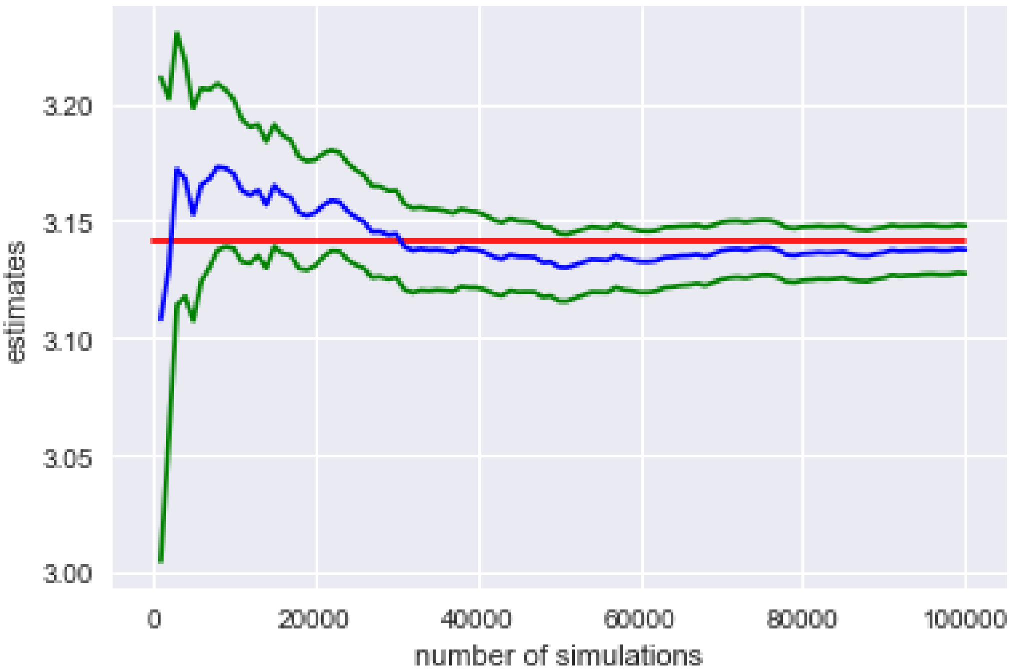
The code `4 if u[0]**2 + u[1]**2<1 else 0` is short-hand for

```
if u[0]**2 + u[1]**2 < 1:
    4
else:
    0
```

Monte Carlo simulation - Example 2 continued

[18]:
```
y_m = []
y_cfl = []
y_cfu = []
for i in range(1000, n+1, 1000):
    y_m.append(np.mean(y[:i]))
    y_cfl.append(np.mean(y[:i] - 1.96 * np.std(y[:i])/np.sqrt(i)))
    y_cfu.append(np.mean(y[:i] + 1.96 * np.std(y[:i])/np.sqrt(i)))
```

[19]:
```
plt.plot( [0,n], [np.pi,np.pi], 'r', range(1000,n+1,1000), y_m, 'b', \
         range(1000,n+1,1000), y_cfl, 'g', range(1000,n+1,1000), y_cfu, 'g');
plt.xlabel('number of simulations');
plt.ylabel('estimates');
```



7.3 Option pricing

Option pricing with Monte Carlo simulation

- Recall: by risk-neutral pricing, the value of a contingent claim with payoff $X = \Phi(S_T)$ is given by

$$e^{-rT}\mathbb{E}^{\mathbb{Q}}[X],$$

  where $\mathbb{Q}$ is the risk-neutral measure.
- Under $\mathbb{Q}$, the bond and stock prices at time T are given by

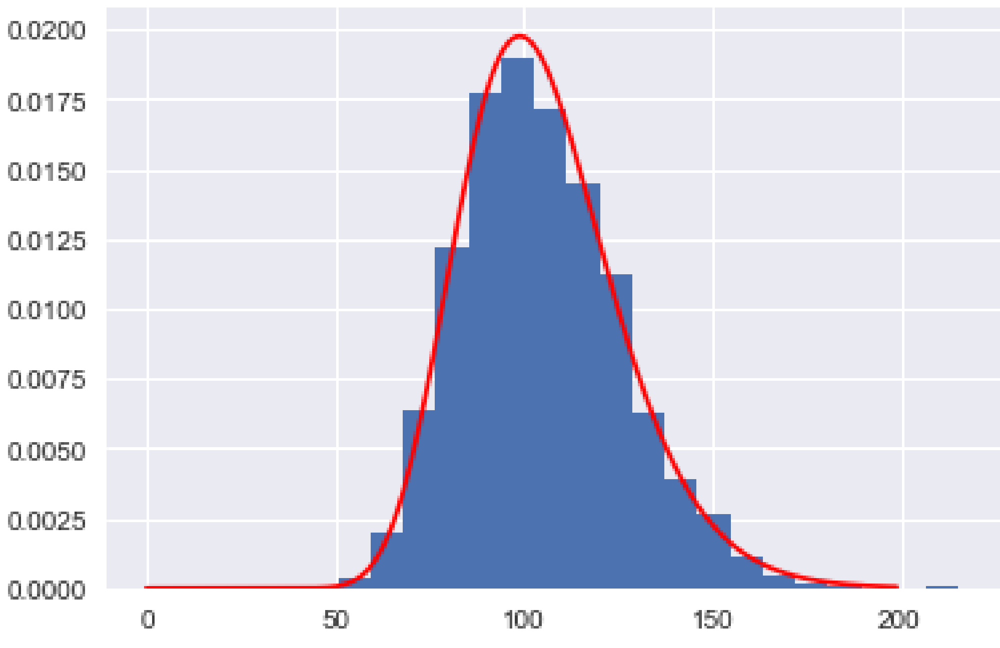
$$\begin{aligned} B_T &= B_0\, e^{rT} \\ S_T &= S_0\, \exp\left((r - 1/2\sigma^2)T + \sigma W_T\right), \end{aligned}$$

  with $W_T \sim \mathrm{N}(0, T)$.

Option pricing with Monte Carlo simulation

- The following graph shows a histogram of 5000 simulated stock prices S_T with parameters $S_0 = 100$, $\sigma = 0.2$, $T = 1$, $r = 0.05$.
- The red line is the density of S_T.

[20]:
```
x = np.arange(0,200,1)
w = npr.standard_normal(5000)
s = 100 * np.exp((0.05 - 0.5 * 0.2**2) * 1 + 0.2 * w)
plt.hist(s, bins=20,density=True);
plt.plot(x, scs.lognorm.pdf(x, s=0.2, scale=np.exp(np.log(100) + 0.05 - 0.5 * 0.
↪2**2)), 'r');
```



Option pricing with Monte Carlo simulation

- $e^{-rT}\mathbb{E}^{\mathbb{Q}}[X]$ is estimated using the following algorithm:
  - Generate uniformly distributed random numbers $u_1, \ldots, u_n$.
  - Transform them to normally distributed random numbers by applying the inverse standard normal distribution function: $\mathrm{N}^{(-1)}(u_1), \ldots, \mathrm{N}^{(-1)}(u_n)$.

- Set $S_{T,i} = S_0 \exp\Big((r - 1/2\sigma^2)T + \sigma\sqrt{T}\mathrm{N}^{(-1)}(u_i)\Big)$.
- Set $X_i = \Phi(S_{T,i})$.
- Set Simulated $\text{Price}_n = e^{-rT}(X_1 + \cdots + X_n)/n$.

Option pricing with Monte Carlo simulation

- For any $n \geq 1$, the estimated price is **unbiased**, that is,

$$\mathbb{E}^{\mathbb{Q}}[\text{Simulated Price}_n] = \text{Price} = e^{-rT}\mathbb{E}^{\mathbb{Q}}[X].$$

- The estimator is **strongly consistent**, that is,

$$\text{Simulated Price}_n \to \text{Price}, \text{ as } n \to \infty.$$

Option pricing with Monte Carlo simulation

- Example: Call option:
- We set $X = \Phi(S_T) = (S_T - K)^+$.
- The Black-Scholes model and option parameters are $S_0 = 100$, $K = 100$, $T = 1$, $\sigma = 0.2$, $r = 0.05$.
- The Black-Scholes call option price is given by Price $= 10.4506$.

Option pricing with Monte Carlo simulation

- Monte Carlo simulation code:

[21]:
```
w = npr.standard_normal(n)
s = 100 * np.exp((0.05 - 0.5 * 0.2**2) * 1 + 0.2 * w)
y = np.exp(-0.05*1) * np.maximum(s-100,0) # np.maximum is an element-wise maximum␣
↪operation
```

[22]:
```
y_m = []
y_cfl = []
y_cfu = []
for i in range(1000, n+1, 1000):
    y_m.append(np.mean(y[:i]))
    y_cfl.append(np.mean(y[:i] - 1.96 * np.std(y[:i])/np.sqrt(i)))
    y_cfu.append(np.mean(y[:i] + 1.96 * np.std(y[:i])/np.sqrt(i)))
```

Option pricing with Monte Carlo simulation

- Simulated option price (solid line) with 95% confidence intervals (green) and target (red line):

[23]:
```
plt.plot( [0,n], [10.4506, 10.4506], 'r', range(1000,n+1,1000), y_m, 'b', \
         range(1000,n+1,1000), y_cfl, 'g', range(1000,n+1,1000), y_cfu, 'g');
plt.xlabel('number of simulations');
plt.ylabel('estimates');
```

7.4 Path-dependent options

- Valuing path-dependent options requires simulating whole sample paths.
- Depending on the payoff a **discretisation error** is introduced leading to **bias** in the value of the option.

Path-dependent options

- Example: Asian option with discrete monitoring
- The payoff of an Asian option depends on the average level of the underlying asset, e.g.

$$\overline{S} = \frac{1}{m} \sum_{j=1}^{m} S_{t_j},$$

for some fixed dates $0 = t_0 < t_1 < \cdots < t_m = T$.
- Calculating $e^{-rT} \mathbb{E}^{\mathbb{Q}}[(\overline{S} - K)^+]$ requires samples of the average $\overline{S}$.
- This is achieved by simulating the path $S_{t_1}, \ldots, S_{t_m}$ via

$$S_{t_{j+1}} = S_{t_j} \exp\left((r - 1/2\sigma^2)(t_{j+1} - t_j) + \sigma\sqrt{t_{j+1} - t_j} Z_{j+1}\right),$$

where $Z_1, \ldots, Z_m$ are independent $\mathrm{N}(0, 1)$ random variables.

Asian option (cont'd)

- As a concrete example, consider the following setup:
  - $S_0 = 100$
  - $\sigma = 0.2$
  - $r = 0.05$
  - $K = 100$
  - $T = 1$
  - discrete monitoring; every month
- The payoff is thus

$$\left(\sum_{k=1}^{12} S_{k/12} - K \right)^+$$

Asian option (cont'd)

- Sample stock price paths monitored every month:

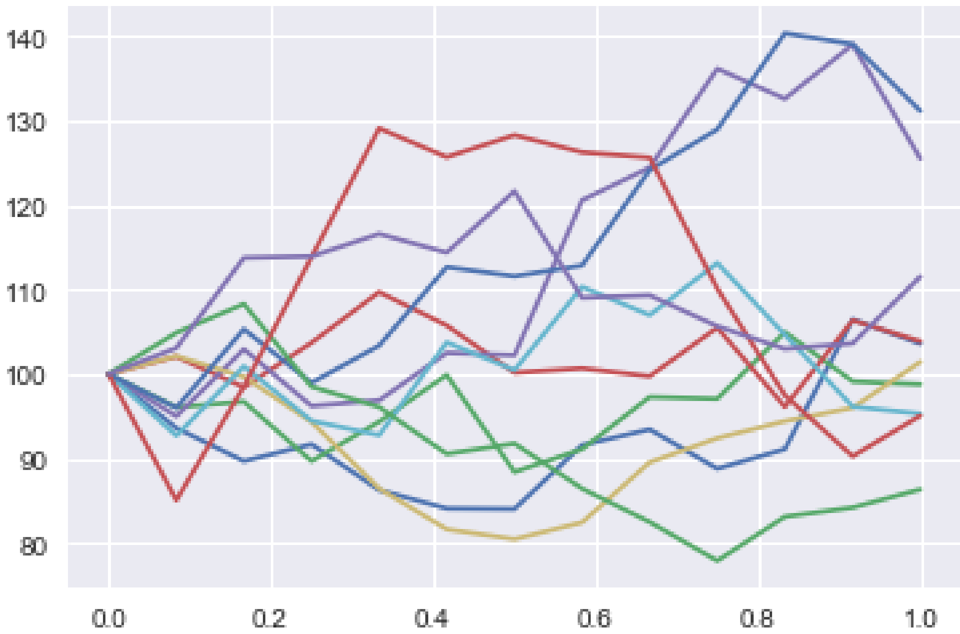
[24]:
```
m=12
t = np.arange(0,1+1/m, 1/m)
w = npr.standard_normal([n,m])
s = np.zeros([n,m+1])
s[:,0] = 100
```

[25]:
```
for i in range(1,m+1):
    dt = t[i]-t[i-1]
    s[:,i] = s[:,i-1] * np.exp((0.05-0.5 * 0.2**2) * dt + 0.2 * np.sqrt(dt) * w[:
↪,i-1])
```

Asian option (cont'd)

- Sample stock price paths monitored every month:

[26]:
```
plt.plot(t, np.transpose(s[0:10]));
```



Asian option (cont'd)

- Monte Carlo simulation code:

[27]:
```
ms=np.mean(s, axis=1)
y = list(map(lambda x: np.exp(-0.05*1) * np.max([x-100,0]), ms))
```

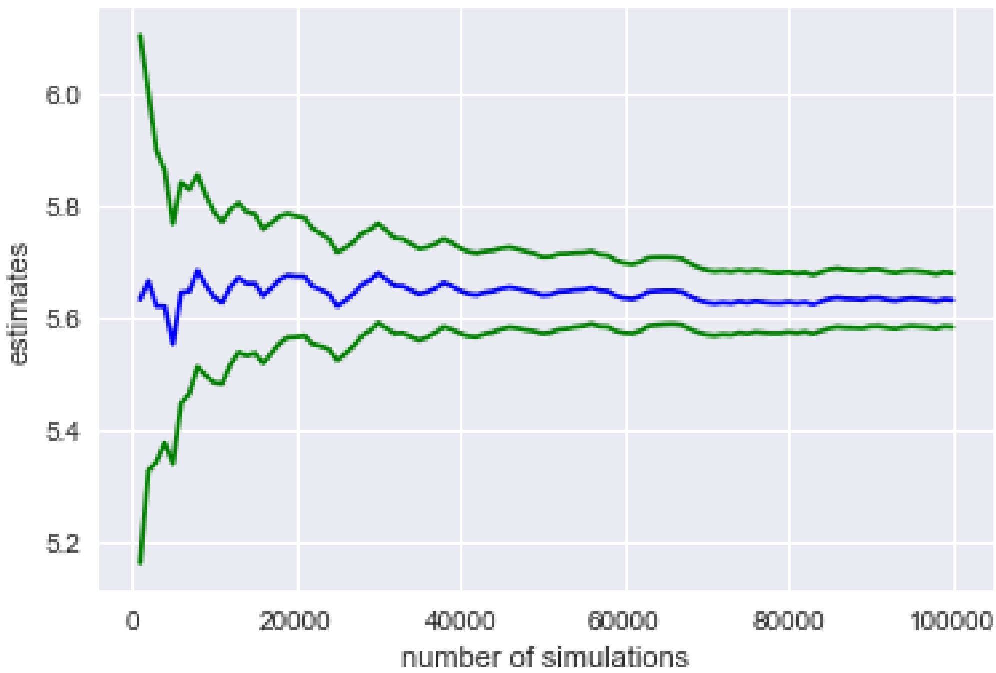
[28]:
```
y_m = []
y_cfl = []
y_cfu = []
for i in range(1000, n+1, 1000):
    y_m.append(np.mean(y[:i]))
```

```
    y_cfl.append(np.mean(y[:i] - 1.96 * np.std(y[:i])/np.sqrt(i)))
    y_cfu.append(np.mean(y[:i] + 1.96 * np.std(y[:i])/np.sqrt(i)))
```

Asian option (cont'd)

- Simulated option price (solid line) with 95% confidence intervals (green)

[29]:
```
plt.plot(range(1000,n+1,1000), y_m, 'b', \
         range(1000,n+1,1000), y_cfl, 'g', range(1000,n+1,1000), y_cfu, 'g');
plt.xlabel('number of simulations');
plt.ylabel('estimates');
```



Path-dependent options

- Now consider the following path-dependent payoffs:
  - **continuously monitored Asian option** with payoff

$$\left(\frac{1}{T}\int_0^T S_u\,du - K\right)^+$$

  - **lookback option** with payoff

$$\max_{0\le t\le T} S_t - S_T$$

- These options cannot be simulated exactly, but only with a discretisation error in the payoff.
- This introduces a bias in the estimated value.
- In the case of the lookback option, the discretised option value will almost surely underestimate the option value.