Chia-Wei Lai | clai2

Simulation Methods

46-773

Homework #5

In [ ]:
```python
import numpy as np
import scipy.stats as stats
```

1) **Replicating Broadie and Glasserman "Greeks" methodology.**

In [ ]:
```python
# Parameters
S0 = 100
K = 100
r = 0.1
q = 0.03
sigma = 0.25
T = 0.2
n = 10000
h = 0.0001

# Payoff function
def payoff(ST, K):
    return np.maximum(ST - K, 0)

# Delta estimates
def resimulation_delta(S0, h, K, T, r, q, sigma, n, control = False):
    # np.random.seed(42)
    z = np.random.normal(size=n)

    ST_up = (S0 + h) * np.exp((r - q - 0.5 * sigma**2) * T + sigma * np.sqrt(T) * z)
    ST_down = (S0 - h) * np.exp((r - q - 0.5 * sigma**2) * T + sigma * np.sqrt(T) * z)

    mc_delta = np.exp(-r * T) * (payoff(ST_up, K) - payoff(ST_down, K)) / (2 * h)

    if control == False:
        delta = np.mean(mc_delta)
        delta_std = np.std(mc_delta, ddof = 1) / np.sqrt(n)
    else:
        final_price = S0 * np.exp((r - q - 0.5 * sigma**2) * T + sigma * np.sqrt(T) * z)

        cov = np.cov(final_price, mc_delta, ddof = 1)
        a_hat = cov[0, 1] / cov[0, 0]
        control_var = mc_delta - a_hat * (final_price - S0 * np.exp((r - q)* T))

        delta = np.mean(control_var)
        delta_std = np.std(control_var, ddof = 1) / np.sqrt(n)

    return delta, delta_std

def pathwise_delta(S0, K, T, r, q, sigma, n, control = False):
    z = np.random.normal(size=n)
    final_price = S0 * np.exp((r - q - 0.5 * sigma**2) * T + sigma * np.sqrt(T) * z)
    pw_delta = np.exp(-r * T) * (final_price > K) * final_price / S0

    if control == False:
        delta = np.mean(pw_delta)
        delta_std = np.std(pw_delta, ddof = 1) / np.sqrt(n)
    else:
        cov = np.cov(final_price, pw_delta, ddof = 1)
        a_hat = cov[0, 1] / cov[0, 0]
        control_var = pw_delta - a_hat * (final_price - S0 * np.exp((r - q)* T))

        delta = np.mean(control_var)
        delta_std = np.std(control_var, ddof = 1) / np.sqrt(n)
    return  delta, delta_std

def likelihood_delta(S0, K, T, r, q, sigma, n, control = False):
    z = np.random.normal(size=n)
    final_price = S0 * np.exp((r - q - 0.5 * sigma**2) * T + sigma * np.sqrt(T) * z)
    ll_delta = np.exp(-r * T) * payoff(final_price, K) * (1 / (S0 * sigma**2 * T)) * (np.log(final_price / S0) - (r - q - 0.5 * sigma**2) * T)

    if control == False:
        delta = np.mean(ll_delta)
        delta_std = np.std(ll_delta, ddof = 1) / np.sqrt(n)
    else:
        cov = np.cov(final_price, ll_delta, ddof = 1)
        a_hat = cov[0, 1] / cov[0, 0]
        control_var = ll_delta - a_hat * (final_price - S0 * np.exp((r - q)* T))

        delta = np.mean(control_var)
        delta_std = np.std(control_var, ddof = 1) / np.sqrt(n)
    return  delta, delta_std
```

In [ ]:
```python
def resimulation_vega(S0, h, K, T, r, q, sigma, n, control = False):
    # np.random.seed(42)
    z = np.random.normal(size=n)

    ST_up = S0 * np.exp((r - q - 0.5 * (sigma + h)**2) * T + (sigma + h) * np.sqrt(T) * z)
    ST_down = S0 * np.exp((r - q - 0.5 * (sigma - h)**2) * T + (sigma - h) * np.sqrt(T) * z)

    mc_vega = np.exp(-r * T) * (payoff(ST_up, K) - payoff(ST_down, K)) / (2 * h)

    if control == False:
        vega = np.mean(mc_vega)
        vega_std = np.std(mc_vega, ddof = 1) / np.sqrt(n)
    else:
        final_price = S0 * np.exp((r - q - 0.5 * sigma**2) * T + sigma * np.sqrt(T) * z)

        cov = np.cov(final_price, mc_vega, ddof = 1)
        a_hat = cov[0, 1] / cov[0, 0]
        control_var = mc_vega - a_hat * (final_price - S0 * np.exp((r - q)* T))

        vega = np.mean(control_var)
        vega_std = np.std(control_var, ddof = 1) / np.sqrt(n)

    return vega, vega_std

def pathwise_vega(S0, K, T, r, q, sigma, n, control = False):
    z = np.random.normal(size=n)
    final_price = S0 * np.exp((r - q - 0.5 * sigma**2) * T + sigma * np.sqrt(T) * z)
    pw_vega = np.exp(-r * T) * (final_price > K) * final_price / sigma * (np.log(final_price / S0) - (r - q + 0.5 * sigma**2) * T)

    if control == False:
        vega = np.mean(pw_vega)
        vega_std = np.std(pw_vega, ddof = 1) / np.sqrt(n)
    else:
        cov = np.cov(final_price, pw_vega, ddof = 1)
        a_hat = cov[0, 1] / cov[0, 0]
        control_var = pw_vega - a_hat * (final_price - S0 * np.exp((r - q)* T))
```

```python
        vega = np.mean(control_var)
        vega_std = np.std(control_var, ddof = 1) / np.sqrt(n)
    return  vega, vega_std

def likelihood_vega(S0, K, T, r, q, sigma, n, control = False):
    z = np.random.normal(size=n)
    final_price = S0 * np.exp((r - q - 0.5 * sigma**2) * T + sigma * np.sqrt(T) * z)

    d = (np.log(final_price / S0) - (r - q - 0.5 * sigma**2) * T) / (sigma * np.sqrt(T))
    ll_vega = np.exp(-r * T) * payoff(final_price, K) * (- d * ((np.log(S0 / final_price) + (r - q + 0.5 * sigma**2) * T) / (sigma**2 * np.sqrt(T))) - 1 / sigma)

    if control == False:
        vega = np.mean(ll_vega)
        vega_std = np.std(ll_vega, ddof = 1) / np.sqrt(n)
    else:
        cov = np.cov(final_price, ll_vega, ddof = 1)
        a_hat = cov[0, 1] / cov[0, 0]
        control_var = ll_vega - a_hat * (final_price - S0 * np.exp((r - q)* T))

        vega = np.mean(control_var)
        vega_std = np.std(control_var, ddof = 1) / np.sqrt(n)
    return  vega, vega_std
```

In [ ]:

```python
estDelta, deltaStd =  resimulation_delta(S0, h, K, T, r, q, sigma, n, control = False)
estVega, vegaStd = resimulation_vega(S0, h, K, T, r, q, sigma, n, control = False)
print(f"Resimulation estimate | Delta Est = {estDelta:.4f}, Delta Std Err = {deltaStd:.4f}, Vega Est = {estVega:.4f}, Vega Ste Err = {vegaStd:.4f}")

estDelta, deltaStd =  resimulation_delta(S0, h, K, T, r, q, sigma, n, control = True)
estVega, vegaStd = resimulation_vega(S0, h, K, T, r, q, sigma, n, control = True)
print(f"Resimulation estimate with control | Delta Est = {estDelta:.4f}, Delta Std Err = {deltaStd:.4f}, Vega Est = {estVega:.4f}, Vega Ste Err = {vegaStd:.4f}")

estDelta, deltaStd = pathwise_delta(S0, K, T, r, q, sigma, n, False)
estVega, vegaStd = pathwise_vega(S0, K, T, r, q, sigma, n, False)
print(f"Pathwise estimate | Delta Est = {estDelta:.4f}, Delta Std Err = {deltaStd:.4f}, Vega Est = {estVega:.4f}, Vega Ste Err = {vegaStd:.4f}")

estDelta, deltaStd = pathwise_delta(S0, K, T, r, q, sigma, n, True)
estVega, vegaStd = pathwise_vega(S0, K, T, r, q, sigma, n, True)
print(f"Pathwise estimate with control | Delta Est = {estDelta:.4f}, Delta Std Err = {deltaStd:.4f}, Vega Est = {estVega:.4f}, Vega Ste Err = {vegaStd:.4f}")

estDelta, deltaStd = likelihood_delta(S0, K, T, r, q, sigma, n, False)
estVega, vegaStd = likelihood_vega(S0, K, T, r, q, sigma, n, False)
print(f"Likelihood estimate | Delta Est = {estDelta:.4f}, Delta Std Err = {deltaStd:.4f}, Vega Est = {estVega:.4f}, Vega Ste Err = {vegaStd:.4f}")

estDelta, deltaStd = likelihood_delta(S0, K, T, r, q, sigma, n, True)
estVega, vegaStd = likelihood_vega(S0, K, T, r, q, sigma, n, True)
print(f"Likelihood estimate with contral | Delta Est = {estDelta:.4f}, Delta Std Err = {deltaStd:.4f}, Vega Est = {estVega:.4f}, Vega Ste Err = {vegaStd:.4f}")
```

```
Resimulation estimate | Delta Est = 0.5672, Delta Std Err = 0.0054, Vega Est = 17.1981, Vega Ste Err = 0.2950
Resimulation estimate with control | Delta Est = 0.5682, Delta Std Err = 0.0030, Vega Est = 17.5048, Vega Ste Err = 0.1569
Pathwise estimate | Delta Est = 0.5767, Delta Std Err = 0.0054, Vega Est = 17.6376, Vega Ste Err = 0.2991
Pathwise estimate with control | Delta Est = 0.5679, Delta Std Err = 0.0029, Vega Est = 17.6220, Vega Ste Err = 0.1542
Likelihood estimate | Delta Est = 0.5700, Delta Std Err = 0.0130, Vega Est = 15.6958, Vega Ste Err = 0.9603
Likelihood estimate with contral | Delta Est = 0.5822, Delta Std Err = 0.0089, Vega Est = 18.0171, Vega Ste Err = 1.1144
```

2) **Applying Broadie and Glasserman to Digital Options.**

(a)

In [ ]:

```python
S0 = 100
K = 105
r = 0.05
sigma = 0.2
T = 1
n = 10000
h = 0.0001
def digital_call_price(S0, K, T, r, sigma):
    d2 = (np.log(S0 / K) + (r - 0.5 * sigma**2) * T) / (sigma * np.sqrt(T))
    return np.exp(-r * T) * stats.norm.cdf(d2)
def digital_call_delta(S0, K, T, r, sigma):
    d2 = (np.log(S0 / K) + (r - 0.5 * sigma**2) * T) / (sigma * np.sqrt(T))
    return np.exp(-r * T) * stats.norm.pdf(d2) / (S0 * sigma * np.sqrt(T))
```

In [ ]:

```python
price = digital_call_price(S0, K, T, r, sigma)
delta = digital_call_delta(S0, K, T, r, sigma)

print(f"Digital call price = {price:.4f}, Delta = {delta:.4f}")
```

```
Digital call price = 0.4400, Delta = 0.0189
```

(b)

In [ ]:

```python
# Payoff function
def digital_payoff(ST, K):
    return (ST > K) * 1

# Delta estimates
def resimulation_digital_delta(S0, h, K, T, r, q, sigma, n, control = False):
    # np.random.seed(42)
    z = np.random.normal(size=n)

    ST_up = (S0 + h) * np.exp((r - q - 0.5 * sigma**2) * T + sigma * np.sqrt(T) * z)
    ST_down = (S0 - h) * np.exp((r - q - 0.5 * sigma**2) * T + sigma * np.sqrt(T) * z)

    mc_delta = np.exp(-r * T) * (digital_payoff(ST_up, K) - digital_payoff(ST_down, K)) / (2 * h)

    if control == False:
        delta = np.mean(mc_delta)
        delta_std = np.std(mc_delta, ddof = 1) / np.sqrt(n)
    else:
        final_price = S0 * np.exp((r - q - 0.5 * sigma**2) * T + sigma * np.sqrt(T) * z)

        cov = np.cov(final_price, mc_delta, ddof = 1)
        a_hat = cov[0, 1] / cov[0, 0]
        control_var = mc_delta - a_hat * (final_price - S0 * np.exp((r - q)* T))

        delta = np.mean(control_var)
        delta_std = np.std(control_var, ddof = 1) / np.sqrt(n)

    return delta, delta_std
```

In [ ]:

```python
price, delta = resimulation_digital_delta(S0, h, K, T, r, 0, sigma, n, control = True)
print(f"Resimulation: Digital call delta = {price:.4f}, Delta stdErr = {delta:.4f}")
```

```
Resimulation: Digital call delta = 0.0000, Delta stdErr = 0.0000
```

(c)

In [ ]:

```python
def likelihood_digital_delta(S0, K, T, r, q, sigma, n, control = False):
    z = np.random.normal(size=n)
    final_price = S0 * np.exp((r - q - 0.5 * sigma**2) * T + sigma * np.sqrt(T) * z)
    ll_delta = np.exp(-r * T) * digital_payoff(final_price, K) * (1 / (S0 * sigma**2 * T)) * (np.log(final_price / S0) - (r - q - 0.5 * sigma**2) * T)

    if control == False:
```

```python
        delta = np.mean(ll_delta)
        delta_std = np.std(ll_delta, ddof = 1) / np.sqrt(n)
    else:
        cov = np.cov(final_price, ll_delta, ddof = 1)
        a_hat = cov[0, 1] / cov[0, 0]
        control_var = ll_delta - a_hat * (final_price - S0 * np.exp((r - q)* T))

        delta = np.mean(control_var)
        delta_std = np.std(control_var, ddof = 1) / np.sqrt(n)
    return  delta, delta_std
```

In [ ]:

```python
price, delta = likelihood_digital_delta(S0, K, T, r, 0, sigma, n, control = True)
print(f"Likelihood: Digital call delta = {price:.4f}, Delta stdErr = {delta:.4f}")
```

```
Likelihood: Digital call delta = 0.0189, Delta stdErr = 0.0001
```

3) **Practice on stratification.**

(a): Standard Monte Carlo Simulation

In [ ]:

```python
# Parameters
S0 = 100
sigma = 0.2
T = 1
r = 0.05
K = 100
n = 1000

# Part (a): Standard Monte Carlo Simulation
uniform_randoms = np.random.rand(n, 2)
normal_randoms = stats.norm.ppf(uniform_randoms)

S1_T = S0 * np.exp((r - 0.5 * sigma**2) * T + sigma * np.sqrt(T) * normal_randoms[:, 0])
S2_T = S0 * np.exp((r - 0.5 * sigma**2) * T + sigma * np.sqrt(T) * normal_randoms[:, 1])
average_S = (S1_T + S2_T) / 2

payoffs = np.maximum(average_S - K, 0)
discounted_payoffs = np.exp(-r * T) * payoffs

estimate_mc = np.mean(discounted_payoffs)
standard_error_mc = np.std(discounted_payoffs) / np.sqrt(n)

print(f"call price = {estimate_mc:.4f}, stdErr = {standard_error_mc:.4f}")
```

```
call price = 7.7739, stdErr = 0.3155
```

(b): Bivariate Stratification

In [ ]:

```python
B = 10
N = int(n/float(B))   #B = number of bins
f1 = (r - .5 * sigma**2) * T
f2 = sigma * np.sqrt(T)
pv = np.exp(-r * T)
# Create a grid of stratified samples
A = np.ones((N, B, B))
A1 = np.cumsum(A, axis=1) - 1
A2 = np.cumsum(A, axis=2) - 1

U1 = np.random.random((N, B, B))
U2 = np.random.random((N, B, B))

A = (A1 + U1) / float(B)
B = (A2 + U2) / float(B)

z1 = stats.norm.ppf(A)
z2 = stats.norm.ppf(B)

S1_T = S0*np.exp(f1+f2*z1)
S2_T = S0*np.exp(f1+f2*z2)

average_S = (S1_T + S2_T) / 2

payoffs = np.maximum(average_S - K, 0)
discounted_payoffs = pv * payoffs

price_vec = np.mean(discounted_payoffs, axis = 0)
var_vec = np.var(discounted_payoffs, axis = 0)

estimate_strata = np.mean(price_vec)
standard_error_strata = np.sqrt(np.mean(var_vec)/n)

print(f"call price = {estimate_strata:.4f} , stdErr = {standard_error_strata:.4f}")
```

```
call price = 8.2579 , stdErr = 0.0914
```

4) **Practice on the Brownian bridge method.**

In [ ]:

```python
S0 = 50
K = 50
T = 0.25
r = 0.1
sigma = 0.25
time_step = 30
n = 1000
dt = T / time_step

z = np.random.normal(size = (n, time_step + 1))

loc = (r - 0.5 * sigma**2) * dt
scale = sigma * np.sqrt(dt)

S = np.zeros((n, time_step + 1))
S[:, 0] = S0
for t in range(1, time_step + 1):
    S[:, t] = S[:, t - 1] + r * S[:, t - 1] * dt + sigma * S[:, t - 1] * np.sqrt(dt) * z[:, t]
```

(a)

In [ ]:

```python
max_call_payoffs = np.maximum(np.max(S[:, 1:], axis = 1) - K, 0)
Cj = np.exp(-r * T) * max_call_payoffs

price = np.mean(Cj)
stdErr = np.std(Cj, ddof=1) / np.sqrt(n)
print(f"The max call price without BDZ = {price:.4f}, stdErr = {stdErr:.4f}")
```

```
The max call price without BDZ = 4.9967, stdErr = 0.1363
```

In [ ]:

```python
b = (S[:, 1:] - S[:, :-1]) / (sigma * S[:, :-1])
u = np.random.uniform(size = (n, time_step))
Mj = (b + np.sqrt(b**2 - 2 * dt * np.log(1 - u))) / 2
St = S[:,:-1] + sigma * S[:,:-1] * Mj
max_call_payoffs = np.maximum(np.max(St, axis = 1) - K, 0)
Cj = np.exp(-r * T) * max_call_payoffs
```

```
price = np.mean(Cj)
stdErr = np.std(Cj, ddof=1) / np.sqrt(n)
print(f"The max call price with BDZ = {price:.4f}, stdErr = {stdErr:.4f}")
```

```
The max call price with BDZ = 5.6595, stdErr = 0.1383
```

(b)

In [ ]:
```
max_call_payoffs = np.maximum(np.max(S[:, 1:], axis = 1) - S[:, -1], 0)
Cj = np.exp(-r * T) * max_call_payoffs

price = np.mean(Cj)
stdErr = np.std(Cj, ddof=1) / np.sqrt(n)
print(f"The max call price with ST as strike without BDZ = {price:.4f}, stdErr = {stdErr:.4f}")
```

```
The max call price with ST as strike without BDZ = 3.8961, stdErr = 0.1009
```

In [ ]:
```
b = (S[:, 1:] - S[:, :-1]) / (sigma * S[:, :-1])
u = np.random.uniform(size = (n, time_step))
Mj = (b + np.sqrt(b**2 - 2 * dt * np.log(1 - u))) / 2
St = S[:,:-1] + sigma * S[:,:-1] * Mj
max_call_payoffs = np.maximum(np.max(St, axis = 1) - S[:, -1], 0)
Cj = np.exp(-r * T) * max_call_payoffs

price = np.mean(Cj)
stdErr = np.std(Cj, ddof=1) / np.sqrt(n)
print(f"The max call price with ST as strike with BDZ = {price:.4f}, stdErr = {stdErr:.4f}")
```

```
The max call price with ST as strike with BDZ = 4.6379, stdErr = 0.1024
```

5) **Two-Asset Down-and-Out Call Option Pricing (Discrete and Continuous)**

(a)

In [ ]:
```
S1_0 = S2_0 = K = 100
r = 0.1
sigma = 0.3
rho = 0.5
T = 0.2
H = 95
time_step = 50
dt = T / time_step
n = 10000

def gbm_path(S_0, ttm, rf, sigma, N, z):
    delta_t = ttm / N
    zs = np.cumsum((rf - 0.5 * sigma**2) * delta_t + sigma * np.sqrt(delta_t) * z)
    S = S_0 * np.exp(zs)
    return np.append(S_0 ,S)

def call_payoff(S1, S2, K, H):
    return np.maximum(S1[-1] - K, 0) * ~np.any(S2 < H)

corr = np.array([[1, rho], [rho, 1]])
chol = np.linalg.cholesky(corr)
```

In [ ]:
```
mc_payoff = np.zeros(n)
for j in range(n):
    z = np.matmul(chol, np.random.normal(size=(2, time_step)))
    S1 = gbm_path(S1_0, T, r, sigma, time_step, z[0, :])
    S2 = gbm_path(S2_0, T, r, sigma, time_step, z[1, :])
    mc_payoff[j] = call_payoff(S1, S2, K, H)

Cj = np.exp(-r * T) * mc_payoff
price = np.mean(Cj)
stdErr = np.std(Cj, ddof = 1) / np.sqrt(n)

print(f"call price = {price:.4f}, stdErr = {stdErr:.4f}")
```

```
call price = 3.6279, stdErr = 0.0817
```

In [ ]:
```
mc_payoff = np.zeros(n)
for j in range(n):
    z = np.matmul(chol, np.random.normal(size=(2, time_step)))
    S1 = gbm_path(S1_0, T, r, sigma, time_step, z[0, :])
    S2 = gbm_path(S2_0, T, r, sigma, time_step, z[1, :])

    b = (S2[1:] - S2[:-1]) / (sigma * S2[:-1])
    u = np.random.uniform(size = time_step)
    Mj = (b - np.sqrt(b**2 - 2 * dt * np.log(1 - u))) / 2
    S2_t = S2[:-1] + sigma * S2[:-1] * Mj
    mc_payoff[j] = call_payoff(S1, S2_t, K, H)

Cj = np.exp(-r * T) * mc_payoff
price = np.mean(Cj)
stdErr = np.std(Cj, ddof = 1) / np.sqrt(n)

print(f"call price with BDZ= {price:.4f}, stdErr = {stdErr:.4f}")
```

```
call price with BDZ= 3.2634, stdErr = 0.0768
```