HW4 - Waldyr Faustini

April 21, 2021

1 Homework 4 FE621

1.0.1 Waldyr Faustini

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from tqdm.notebook import tqdm
from time import time
from scipy import stats


from numba import jit
```

2 Problem 1: Comparing Monte-Carlo schemes

2.1 1. Simple MC pricer

Observation: We will use numba to accelerate native numpy calculations, in order to allow for ~1 million or more Monte-Carlo samples.

We will use that the standard deviation of a sequence of iid variables $(X_1, \ldots, X_n)$ can be estimated as

$$s = \frac{1}{n-1}\left[\sum_{i=1}^{n} x_i^2 - \frac{1}{n}\left(\sum_{j=1}^{n} x_j\right)^2\right]$$

with the standard error then defined as $s/\sqrt{n}$.

[3]:
```
@jit(nopython=True)
def _vanilla_option(S0, K, r, div, sigma, T,
                        option_type='call',
                        n_steps=300,
                        n_sims=10000):

    # Use MC code in Figure 4.2 of the textbook
    dt = T/n_steps
    nudt = (r - div - 0.5*sigma**2)*dt
    sigsdt = sigma * np.sqrt(dt)
```

```
    sum_CT, sum_CT2 = 0, 0

    for _ in range(n_sims):
        lnS = np.log(S0)

        for i in range(n_steps):
            lnS += nudt + np.random.randn() * sigsdt

        ST = np.exp(lnS)
        if option_type == 'call':
            CT = max(0, ST-K)
        else:
            CT = max(0, K-ST)
        sum_CT += CT
        sum_CT2 += CT**2

    opt_value = np.exp(-r*T) * (sum_CT/n_sims)
    opt_std = np.exp(-r*T) * 1/(n_sims-1) * (sum_CT2 - (1/n_sims)* (sum_CT)**2)
    opt_se = opt_std/np.sqrt(n_sims)

    return opt_value, opt_se


def time_function(func):

    def timed(*args, **kwargs):
        t0 = time()
        opt_value, opt_se = func(*args, **kwargs)
        duration = time() - t0
        return opt_value, opt_se, duration
    return timed

vanilla_option = time_function(_vanilla_option)

# run once to compile
_, _ = _vanilla_option(S0=100.0, K=100.0, r=0.03, div=0.01, sigma=0.2, T=1.0,␣
 ↪n_sims=10, n_steps=10)

def bsm_value(S0, K, r, div, sigma, T, option_type):
    from scipy import stats

    phi = 1 if option_type == 'call' else -1
    N = lambda x: stats.norm.cdf(x, loc=0.0, scale=1.0)
    d1 = 1 if T == 0 else (np.log(S0/K) + (r - div + sigma**2/2)*T)/(sigma*np.
 ↪sqrt(T))
    d2 = d1 - sigma*np.sqrt(T)
```

```
    return phi*(np.exp(-div*T) * S0 * N(phi*d1) - K * np.exp(-r*T) * N(phi*d2))
```

Let us calculate these for: 300 and 700 time steps, for 1 and 5 million samples.

[43]:
```
res_list = []
for n_steps in tqdm([300, 700]):
    for n_sims in tqdm([x * 10**6 for x in  [3,5]]):
        price, err, duration = vanilla_option(S0=100.0, K=100.0, r=0.03, div=0.
 ↪01, sigma=0.2, T=1.0, n_sims=n_sims, n_steps=n_steps)
        res_list.append({'n_steps': n_steps, 'n_sims': n_sims, 'price': price,␣
 ↪'err': err, 'duration': duration})
```

```
HBox(children=(HTML(value=''), FloatProgress(value=0.0, max=2.0),␣
 ↪HTML(value='')))
```

```
HBox(children=(HTML(value=''), FloatProgress(value=0.0, max=2.0),␣
 ↪HTML(value='')))
```

```
HBox(children=(HTML(value=''), FloatProgress(value=0.0, max=2.0),␣
 ↪HTML(value='')))
```

Display results:

[44]:
```
pd.DataFrame(res_list).round(2)
```

[44]:

	n_steps	n_sims	price	err	duration
0	300	3000000	8.82	0.11	24.18
1	300	5000000	8.82	0.09	41.63
2	700	3000000	8.81	0.11	57.79
3	700	5000000	8.82	0.09	98.34

[37]:
```
real_price = bsm_value(S0=100, K=100, r=0.03, div=0.01, sigma=0.2, T=1.0,␣
 ↪option_type='call')
print("Actual (Black-Scholes) price: %.2f" % real_price)
```

```
Actual (Black-Scholes) price: 8.83
```

2.2 2. Variance reduction methods

2.2.1 Antithetic variables

The antithetic variables technique consists of creating two correlated estimators θ_1 and θ_2 for the same variable or parameter, with negative correlation, so that if we define

$$\theta := \frac{\theta_1 + \theta_2}{2}$$

then

$$\text{Var}(\theta) = \frac{1}{4}(\text{Var}(\theta_1) + \text{Var}(\theta_2) + 2\text{Cov}(\theta_1, \theta_2))$$

is smaller than the individual variance of either θ_1 or θ_2.

Below, we use the fact that if W_t is a Brownian motion, then so is $-W_t$ and thus we can use two GBMs based on these two factors to generate antithetic variables.

[4]:
```
@jit(nopython=True)
def _vanilla_option_anti(S0, K, r, div, sigma, T,
                         option_type='call',
                         n_steps=300,
                         n_sims=10000):

    # Use MC code in Figure 4.2 of the textbook
    dt = T/n_steps
    nudt = (r - div - 0.5*sigma**2)*dt
    sigsdt = sigma * np.sqrt(dt)

    sum_CT, sum_CT2 = 0, 0

    for _ in range(n_sims):
        lnS1 , lnS2 = np.log(S0), np.log(S0)

        for i in range(n_steps):
            eps = np.random.randn()
            lnS1 += nudt + eps * sigsdt
            lnS2 += nudt - eps * sigsdt

        ST1, ST2 = np.exp(lnS1), np.exp(lnS2)
        if option_type == 'call':
            CT = 0.5 * (max(0, ST1-K) + max(0, ST2-K))
        else:
            CT = 0.5 * (max(0, K-ST1) + max(0, K-ST2))
        sum_CT += CT
        sum_CT2 += CT**2

    opt_value = np.exp(-r*T) * (sum_CT/n_sims)
    opt_std = np.exp(-r*T) * 1/(n_sims-1) * (sum_CT2 - (1/n_sims)* (sum_CT)**2)
    opt_se = opt_std/np.sqrt(n_sims)

    return opt_value, opt_se

vanilla_option_anti = time_function(_vanilla_option_anti)

# run once to compile
_,_,_ = vanilla_option_anti(S0=100.0, K=100.0, r=0.03, div=0.01, sigma=0.2, T=1.
↪0, n_sims=10, n_steps=10)
```

2.2.2 Delta-based control variates

This method relies on the fact that a delta-hedged portfolio, when thought of as a random variable, has a much smaller variance than the option price itself. For a delta-based control variable, we emulate that procedure using a Black-Scholes delta.

Note: to make this efficient, instead of using the (Numba-incompatible) `scipy` implementation of the Gaussian CDF, we will use the Beasley-Springer-Moro algorithm as described in here: https://www.quantstart.com/articles/Statistical-Distributions-in-C/

[5]:
```
@jit(nopython=True)
def normal_cdf(x):
    k = 1.0/(1.0 + 0.2316419*x);
    k_sum = k*(0.319381530 + k*(-0.356563782 + k*(1.781477937 + k*(-1.821255978␣
 ↪+ 1.330274429*k))))

    if x >= 0:
        return (1.0 - (1.0/np.sqrt(2*np.pi))*np.exp(-0.5*x*x) * k_sum)
    else:
        return 1.0 - normal_cdf(-x)

@jit(nopython=True)
def bsm_delta(S0, K, r, div, sigma, T, option_type):

    phi = 1 if option_type == 'call' else -1
    d1 = 1 if T == 0 else (np.log(S0/K) + (r - div + sigma**2/2)*T)/(sigma*np.
 ↪sqrt(T))
    return phi*np.exp(-div*T)*normal_cdf(phi*d1)

@jit(nopython=True)
def _vanilla_option_deltacontrol(S0, K, r, div, sigma, T,
                     option_type='call',
                     n_steps=300,
                     n_sims=10000):

    # Use MC code in Figure 4.2 of the textbook
    dt = T/n_steps
    nudt = (r - div - 0.5*sigma**2)*dt
    sigsdt = sigma * np.sqrt(dt)
    erddt = np.exp((r-div)*dt)

    beta1 = -1

    sum_CT, sum_CT2 = 0, 0

    for _ in range(n_sims):
        St = S0
        cv = 0
```

```
        for i in range(n_steps):

            t = i*dt
            delta = bsm_delta(St, K, r, div, sigma, T, option_type)
            eps = np.random.randn()
            Stn = St*np.exp(nudt + eps * sigsdt)
            cv += delta*(Stn - St*erddt)
            St = Stn

        if option_type == 'call':
            CT = max(0, St-K) + beta1*cv
        else:
            CT = max(0, K-St) + beta1*cv
        sum_CT += CT
        sum_CT2 += CT**2

    opt_value = np.exp(-r*T) * (sum_CT/n_sims)
    opt_std = np.exp(-r*T) * 1/(n_sims-1) * (sum_CT2 - (1/n_sims)* (sum_CT)**2)
    opt_se = opt_std/np.sqrt(n_sims)

    return opt_value, opt_se

vanilla_option_deltacontrol = time_function(_vanilla_option_deltacontrol)
# run once to compile
_ = bsm_delta(S0=100.0, K=100.0, r=0.03, div=0.01, sigma=0.2, T=1.0,␣
 ↪option_type='call')
_,_,_ = vanilla_option_deltacontrol(S0=100.0, K=100.0, r=0.03, div=0.01,␣
 ↪sigma=0.2, T=1.0, n_sims=10, n_steps=10)
```

2.3 Adding both

[10]:
```
@jit(nopython=True)
def _vanilla_option_anti_deltacontrol(S0, K, r, div, sigma, T,
                      option_type='call',
                      n_steps=300,
                      n_sims=10000):

    # Use MC code in Figure 4.2 of the textbook
    dt = T/n_steps
    nudt = (r - div - 0.5*sigma**2)*dt
    sigsdt = sigma * np.sqrt(dt)
    erddt = np.exp((r-div)*dt)

    beta1 = -1

    sum_CT, sum_CT2 = 0, 0
```

```
    for _ in range(n_sims):
        St1, St2 = S0, S0
        cv1, cv2 = 0, 0
        for i in range(n_steps):

            t = i*dt
            delta1 = bsm_delta(St1, K, r, div, sigma, T, option_type)
            delta2 = bsm_delta(St2, K, r, div, sigma, T, option_type)

            eps = np.random.randn()

            Stn1 = St1*np.exp(nudt + eps * sigsdt)
            Stn2 = St2*np.exp(nudt - eps * sigsdt)
            cv1 += delta1*(Stn1 - St1*erddt)
            cv2 += delta2*(Stn2 - St2*erddt)

            St1 = Stn1
            St2 = Stn2

        if option_type == 'call':
            CT = 0.5*(max(0, St1-K) + beta1*cv1 + max(0, St2-K) + beta1*cv2)
        else:
            CT = 0.5*(max(0, K-St1) + beta1*cv1 + max(0, K-St2) + beta1*cv2)
        sum_CT += CT
        sum_CT2 += CT**2

    opt_value = np.exp(-r*T) * (sum_CT/n_sims)
    opt_std = np.exp(-r*T) * 1/(n_sims-1) * (sum_CT2 - (1/n_sims)* (sum_CT)**2)
    opt_se = opt_std/np.sqrt(n_sims)

    return opt_value, opt_se

vanilla_option_anti_deltacontrol =␣
 ↪time_function(_vanilla_option_anti_deltacontrol)
_,_,_ = vanilla_option_anti_deltacontrol(S0=100.0, K=100.0, r=0.03, div=0.01,␣
 ↪sigma=0.2, T=1.0, n_sims=10, n_steps=10)
```

Let us compare the methods using 500 steps and 100,000 MC samples:

[19]:
```
methods = ['MC', 'MC_anti', 'MC_delta', 'MC_both']
funcs  = [vanilla_option, vanilla_option_anti, vanilla_option_deltacontrol,␣
 ↪vanilla_option_anti_deltacontrol]
types = ['call', 'put']
res_list = []
for opt_type in types:
    for method, func in tqdm(zip(methods, funcs), total=len(methods)):
```

```
        price, err, duration = func(S0=100.0, K=100.0, r=0.03, div=0.01,␣
↪sigma=0.2, T=1.0,
                                    option_type=opt_type,
                                    n_sims=10**5, n_steps=500)
        res_list.append({'type': opt_type, 'method': method, 'price': price,␣
↪'err': err, 'duration_seconds': duration})
```

```
HBox(children=(HTML(value=''), FloatProgress(value=0.0, max=4.0),␣
↪HTML(value='')))
```

```
HBox(children=(HTML(value=''), FloatProgress(value=0.0, max=4.0),␣
↪HTML(value='')))
```

[20]:
```
aux = pd.DataFrame(res_list)
```

[23]:
```
# calls
aux[aux['type'] == 'call'].style.bar(subset=['err', 'duration_seconds'])
```

[23]: <pandas.io.formats.style.Styler at 0x24b1e3d1c48>

[24]:
```
# puts
aux[aux['type'] == 'put'].style.bar(subset=['err', 'duration_seconds'])
```

[24]: <pandas.io.formats.style.Styler at 0x24b1e2e8908>

In all cases, the error goes down by using more complex variance reduction methods. Since the amount of calculations involved also increases (especially in control methods, where the expensive calculation of the Black-Scholes delta comes in) the duration also goes up; using antithetic + delta-based control is almost 10x more expensive than naive MC.

3 Problem 2: path-dependent options

3.1 Arithmetic Asian call

We increment the baseline MC code with

- (1) a check whether the time step we current are is an observation date
- (2) add this to a running average

[34]:
```
@jit(nopython=True)
def asian_call(S0, K, r, div, sigma, T, n_steps=120, n_sims=10**6):

    # this ensures we observe by the end of every month, if n_steps=120
    observation_points = [10*i for i in range(1,13)]
```

```
    dt = T/n_steps
    nudt = (r - div - 0.5*sigma**2)*dt
    sigsdt = sigma * np.sqrt(dt)

    sum_CT, sum_CT2 = 0, 0

    for _ in range(n_sims):
        lnS = np.log(S0)
        avg_s = 0

        for i in range(1, n_steps+1):  # small modification to ensure we end at␣
↪120
            lnS += nudt + np.random.randn() * sigsdt

            if i in observation_points:
                avg_s += 1/12 * (np.exp(lnS))
        CT = max(0, avg_s-K)

        sum_CT += CT
        sum_CT2 += CT**2

    opt_value = np.exp(-r*T) * (sum_CT/n_sims)
    opt_std = np.exp(-r*T) * 1/(n_sims-1) * (sum_CT2 - (1/n_sims)* (sum_CT)**2)
    opt_se = opt_std/np.sqrt(n_sims)

    return opt_value, opt_se

_, _= asian_call(10.0, 10.0, 0.01, 0.0, 0.2, 1.0, 10, 10)
```

[35]:
```
%%time
price, err = asian_call(S0=100.0, K=100.0, r=0.03, div=0.01, sigma=0.2, T=1.0,␣
↪n_steps=120, n_sims=10**6)
```

```
Wall time: 5.47 s
```

[36]:
```
print(f"Asian option: price = {round(price,2)} +/- {round(err,2)}")
```

```
Asian option: price = 5.32 +/- 0.07
```

3.2 Up-and-out barrier call

[37]:
```
@jit(nopython=True)
def up_and_out_call(S0, K, r, div, sigma, T, H, n_steps=120, n_sims=10**6):

    # this ensures we observe by the end of every month, if n_steps=120
    observation_points = [10*i for i in range(1,13)]
```

```
    dt = T/n_steps
    nudt = (r - div - 0.5*sigma**2)*dt
    sigsdt = sigma * np.sqrt(dt)

    sum_CT, sum_CT2 = 0, 0

    for _ in range(n_sims):
        lnS = np.log(S0)
        knockout = False
        for i in range(1, n_steps+1):  # small modification to ensure we end at␣
↪120
            lnS += nudt + np.random.randn() * sigsdt

            if i in observation_points:
                checkpoint_s = np.exp(lnS)
                if checkpoint_s > H:
                    knockout = True
                    break

        if knockout:
            CT = 0.0
        else:
            ST = np.exp(lnS)
            CT = max(0, ST-K)

        sum_CT += CT
        sum_CT2 += CT**2

    opt_value = np.exp(-r*T) * (sum_CT/n_sims)
    opt_std = np.exp(-r*T) * 1/(n_sims-1) * (sum_CT2 - (1/n_sims)* (sum_CT)**2)
    opt_se = opt_std/np.sqrt(n_sims)

    return opt_value, opt_se

_, _= up_and_out_call(10.0, 10.0, 0.01, 0.0, 0.2, 1.0, 10, 10, 9.0)
```

[38]:
```
%%time
price, err = up_and_out_call(S0=100.0, K=100.0, r=0.03, div=0.01, sigma=0.2,␣
↪T=1.0, H=110.0, n_steps=120, n_sims=10**6)
```

```
Wall time: 4.14 s
```

[40]:
```
print(f"Up-and-out option: price = {round(price,3)} +/- {round(err,3)}")
```

```
Up-and-out option: price = 0.304 +/- 0.002
```

4 Problem 3: Correlated Brownian motion

4.1 a) Cholesky decomposition

[42]:
```
A = [[1.0, 0.5, 0.2],
     [0.5, 1.0, -0.4],
     [0.2, -0.4, 1.0]]
```

[44]:
```
L = np.linalg.cholesky(A)
```

[45]:
```
L
```

[45]:
```
array([[ 1.        ,  0.        ,  0.        ],
       [ 0.5       ,  0.8660254 ,  0.        ],
       [ 0.2       , -0.57735027,  0.79162281]])
```

[52]:
```
# Prove that this works: L * L^T = A
np.dot(L, (L.transpose()))
```

[52]:
```
array([[ 1. ,  0.5,  0.2],
       [ 0.5,  1. , -0.4],
       [ 0.2, -0.4,  1. ]])
```

4.2 b) Generate paths for assets

Let us use that if z is an n-dimensional independent Gaussian variate, and if L is the Cholesky decomposition of a matrix A, then Lz is a new Gaussian variate with covariance matrix equal to A.

[280]:
```
mu = [0.03, 0.06, 0.02]
sigma = [0.05, 0.2, 0.15]
S0 = [100.0, 101.0, 98.0]
```

[329]:
```
def correlated_gbm_path_3vars(S0_vec, mu_vec, sigma_vec, corr_matrix, T,␣
 ↪n_steps=1000, n_sims=1000):

    dt = T/n_steps
    result = []
    assert set([len(S0_vec), len(mu_vec), len(sigma_vec), len(A)]) == {3}

    chol = np.linalg.cholesky(corr_matrix)

    for i in range(n_sims):
        X, Y, Z = S0_vec
        inter_res = []
        for j in range(n_steps):
            if j > 0:
```

```
                indep_gauss = np.random.randn(3)
                dW = np.dot(chol, indep_gauss)*np.sqrt(dt)

                dX = X*(mu_vec[0]*dt + sigma_vec[0]*dW[0])
                dY = Y*(mu_vec[1]*dt + sigma_vec[1]*dW[1])
                dZ = Y*(mu_vec[2]*dt + sigma_vec[2]*dW[2])

                X += dX
                Y += dY
                Z += dZ
            else:
                pass # for t=0, just add the initial conditions for X, Y and Z
            inter_res.append([X, Y, Z])
        result.append(inter_res)
    return np.array(result)
```

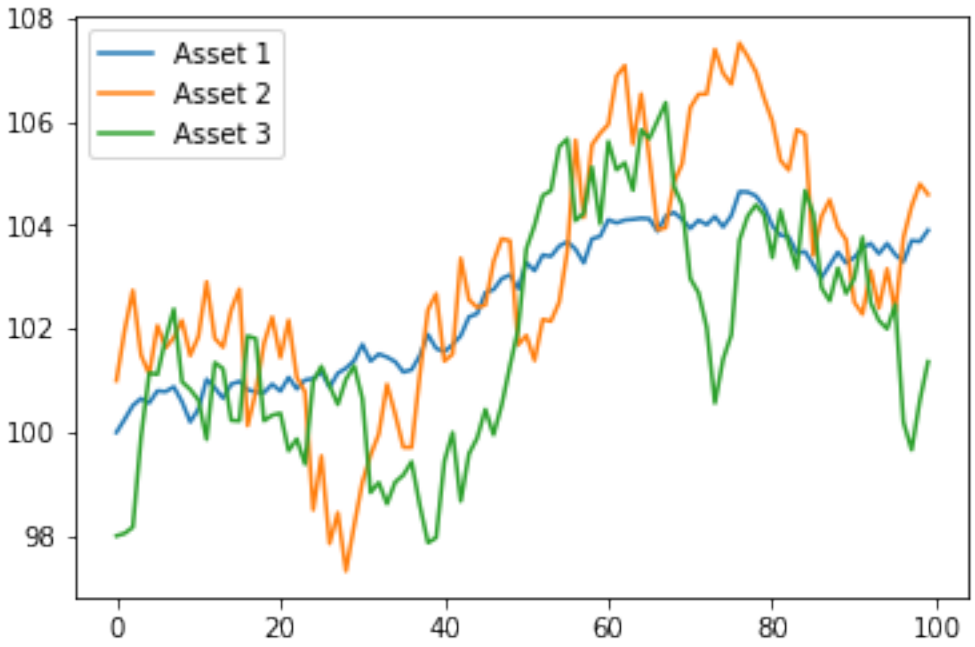
[330]:
```
res = correlated_gbm_path_3vars(S0, mu, sigma, A, T=100/365, n_steps=100,␣
↪n_sims=1000)
```

[331]:
```
res.shape
```

[331]:
```
(1000, 100, 3)
```

4.2.1 Sample plot:

[333]:
```
plt.plot(range(100), res[0][:,0], label='Asset 1')
plt.plot(range(100), res[0][:,1], label='Asset 2')
plt.plot(range(100), res[0][:,2], label='Asset 3')
plt.legend()
plt.show()
```



4.3 c) Price basket option

We will leverage the code from the previous item, adding a payoff in the end.

A few very important considerations:

- We have to **define a risk-free rate** r to use, since it is not specified in the problem statement: let us use $r = 1\%$.
- Assuming we can **change into a risk-neutral measure**, all drifts become exactly r
- Correlations between Brownian motions are not affected by a change of measure, since if $dW_1 dW_2 = \rho dt$, then $d\tilde{W}_i = dW_i + \theta_1 dt$ for $i = 1, 2$, which are required by Girsanov's theorem, will also satisfy $d\tilde{W}_1 d\tilde{W}_2 = \rho dt$
- Because of this, the risk-neutral pricing formula works

[359]:
```
def price_basket_option_3unds(S0_vec, r, sigma_vec, corr_matrix, K, T, weights,␣
↪option_type='call',
                              n_steps=1000, n_sims=1000):

    dt = T/n_steps
    result = []
    assert set([len(S0_vec), len(sigma_vec), len(A), len(weights)]) == {3}

    chol = np.linalg.cholesky(corr_matrix)
    sum_CT, sum_CT2 = 0, 0

    for i in range(n_sims):
```

```
        X, Y, Z = S0_vec
        inter_res = []
        for j in range(n_steps):
            if j > 0:

                indep_gauss = np.random.randn(3)
                dW = np.dot(chol, indep_gauss)*np.sqrt(dt)

                dX = X*(r*dt + sigma_vec[0]*dW[0])
                dY = Y*(r*dt + sigma_vec[1]*dW[1])
                dZ = Y*(r*dt + sigma_vec[2]*dW[2])

                X += dX
                Y += dY
                Z += dZ
            else:
                pass # for t=0, just add the initial conditions for X, Y and Z

        # payoff
        ST = weights[0]*X + weights[1]*Y + weights[2]*Z
        if option_type == 'call':
            CT = max(ST-K, 0)
        else:
            CT = max(K-ST, 0)

        sum_CT += CT
        sum_CT2 += CT**2

    opt_value = np.exp(-r*T) * (sum_CT/n_sims)
    opt_std = np.exp(-r*T) * 1/(n_sims-1) * (sum_CT2 - (1/n_sims)* (sum_CT)**2)
    opt_se = opt_std/np.sqrt(n_sims)

    return opt_value, opt_se
```

[360]:
```
call_price, err = price_basket_option_3unds(S0_vec=S0, r=0.01, sigma_vec=sigma,␣
↪corr_matrix=A, K=100.0,
                                            T=100/365, weights=[1/3,1/3,1/3],
                                            option_type='call', n_steps=100,␣
↪n_sims=10000)
print("Call price with r = 0.01 is %.2f" % call_price)
```

```
Call price with r = 0.01 is 1.60
```

[361]:
```
put_price, err = price_basket_option_3unds(S0_vec=S0, r=0.01, sigma_vec=sigma,␣
↪corr_matrix=A, K=100.0,
                                            T=100/365, weights=[1/3,1/3,1/3],
```

```
                                        option_type='put', n_steps=100,␣
↪n_sims=10000)
print("Put price with r = 0.01 is %.2f" % put_price)
```

```
Put price with r = 0.01 is 1.64
```

Checking put-call parity: $C(t) - P(T) = S(t) - Ke^{-r(T-t)}$ (which works identically as the single-underlying case)

[366]:
```
round(call_price-put_price,2)
```

[366]:
```
-0.04
```

[367]:
```
round(np.dot(S0, [1/3,1/3,1/3]) - 100*np.exp(-0.01*(100/365)),2)
```

[367]:
```
-0.06
```

This seems to be close enough.

4.4 d) Exotic payoff

We *assume there was a typo* in condition (iii), which says "average" but only shows a sum.

[376]:
```
def price_basket_option_3unds_exotic(S0_vec, r, sigma_vec, corr_matrix, K, B,␣
↪T, weights,
                                n_steps=1000, n_sims=1000):

    def payoff1(X, Y, Z, avgY):
        return max(Y-K, 0)

    def payoff2(X, Y, Z, avgY):
        return max(Y**2-K, 0)

    def payoff3(X, Y, Z, avgY):
        return max(avgY-K, 0)

    def payoff4(X, Y, Z, avgY):
        ST = weights[0]*X + weights[1]*Y + weights[2]*Z
        return max(ST-K, 0)

    # start with the baseline payoff for the basket
    payoff = payoff4

    dt = T/n_steps
    result = []
    assert set([len(S0_vec), len(sigma_vec), len(A), len(weights)]) == {3}

    chol = np.linalg.cholesky(corr_matrix)
```

```
    sum_CT, sum_CT2 = 0, 0

    for i in range(n_sims):
        X, Y, Z = S0_vec
        avgY, avgZ = 0, 0
        maxY, maxZ = 0, 0

        inter_res = []
        can_change_payoff = True  # add this so that the barrier condition␣
↪"turns off" other payoffs if satisfied
        for j in range(n_steps):
            if j > 0:

                indep_gauss = np.random.randn(3)
                dW = np.dot(chol, indep_gauss)*np.sqrt(dt)

                dX = X*(r*dt + sigma_vec[0]*dW[0])
                dY = Y*(r*dt + sigma_vec[1]*dW[1])
                dZ = Y*(r*dt + sigma_vec[2]*dW[2])

                # updates (now include averages and maxs)
                X += dX
                Y += dY
                Z += dZ

                avgY += 1/n_steps * Y
                avgZ += 1/n_steps * Z
                maxY = max(maxY, Y)
                maxZ = max(maxZ, Z)

                # checks for payoffs
                if Y > B:
                    payoff = payoff1
                    can_change_payoff = False
            else:
                pass # for t=0, just add the initial conditions for X, Y and Z

        if can_change_payoff:
            if (maxY > maxZ):
                payoff = payoff2
            elif avgY > avgZ:
                payoff = payoff3
            else:
                payoff = payoff4

        CT = payoff(X, Y, Z, avgY)
```

```
        sum_CT += CT
        sum_CT2 += CT**2

    opt_value = np.exp(-r*T) * (sum_CT/n_sims)
    opt_std = np.exp(-r*T) * 1/(n_sims-1) * (sum_CT2 - (1/n_sims)* (sum_CT)**2)
    opt_se = opt_std/np.sqrt(n_sims)

    return opt_value, opt_se
```

[377]:
```
price, err = price_basket_option_3unds_exotic(S0_vec=S0, r=0.01,␣
 ↪sigma_vec=sigma, corr_matrix=A, K=100.0,
                                        B=104, T=100/365, weights=[1/3,1/3,1/3],
                                        n_steps=100, n_sims=20000)
print("Price for exotic option with r = 0.01 is %.2f" % price)
```

```
Price for exotic option with r = 0.01 is 652.43
```

5 Problem 4 (BONUS): Simulating the Heston model

5.1 Simulating the Euler schemes

Here we consider the Heston model

$$\frac{dS_t}{S_t} = rdt + \sqrt{V_t}dW_t$$

$$dV_t = -\kappa(V_t - \theta_t)dt + \sigma\sqrt{V_t}dZ_t$$

with $dW_t dZ_t = \rho dt$. We have set $\lambda = 1$ in the paper's notation, set the drift to be the risk-free rate, and set the vol of vol to σ.

The generalized framework presented by the paper for (Euler) simulating the Heston model is:

$$\begin{cases} \ln S_{t+\Delta t} & = \ln S_t + \left(r - \frac{1}{2}V_t\right)\Delta t + \sqrt{V_t}\Delta W_t \\ \tilde{V}_{t+\Delta t} & = f_1(\tilde{V}_t) - \kappa\Delta t(f_2(\tilde{V}(t) - \bar{V}) + \sigma\sqrt{f_3(\tilde{V}_t)}\Delta Z_t \\ V_{t+\Delta t} & = f_3(\tilde{V}_{t+\Delta t}) \end{cases}$$

with $V(0) = \tilde{V}(0)$, and $\Delta W_t, \Delta Z_t$ must be simulated from a correlated Gaussian pair. Effectively,

$$\begin{cases} \Delta Z_t & = X_1 \\ \Delta W_t & = \rho X_1 + \sqrt{1-\rho^2}X_2 \end{cases}$$

if X_1, X_2 are independent normal variates $\mathcal{N}(0,1)$. The shapes of the functions f_1, f_2 and f_3 depends on the choice of the model.

In the above, we will also take $\bar{V}$ to be the asymptotic mean θ of the volatility.

Let C_i be the simulated option price for a given MC sample i, and let C_0 be the benchmark price. We will define

$$\text{bias} := \left| \frac{1}{N} \sum_{i=1}^{N} C_i - C_0 \right|$$

and RMSE to be the standard error.

We will simulate with 10,000 samples and 1000 time steps.

[79]:
```
@jit(nopython=True)
def _heston_call(S0, K, r, V0, kappa, theta, sigma, rho, T, n_steps, n_sims,␣
 ↪model=None):

    C_orig = 6.8061
    identity = lambda x: x
    positive = lambda x: max(x,0)
    module   = lambda x: abs(x)

    if model == 'absorption':
        f1, f2, f3 = positive, positive, positive
    elif model == 'reflection':
        f1, f2, f3 = module, module, module
    elif model == 'higham_mao':
        f1, f2, f3 = identity, identity, module
    elif model == 'partial_truncation':
        f1, f2, f3 = identity, identity, positive
    elif model == 'full_truncation':
        f1, f2, f3 = identity, positive, positive
    else:
        raise Exception("Invalid model")

    dt = T/n_steps
    sum_CT, sum_CT2, mse = 0, 0, 0

    for _ in range(n_sims):
        lnS = np.log(S0)
        V = V0
        Vtilde = V0

        for i in range(n_steps):

            # correlated Gaussians
            X1 = np.random.randn()
            X2 = rho*X1 + np.sqrt(1-rho**2)*np.random.randn()

            # Brownians dZ, dW
            dZ = np.sqrt(dt)*X1
            dW = np.sqrt(dt)*X2
```

```
            # evolving the vol and spot
            lnS = lnS + (r - V/2)*dt + np.sqrt(V)*dW
            Vtilde = f1(Vtilde) - kappa * dt * (f2(Vtilde) - theta) + sigma*np.
 ↪sqrt(f3(Vtilde)) * dZ
            V = f3(Vtilde)

        ST = np.exp(lnS)

        # payoff
        CT = max(0, ST-K)

        sum_CT += CT
        sum_CT2 += CT**2
        mse += (CT - C_orig)**2

    opt_value = np.exp(-r*T) * (sum_CT/n_sims)
    opt_std = np.exp(-r*T) * 1/(n_sims-1) * (sum_CT2 - (1/n_sims)* (sum_CT)**2)
    opt_se = opt_std/np.sqrt(n_sims)
    bias = abs(C_orig - opt_value)

    return opt_value, opt_se, bias

def time_function2(func):

    def timed(*args, **kwargs):
        t0 = time()
        opt_value, opt_se, bias = func(*args, **kwargs)
        duration = time() - t0
        return opt_value, opt_se, bias, duration
    return timed


heston_call = time_function2(_heston_call)
```

[80]:
```
results = []
for model in tqdm(['absorption', 'reflection', 'higham_mao',␣
 ↪'partial_truncation', 'full_truncation']):
    price, rmse, bias, duration = heston_call(S0=100.0, K=100.0, r=0.0319, V0=0.
 ↪010201, kappa=6.21, theta=0.019, sigma=0.61, rho=-0.7, T=1.0,
                n_steps=1000, n_sims=10**5, model=model)
    results.append({'model': model, 'price': price, 'bias': bias, 'RMSE': rmse,␣
 ↪'duration': duration})
```

```
HBox(children=(HTML(value=''), FloatProgress(value=0.0, max=5.0),␣
 ↪HTML(value='')))
```

```
<ipython-input-79-f744c05763a4>:30: NumbaExperimentalFeatureWarning:
First-class function type feature is experimental
```

```
    for i in range(n_steps):
```

[81]:
```
pd.DataFrame(results)
```

[81]:

	model	price	bias	RMSE	duration
0	absorption	6.829784	0.023684	0.182833	8.131362
1	reflection	6.863025	0.056925	0.187439	6.595043
2	higham_mao	6.807183	0.001083	0.180745	6.624242
3	partial_truncation	6.815706	0.009606	0.179412	6.662453
4	full_truncation	6.817356	0.011256	0.179994	6.749154

As we can see, these are all close to the original price and very much within the standard error.

5.2 Solving by quadrature

The goal is to directly compute the price from Heston's formula,

$$C(S_0, K, V_0, \tau = T - t) = S_0 P_1 - K e^{-(r-q)\tau} P_2$$

using the inverse Fourier transform

$$P_j = \frac{1}{2} + \frac{1}{\pi} \int_0^\infty \mathrm{Re}\left[\frac{e^{-iu \ln K} \varphi_j(S_0, V_0, \tau, u)}{iu} \right] du$$

where we have explicit formulas for φ_1, φ_2.

Start with a Simpson integrator:

[260]:
```
@jit(nopython=True) # can only jit if integrand is also jitted
def simpson_integrator(f, a, b, max_error=1e-3):

    # start with 50000 partitions and increase that number as long as
    # error is greater than tolerance
    n_partitions = 0
    res_new, res_old = 0, np.inf

    while abs(res_old-res_new) > max_error:
        n_partitions += 10000
        dx = (b-a)/n_partitions
        full_range = np.linspace(a, b, n_partitions+1)
        fx = np.array([f(x) for x in full_range])

        res_old = res_new
        res_new = (dx/3)*(fx[0] + fx[-1] + 4*fx[1:-1][::2].sum()+2 * fx[1:-1][1:
↪:2].sum())

    return res_new
```

```
@jit(nopython=True)
def mysquare(x):
    return x**2

@jit(nopython=True)
def mycube(x):
    return x**3

assert np.isclose(simpson_integrator(mysquare, 1, 10), 333), "Integrator not␣
 ↪working"
assert np.isclose(simpson_integrator(mycube, 1, 5), 624/4), "Integrator not␣
 ↪working"
```

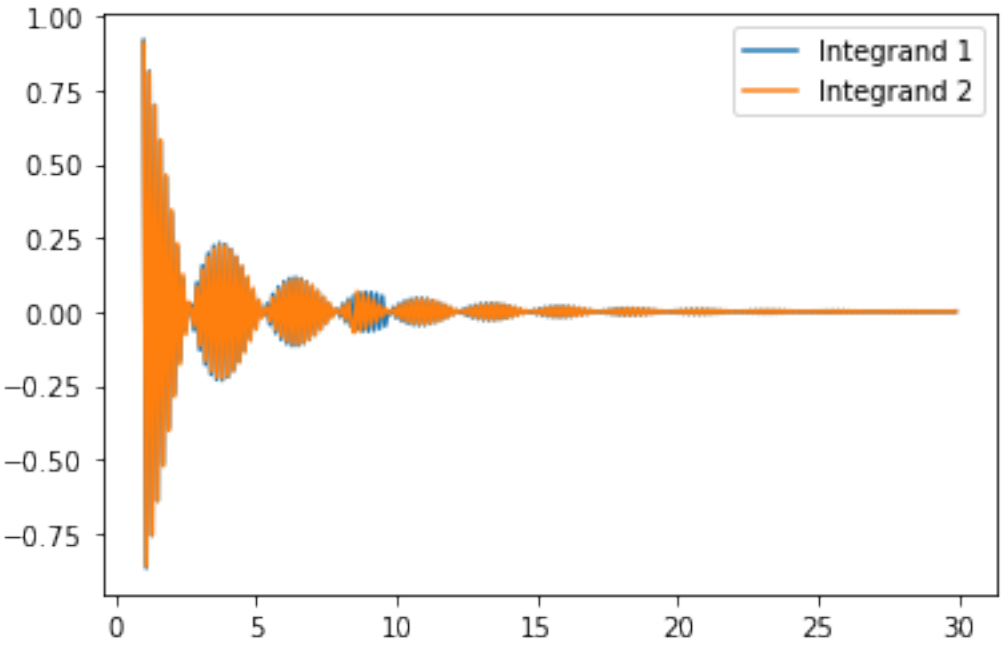
Now we implement the characteristic functions, using that the market price of risk (λ) is zero:

[261]:
```
S0=100.0
K=100.0
r=0.0319
V0=0.010201
kappa=6.21
theta=0.019
sigma=0.61
rho=-0.7
T=1.0
```

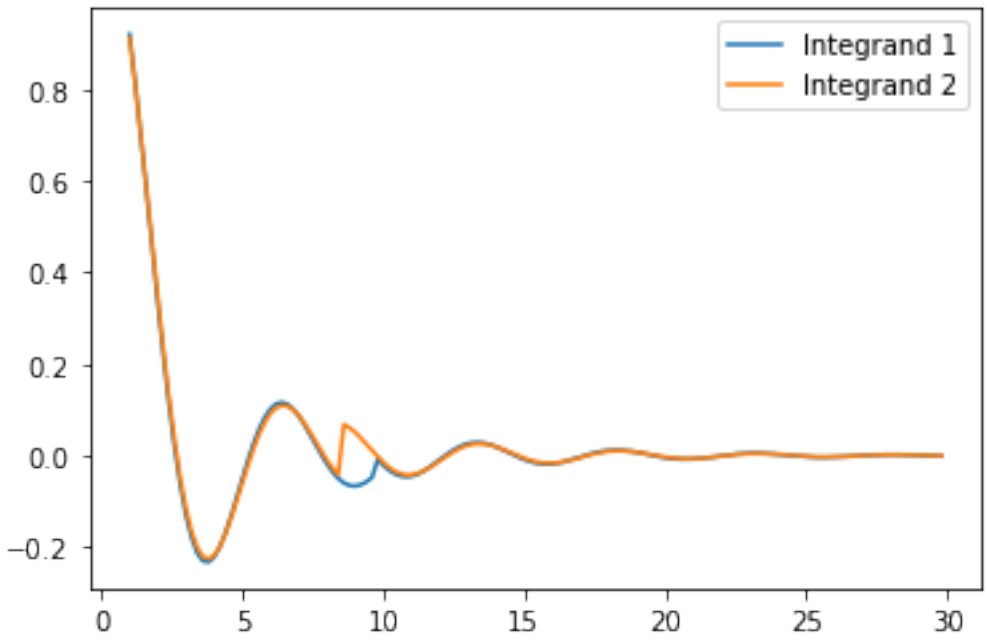
[262]:
```
@jit(nopython=True)
def charac_func(i: int, phi: float, S0: float, K:float, r:float, V0:float,␣
 ↪kappa: float, theta:float, sigma:float, rho:float, tau:float):

    j = np.complex(0,1)

    assert i in [1,2]
    b = kappa-rho*sigma*(i==1)
    u = 0.5 - 1*(i==2)

    d = np.sqrt((rho*sigma*phi*j - b)**2 - sigma**2 * (2*u*phi*j - phi**2))
    g = (b - rho*sigma*phi*j + d)/(b - rho*sigma*phi*j - d)

    C = (r)*phi*j*tau + kappa*theta/(sigma**2)*((b - rho*sigma*phi*j + d)*tau -␣
 ↪2*np.log((1-g * np.exp(d*tau))/(1-g)))
    D = (b - rho*sigma*phi*j + d)/(sigma**2) *((1-np.exp(d*tau))/(1-g*np.
 ↪exp(d*tau)))

    return np.exp(C + V0 * D + j*phi*S0)
```

[263]:
```
@jit(nopython=True)
def integrand1(u):

    j = np.complex(0,1)
    return (np.exp(-j*u*np.log(K))*charac_func(i=1, phi=u, S0=S0, K=K, r=r,␣
 ↪V0=V0, kappa=kappa, theta=theta,
                                           sigma=sigma, rho=rho, tau=T)/(j*u)).real

@jit(nopython=True)
def integrand2(u):

    j = np.complex(0,1)
    return (np.exp(-j*u*np.log(K))*charac_func(i=2, phi=u, S0=S0, K=K, r=r,␣
 ↪V0=V0, kappa=kappa, theta=theta,
                                           sigma=sigma, rho=rho, tau=T)/(j*u)).real

_ = integrand1(0.3)
_ = integrand2(0.3)
```

[275]:
```
# To understand how big the integration domain needs to be, let us plot both␣
 ↪integrands
# with different frequencies:
x = np.arange(1,30,0.1)
plt.plot(x, [integrand1(xx) for xx in x], label='Integrand 1')
plt.plot(x, [integrand2(xx) for xx in x], label='Integrand 2')
plt.legend()
plt.show()
```



[267]:
```
x = np.arange(1,30,0.20)
plt.plot(x, [integrand1(xx) for xx in x], label='Integrand 1')
plt.plot(x, [integrand2(xx) for xx in x], label='Integrand 2')
plt.legend()
plt.show()
```



[258]:
```
P1 = 0.5 + (1/np.pi) * simpson_integrator(f=integrand1, a=0.0001, b=25,␣
 ↪max_error=1e-4)
P2 = 0.5 + (1/np.pi) * simpson_integrator(f=integrand2, a=0.0001, b=25,␣
 ↪max_error=1e-4)
price = S0*P1 - K*np.exp(-r*T)*P2
```

[259]:
```
print("Price: %.2f" % price)
```

```
Price: 3.18
```

5.3 Comparison

Be it by numerical issues when computing the complex logarithm, be it due to how the implementation works, the quadrature method is less precise than the Euler method. Clearly the integrand oscillates extremely fast, so that there might be issues with the oscillatory integral as well.

Notice that the Feller condition $2\kappa\theta > \sigma^2$ is **not** satisfied in this instance, too, and this is also known to lead to instabilities (see eg. https://perswww.kuleuven.be/~u0009713/HestonTrap.pdf)