Simulation Homework V

Ze Yang (zey@andrew.cmu.edu)

February 21, 2018

```
In [1]: from abc import ABCMeta, abstractmethod
        import time
        import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        import seaborn as sns
        plt.style.use('ggplot')
        plt.rc('text', usetex=True)
        plt.rc('font', family='serif', size=15)
        %matplotlib inline

        import scipy.stats as stats
        from scipy.stats import norm
        from progressbar import ProgressBar
```

1 Stratification

```
In [113]: def standard_mc_pair(S0, K, sigma, r, T, n):
              U = np.random.uniform(size=(n,2))
              Z = norm.ppf(U)
              S1 = S0*np.exp((
                  r-0.5*sigma**2)*T + sigma*np.sqrt(T)*Z[:,0])
              S2 = S0*np.exp((
                  r-0.5*sigma**2)*T + sigma*np.sqrt(T)*Z[:,1])
              sample = np.exp(-r*T)*np.clip((S1+S2)/2-K, 0, None)
              sample_mean = np.mean(sample)
              se = stats.sem(sample, ddof=0)
              return sample, sample_mean, se

          def stratefied_mc_pair(S0, K, sigma, r, T, B, bin_size):
              bins = np.linspace(0, 1, B+1)
              price_grid = np.zeros((B,B))
              var_grid = np.zeros((B,B))
              n = B*B*bin_size
              for i in range(B):
                  for j in range(B):
                      U1 = np.random.uniform(
```

```python
                low=bins[i], high=bins[i+1],
                size=bin_size)
            U2 = np.random.uniform(
                low=bins[j], high=bins[j+1],
                size=bin_size)
            Z1 = norm.ppf(U1)
            Z2 = norm.ppf(U2)
            S1 = S0*np.exp((
                r-0.5*sigma**2)*T + sigma*np.sqrt(T)*Z1)
            S2 = S0*np.exp((
                r-0.5*sigma**2)*T + sigma*np.sqrt(T)*Z2)
            sample = np.exp(-r*T)*np.clip((S1+S2)/2-K, 0, None)
            price_grid[i][j] = np.mean(sample)
            var_grid[i][j] = np.var(sample)
    sample_mean = np.mean(price_grid)
    se = np.sqrt(np.sum(var_grid)/(B**2))/np.sqrt(n)
    return sample_mean, se

def stratified_projection(S0, K, sigma, r, T, B, bin_size):
    n = B*bin_size
    bins = np.linspace(0, 1, B+1)
    nu = np.array([1.0,1.0]) / np.sqrt(2)
    Sigma_Z = np.identity(2) - nu.reshape(
        2,1).dot(nu.reshape(1,2))
    price_grid = np.zeros(B)
    var_grid = np.zeros(B)
    for i in range(B):
        U = np.random.uniform(
            low=bins[i], high=bins[i+1],
            size=bin_size)
        X = norm.ppf(U)
        sample_bin = np.zeros(bin_size)
        for j in range(bin_size):
            Z = np.random.multivariate_normal(nu*X[j], Sigma_Z)
            S1 = S0*np.exp((
                r-0.5*sigma**2)*T + sigma*np.sqrt(T)*Z[0])
            S2 = S0*np.exp((
                r-0.5*sigma**2)*T + sigma*np.sqrt(T)*Z[1])
            sample_bin[j] = np.exp(-r*T)*np.clip(
                (S1+S2)/2-K, 0, None)
        price_grid[i] = np.mean(sample_bin)
        var_grid[i] = np.var(sample_bin)
    sample_mean = np.mean(price_grid)
    se = np.sqrt(np.sum(var_grid)/B)/np.sqrt(n)
    return sample_mean, se
```

1.1 Standard Monte-Carlo

```
In [115]: table = []
          _, price, se = standard_mc_pair(
              S0=100, K=100, sigma=0.2, r=0.05, T=1, n=1000)
          print('Standard MC: Price = {}, SE = {}'.format(price, se))
          table.append(['Standard MC', price, se])
```

```
Standard MC: Price = 8.717088273068166, SE = 0.34969214409253035
```

1.2 2-Dimensional Stratification

```
In [116]: price, se = stratefied_mc_pair(
              S0=100, K=100, sigma=0.2, r=0.05, T=1, B=10, bin_size=10)
          print('Stratified MC: Price = {}, SE = {}'.format(price, se))
          table.append(['2D Stratified MC', price, se])
```

```
Stratified MC: Price = 8.227946729386634, SE = 0.0781876992906378
```

1.3 Stratification of Projection

```
In [117]: price, se = stratified_projection(
              S0=100, K=100, sigma=0.2, r=0.05, T=1, B=250, bin_size=4)
          print('Stratification of Projection: Price = {}, SE = {}'.format(price, se))
          table.append(['Stratified Projection', price, se])
```

```
Stratification of Projection: Price = 8.361912744826364, SE = 0.03789547033221843
```

```
In [118]: summary = pd.DataFrame(table, columns=[
              'Method', 'Price', 'SE'])
          summary
```

Out[118]:

	Method	Price	SE
0	Standard MC	8.717088	0.349692
1	2D Stratified MC	8.227947	0.078188
2	Stratified Projection	8.361913	0.037895

2 Brownian Bridge

```
In [118]: def gbm_bdz(S0, sigma, r, T, n_steps):
              dt = T/n_steps
              S_last, S, i = S0, S0, 0
              Z = np.random.normal(size=n_steps)
              while i < n_steps:
                  # euler scheme
                  S_last = S
```

```
            S += r*S*dt + sigma*S*np.sqrt(dt)*Z[i]
            b = (S-S_last) / (sigma*S_last)
            u = np.random.uniform()
            B_max = 0.5*(b+np.sqrt(b**2 - 2*dt*np.log(u)))
            B_min = 0.5*(b-np.sqrt(b**2 - 2*dt*np.log(u)))
            M = S_last + sigma*S_last*B_max
            m = S_last + sigma*S_last*B_min
            yield (S, M, m)
            i += 1
```

2.1 Max Call Option

```
In [119]: def max_call(S0, K, T, r, sigma,
                       n_steps, n_size):
              sample = np.zeros(n_size)
              sample_bdz = np.zeros(n_size)
              bar = ProgressBar()
              for j in bar(range(n_size)):
                  S, M, _ = zip(*gbm_bdz(
                      S0, sigma, r, T, n_steps))
                  sample_bdz[j] = np.exp(-r*T)*np.clip(
                      np.max(M)-K, 0, None)
                  sample[j] = np.exp(-r*T)*np.clip(
                      np.max(S)-K, 0, None)
              sample_mean = np.mean(sample)
              se = stats.sem(sample, ddof=0)
              sample_mean_bdz = np.mean(sample_bdz)
              se_bdz = stats.sem(sample_bdz, ddof=0)
              return sample_mean, se, sample_mean_bdz, se_bdz
```

```
In [120]: price, se, price_bdz, se_bdz = max_call(
              S0=50, K=50, T=0.25, r=0.1, sigma=0.25,
              n_steps=30, n_size=1000)
          print('Standard MC: Max Call Price = {}, SE = {}'.format(price, se))
          print('Brownian Bridge Correction: Max Call Price = {}, SE = {}'.format(
              price_bdz, se_bdz))
```

```
100% (1000 of 1000) |#####################| Elapsed Time: 0:00:00 Time: 0:00:00


Standard MC: Max Call Price = 4.8759584965664935, SE = 0.14101773642732715
Brownian Bridge Correction: Max Call Price = 5.548656198264789, SE = 0.14254263465548778
```

2.2 Max Call Option, Strike = S_T

```
In [121]: def max_call_ST(S0, K, T, r, sigma,
                          n_steps, n_size):
              sample = np.zeros(n_size)
```

```
            sample_bdz = np.zeros(n_size)
            bar = ProgressBar()
            for j in bar(range(n_size)):
                S, M, _ = zip(*gbm_bdz(
                    S0, sigma, r, T, n_steps))
                sample_bdz[j] = np.exp(-r*T)*np.clip(
                    np.max(M)-S[-1], 0, None)
                sample[j] = np.exp(-r*T)*np.clip(
                    np.max(S)-S[-1], 0, None)
            sample_mean = np.mean(sample)
            se = stats.sem(sample, ddof=0)
            sample_mean_bdz = np.mean(sample_bdz)
            se_bdz = stats.sem(sample_bdz, ddof=0)
            return sample_mean, se, sample_mean_bdz, se_bdz
```

```
In [124]: price, se, price_bdz, se_bdz = max_call_ST(
              S0=50, K=50, T=0.25, r=0.1, sigma=0.25,
              n_steps=30, n_size=1000)
          print('Standard MC: Max Call Price = {}, SE = {}'.format(price, se))
          print('Brownian Bridge Correction: Max Call Price = {}, SE = {}'.format(
              price_bdz, se_bdz))
```

```
100% (1000 of 1000) |#####################| Elapsed Time: 0:00:00 Time: 0:00:00


Standard MC: Max Call Price = 3.8798771180345355, SE = 0.10474966632999966
Brownian Bridge Correction: Max Call Price = 4.618006133108648, SE = 0.10623307153227779
```

2.3 Knock-out Option

```
In [126]: def knock_out(S0, K, H, T, r, sigma,
                        n_steps, n_size):
              sample = np.zeros(n_size)
              sample_bdz = np.zeros(n_size)
              bar = ProgressBar()
              for j in bar(range(n_size)):
                  S, _, m = zip(*gbm_bdz(
                      S0, sigma, r, T, n_steps))
                  sample_bdz[j] = np.exp(-r*T)*np.clip(
                      S[-1]-K, 0, None)*(np.min(m)>H)
                  sample[j] = np.exp(-r*T)*np.clip(
                      S[-1]-K, 0, None)*(np.min(S)>H)
              sample_mean = np.mean(sample)
              se = stats.sem(sample, ddof=0)
              sample_mean_bdz = np.mean(sample_bdz)
              se_bdz = stats.sem(sample_bdz, ddof=0)
              return sample_mean, se, sample_mean_bdz, se_bdz
```

```
In [128]: price, se, price_bdz, se_bdz = knock_out(
              S0=50, K=50, H=45, T=0.25, r=0.1, sigma=0.5,
              n_steps=30, n_size=100000)
          print('Standard MC: Knock-out Call Price = {}, SE = {}'.format(price, se))
          print('Brownian Bridge Correction: Knock-out Call Price = {}, SE = {}'.format(
              price_bdz, se_bdz))
```

```
Standard MC: Knock-out Call Price = 4.532198747276875, SE = 0.027182389608430696
Brownian Bridge Correction: Knock-out Call Price = 4.021150035286275, SE = 0.0265014658467239
```

3 Two-Asset Down-and-Out Call

```
In [139]: def gbm_2d_bdz(S10, S20, sigma1, sigma2, rho, r, T, n_steps):
              dt = T/n_steps
              S1, S2, m2, i = S10, S20, S20, 0
              corr = np.array(
                  [[1, rho],
                   [rho, 1]])
              while i < n_steps:
                  Z = np.random.multivariate_normal(np.zeros(2), corr)
                  dS1 = r*S1*dt + sigma1*S1*np.sqrt(dt)*Z[0]
                  dS2 = r*S2*dt + sigma2*S2*np.sqrt(dt)*Z[1]
                  b = dS2 / (sigma2*S2)
                  u = np.random.uniform()
                  B_min = 0.5*(b-np.sqrt(b**2 - 2*dt*np.log(u)))
                  m2 = S2 + sigma2*S2*B_min
                  S1 += dS1; S2 += dS2
                  yield (S1, S2, m2)
                  i += 1

          def knock_out_2d(S10, S20, K, H, sigma1, sigma2, rho,
                           T, r, n_steps, n_size):
              sample = np.zeros(n_size)
              sample_bdz = np.zeros(n_size)
              bar = ProgressBar()
              for j in bar(range(n_size)):
                  S1, S2, m2 = zip(*gbm_2d_bdz(
                      S10, S20, sigma1, sigma2, rho, r, T, n_steps))
                  sample_bdz[j] = np.exp(-r*T)*np.clip(
                      S1[-1]-K, 0, None)*(np.min(m2)>H)
                  sample[j] = np.exp(-r*T)*np.clip(
                      S1[-1]-K, 0, None)*(np.min(S2)>H)
              sample_mean = np.mean(sample)
              se = stats.sem(sample, ddof=0)
              sample_mean_bdz = np.mean(sample_bdz)
              se_bdz = stats.sem(sample_bdz, ddof=0)
              return sample_mean, se, sample_mean_bdz, se_bdz
```

```
In [143]: price, se, price_bdz, se_bdz = knock_out_2d(
              S10=100, S20=100, K=100, H=95,
              sigma1=0.3, sigma2=0.3, rho=0.5,
              T=0.2, r=0.1, n_steps=50, n_size=10000)
          print('Standard MC: Knock-out Call Price = {}, SE = {}'.format(price, se))
          print('Brownian Bridge Correction: Knock-out Call Price = {}, SE = {}'.format(
              price_bdz, se_bdz))
```

```
100% (10000 of 10000) |###################| Elapsed Time: 0:01:43 Time: 0:01:43


Standard MC: Knock-out Call Price = 3.65625860103613, SE = 0.08043259974580645
Brownian Bridge Correction: Knock-out Call Price = 3.191486056258811, SE = 0.0766601904086251
```

We find that the BDZ solution matches the continuous time analytical solution very closely.

4 Credit Derivatives

```
In [180]: # copy the code from HW2
          class GaussianCopula():

              def __init__(self, cov_matrix, marginal_inv_cdfs=None):
                  self.inv_cdfs = marginal_inv_cdfs
                  self.cov = cov_matrix
                  self.std = np.sqrt(np.diag(self.cov))
                  self.d = len(self.cov)
                  self.A = np.linalg.cholesky(self.cov).T

              def draw(self, n):
                  Z = np.random.normal(size=(n, self.d))
                  Y = Z.dot(self.A)
                  if not self.inv_cdfs: return Y
                  U = norm.cdf(Y/self.std)
                  return np.apply_along_axis(self.inv_cdfs, 1, U)

              def reset_cov(self, cov_matrix):
                  assert len(cov_matrix) == self.d
                  self.cov = cov_matrix
                  self.std = np.sqrt(np.diag(self.cov))
                  self.A = np.linalg.cholesky(self.cov).T

          # inverse marginal cdfs
          def exponential_marginal_inv(lam_vec):
              return lambda u: np.array(
                  [(-1/lam)*np.log(u[i]) for i,lam in enumerate(lam_vec)])
```

```
In [192]: table = []
          N = 5
```

```
T = 5
r = 0.04
s = 0.01
R = 0.35
lam = s/(1-R) # using Taylor approx
n_sample = 100000

gsexp_copula_sampler = GaussianCopula(
    cov_matrix=np.identity(N),
    marginal_inv_cdfs=exponential_marginal_inv([lam]*N)
)

bar = ProgressBar()
for i, rho in bar(list(enumerate([0, 0.2, 0.4, 0.6, 0.8, 0.9999999]))):
    gsexp_copula_sampler.reset_cov(
        rho*np.ones((N,N))+(1-rho)*np.identity(N)
    )
    time_to_default = gsexp_copula_sampler.draw(n_sample)
    num_defaults = np.apply_along_axis(
        lambda row: sum(row<T), 1, time_to_default)
    prob, value = [], []
    for k in range(N):
        # probability of itD derivative is in the money:
        p_itm = sum(num_defaults >= (k+1)) / n_sample
        # value of the itD derivative:
        v = np.exp(-r*T)*(1-R)*p_itm
        prob.append(p_itm)
        value.append(v)
    table.append(prob)
    table.append(value)
```

```
100% (6 of 6) |###########################| Elapsed Time: 0:00:29 Time: 0:00:29
```

In [193]:
```
index = [['0', '0.2', '0.4', '0.6', '0.8', '1.0'],
         ['Probability of Paying', 'Value']]
index = pd.MultiIndex.from_product(
    index, names=['Rho', ' '])
summary = pd.DataFrame(table, columns=[
    'FtD', '2tD', '3tD', '4tD', '5tD'], index=index)
summary
```

Out[193]:

Rho		FtD	2tD	3tD	4tD	5tD
0	Probability of Paying	0.319560	0.048520	0.003690	0.000160	0.000000
	Value	0.170062	0.025821	0.001964	0.000085	0.000000
0.2	Probability of Paying	0.284870	0.068870	0.013820	0.002140	0.000220
	Value	0.151601	0.036651	0.007355	0.001139	0.000117

```
0.4 Probability of Paying  0.249430  0.082550  0.027740  0.007900  0.001620
    Value                  0.132740  0.043931  0.014763  0.004204  0.000862
0.6 Probability of Paying  0.209090  0.092640  0.043920  0.018800  0.006410
    Value                  0.111272  0.049301  0.023373  0.010005  0.003411
0.8 Probability of Paying  0.164690  0.095020  0.059370  0.036240  0.018360
    Value                  0.087644  0.050567  0.031595  0.019286  0.009771
1.0 Probability of Paying  0.075140  0.075100  0.075090  0.075070  0.075020
    Value                  0.039988  0.039966  0.039961  0.039950  0.039924
```

```
In [ ]:
```