Simulation HW VI

Ze Yang (zey@andrew.cmu.edu)

March 1, 2018

```
In [1]: import time
        import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        import operator as op
        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
        from sklearn.linear_model import LinearRegression
```

1 Longstaff-Schwartz Method

```
In [12]: def bs(t, S, K, T, sigma, r, div=0):
             tau = T - t
             rexp, dexp = np.exp(-r*tau), np.exp(-div*tau)
             d1 = (np.log(S/K) + (r-div+0.5*sigma**2)*tau) / (
                 sigma*np.sqrt(tau))
             d2 = d1 - sigma*np.sqrt(tau)
             call = S*dexp*norm.cdf(d1) - K*rexp*norm.cdf(d2)
             put = K*rexp - S*dexp + call
             return call, put

         # Generators
         def gbm_antithetic(S0, sigma, r, div, T,
                            n_steps, size, antithetic=True):
             dt = T/n_steps
             print('Generating GBM paths...')
             S = np.random.normal(size=(n_steps, size))
             S_a = -S
             bar = ProgressBar()
             time.sleep(0.5)
             for j in bar(range(size)):
```

```
            z = S[:,j]; z_a = S_a[:,j]
            logr = np.cumsum(sigma*np.sqrt(dt)*z + (
                r-div-0.5*sigma**2)*dt)
            logr_a = np.cumsum(sigma*np.sqrt(dt)*z_a + (
                r-div-0.5*sigma**2)*dt)
            S[:,j] = S0*np.exp(logr)
            S_a[:,j] = S0*np.exp(logr_a)
        if not antithetic: return S
        return S, S_a
```

In [3]:
```
def laguerre_poly(St, X):
    X[:,1] = np.exp(-St/2)
    X[:,2] = X[:,1]*(1-St)
    X[:,3] = X[:,1]*(1-2*St+(St**2)/2)
    return X

def longstaff_schwartz(S0, K, sigma, r, div, T,
                       n_steps, size):
    dt = T/n_steps
    S, Sa = gbm_antithetic(
        S0, sigma, r, div, T, n_steps, size)
    S = np.concatenate((S,Sa), 1); size*=2
    cash_flows = np.zeros((n_steps, size))
    # tau are indices!
    tau = (n_steps-1) * np.ones(size, dtype=int)
    c = np.zeros(size)
    path_in = np.ones(size, dtype=bool)
    X = np.zeros((size, 4))
    y = np.zeros((size, 1))
    cash_flows[-1,:] = np.clip(K-S[-1,:], 0, None)
    bar = ProgressBar()
    lm = LinearRegression(fit_intercept=False)
    print('Running backward induction...')
    time.sleep(0.5)
    for i in bar(range(n_steps-1)[::-1]):
        path_in = (K-S[i,:] > 0)
        y = cash_flows[tau, range(size)] * path_in
        X = np.apply_along_axis(
            op.__mul__, 0,
            laguerre_poly(S[i,:]/K, X), path_in)
        lm.fit(X,y)
        y_hat = lm.predict(X)
        # y_hat = X.dot(np.linalg.inv(X.T.dot(X))).dot(X.T).dot(y)
        # Calculating hat matrix is too slow
        exec_ = (y_hat < np.clip(K-S[i,:], 0, None))
        cash_flows[i,exec_] = np.clip(
            K-S[i,:], 0, None)[exec_]
        tau[exec_] = i
```

```
        c = cash_flows[tau, range(size)]*np.exp(-r*(tau+1)*dt)
        sample = np.fmax(c, np.clip(K-S0, 0, None))
        # antithetic adjustments
        half = int(size/2)
        sample = (sample[:half] + sample[half:]) / 2
        sample_mean = np.mean(sample)
        se = stats.sem(sample, ddof=0)
        return sample, sample_mean, se
```

```
In [9]: table = []
        _, price, se = longstaff_schwartz(
            S0=40, K=40, sigma=0.2, r=0.06, div=0,
            T=1, n_steps=50, size=50000)
        eu_price = bs(0, 40, 40, 1, 0.2, 0.06)[1]
        table.append([40, 0.2, 1, price, se,
                      eu_price, price-eu_price])
        print('''
        Longstaff Schwartz American put option (T=1):
        Price = {}, SE = {}\n'''.format(price, se))
```

```
Generating GBM paths...


100% (50000 of 50000) |###################| Elapsed Time: 0:00:01 Time: 0:00:01


Running backward induction...


100% (49 of 49) |#########################| Elapsed Time: 0:00:01 Time: 0:00:01



Longstaff Schwartz American put option (T=1):
Price = 2.310152934704085, SE = 0.0056357320791467335
```

```
In [10]: _, price, se = longstaff_schwartz(
             S0=40, K=40, sigma=0.2, r=0.06, div=0,
             T=2, n_steps=100, size=50000)
         eu_price = bs(0, 40, 40, 2, 0.2, 0.06)[1]
         table.append([40, 0.2, 2, price, se,
                       eu_price, price-eu_price])
         print('''
         Longstaff Schwartz American put option (T=2):
         Price = {}, SE = {}\n'''.format(price, se))
```

```
Generating GBM paths...
```

```
100% (50000 of 50000) |###################| Elapsed Time: 0:00:01 Time: 0:00:01

Running backward induction...

100% (99 of 99) |########################| Elapsed Time: 0:00:03 Time: 0:00:03

Longstaff Schwartz American put option (T=2):
Price = 2.863737791388795, SE = 0.007071192795570061
```

```
In [11]: summary = pd.DataFrame(table, columns=[
             'S', 'sigma', 'T', 'Simulated American', 'SE',
             'Closed Form European', 'Early Exercise Value'])
         summary
```

```
Out[11]:     S  sigma  T  Simulated American        SE  Closed Form European  \
         0  40    0.2  1            2.310153  0.005636              2.066401
         1  40    0.2  2            2.863738  0.007071              2.355866

            Early Exercise Value
         0              0.243752
         1              0.507872
```

2 GHS and Capriotti Importance Sampling

2.1 GHS Method

```
In [95]: from scipy.optimize import newton, least_squares

         def standard_mc(S0, K, sigma, r, div,
                         T, n_steps, size):
             dt = T/n_steps
             Z = np.random.normal(size=size)
             S = S0*np.exp(sigma*np.sqrt(T)*Z+(
                 r-0.5*sigma**2)*T)
             sample = np.exp(-r*T)*np.clip(S-K, 0, None)
             sample_mean = np.mean(sample)
             se = stats.sem(sample, ddof=0)
             return sample, sample_mean, se

         def ghs_mc(S0, K, sigma, r, div,
                    T, n_steps, size):
             dt = T/n_steps
             Z = np.random.normal(size=size)
```

```
    def f(m):
        y = S0*np.exp(sigma*np.sqrt(T)*m+(
            r-0.5*sigma**2)*T)
        return (sigma*np.sqrt(T)*y)/(y-K)-m
    m0 = (np.log(K/S0)-(r-0.5*sigma**2)*T)/(
        sigma*np.sqrt(T)) + K/10e6
    m = newton(f, m0)
    S = S0*np.exp(sigma*np.sqrt(T)*(Z+m)+(
        r-0.5*sigma**2)*T)
    sample = np.exp(-r*T)*np.clip(
        S-K, 0, None)*np.exp(-m*Z-0.5*m**2)
    sample_mean = np.mean(sample)
    se = stats.sem(sample, ddof=0)
    return sample, sample_mean, se, m

def capriotti_mc(S0, K, sigma, r, div,
                 T, n_steps, size):
    dt = T/n_steps
    Z = np.random.normal(size=size)
    def resid(theta):
        S = S0*np.exp(sigma*np.sqrt(T)*Z+(
            r-0.5*sigma**2)*T)
        G = np.exp(-r*T)*np.clip(S-K, 0, None)
        W_theta = np.exp(-theta*Z+0.5*theta**2)
        return G*np.sqrt(W_theta)
    opt_out = least_squares(resid, 1, method='lm')
    m = opt_out['x'][0]
    S = S0*np.exp(sigma*np.sqrt(T)*(Z+m)+(
        r-0.5*sigma**2)*T)
    sample = np.exp(-r*T)*np.clip(
        S-K, 0, None)*np.exp(-m*Z-0.5*m**2)
    sample_mean = np.mean(sample)
    se = stats.sem(sample, ddof=0)
    return sample, sample_mean, se, m

def capriotti_mc_2dopt(S0, K, sigma, r, div,
                       T, n_steps, size):
    dt = T/n_steps
    Z = np.random.normal(size=size)
    def resid(theta):
        m, s = theta[0], theta[1]
        S = S0*np.exp(sigma*np.sqrt(T)*Z+(
            r-0.5*sigma**2)*T)
        G = np.exp(-r*T)*np.clip(S-K, 0, None)
        W_theta = s*np.exp(-0.5*(Z**2-((Z-m)/s)**2))
        return G*np.sqrt(W_theta)
    opt_out = least_squares(
        resid, np.array([1,0.6]), method='lm')
```

```
        m, s = opt_out['x'][0], opt_out['x'][1]
        X = np.random.normal(m, s, size=size)
        S = S0*np.exp(sigma*np.sqrt(T)*X+(
            r-0.5*sigma**2)*T)
        W = s*np.exp(-0.5*(X**2-((X-m)/s)**2))
        sample = np.exp(-r*T)*np.clip(
            S-K, 0, None)*W
        sample_mean = np.mean(sample)
        se = stats.sem(sample, ddof=0)
        return sample, sample_mean, se, m, s
```

```
In [73]: table = []
         for K in [120, 140, 160]:
             _, price, se = standard_mc(
                 S0=100, K=K, sigma=0.2,
                 r=0.05, div=0, T=1,
                 n_steps=50, size=10000)
             print('''
             Standard Monte Carlo, K={}:
             Price = {}, SE = {}\n'''.format(K, price, se))
             _, price_ghs, se_ghs, m = ghs_mc(
                 S0=100, K=K, sigma=0.2,
                 r=0.05, div=0, T=1,
                 n_steps=50, size=10000)
             print('''
             GHS Importantance Sampling, K={}:
             Price = {}, SE = {}\n'''.format(K, price_ghs, se_ghs))
             bsp = bs(0, 100, K, 1, 0.2, 0.05)[0]
             table.append([K, bsp, price, se, m, price_ghs, se_ghs])
```

```
Standard Monte Carlo, K=120:
Price = 3.1873007231456136, SE = 0.0845664180964543


GHS Importantance Sampling, K=120:
Price = 3.218273618850227, SE = 0.02238118486963999


Standard Monte Carlo, K=140:
Price = 0.7796532195664344, SE = 0.042337653131319035


GHS Importantance Sampling, K=140:
Price = 0.7803352637969179, SE = 0.006771858007465234
```

```
Standard Monte Carlo, K=160:
Price = 0.16663257579415966, SE = 0.018700978756931275


GHS Importantance Sampling, K=160:
Price = 0.15966767729065617, SE = 0.001614465090058512
```

```
In [74]: summary = pd.DataFrame(table, columns=[
             'Strike', 'BS Price', 'Std MC Price', 'SE', 'm hat',
             'GHS Price', 'SE'])
         summary
```

Out[74]:

	Strike	BS Price	Std MC Price	SE	m hat	GHS Price	SE
0	120	3.247477	3.187301	0.084566	1.484927	3.218274	0.022381
1	140	0.784965	0.779653	0.042338	2.046545	0.780335	0.006772
2	160	0.158954	0.166633	0.018701	2.600200	0.159668	0.001614

2.2 Capriotti Method

```
In [79]: table = []
         for K in [120, 140, 160]:
             _, price_cap, se_cap, m = capriotti_mc(
                 S0=100, K=K, sigma=0.2,
                 r=0.05, div=0, T=1,
                 n_steps=50, size=10000)
             print('''
             Capriotti Importantance Sampling, K={}:
             Price = {}, SE = {}\n'''.format(K, price_cap, se_cap))
             bsp = bs(0, 100, K, 1, 0.2, 0.05)[0]
             table.append([K, m, price_cap, se_cap])
```

```
Capriotti Importantance Sampling, K=120:
Price = 3.2221674200398107, SE = 0.021767249389525834


Capriotti Importantance Sampling, K=140:
Price = 0.7693660057898325, SE = 0.00669979087195573


Capriotti Importantance Sampling, K=160:
Price = 0.16048166685096502, SE = 0.001604889018391324
```

In [80]:
```
summary = pd.DataFrame(table, columns=[
        'Strike', 'm hat', 'Capriotti Price', 'SE'])
summary
```

Out[80]:

	Strike	m hat	Capriotti Price	SE
0	120	1.673981	3.222167	0.021767
1	140	2.211192	0.769366	0.006700
2	160	2.800994	0.160482	0.001605

2.3 Capriotti Method with Both m and s^2

In [102]:
```
table = []
for K in [120, 140, 160]:
    _, price_cap, se_cap, m, s = capriotti_mc_2dopt(
        S0=100, K=K, sigma=0.2,
        r=0.05, div=0, T=1,
        n_steps=50, size=10000)
    print('''
    Capriotti Importantance Sampling, K={}:
    Price = {}, SE = {}\n'''.format(K, price_cap, se_cap))
    table.append([K, m, s, price_cap, se_cap])
```

```
Capriotti Importantance Sampling, K=120:
Price = 3.2422467794496246, SE = 0.011497775330273638


Capriotti Importantance Sampling, K=140:
Price = 0.7803004463362502, SE = 0.002993015991838408


Capriotti Importantance Sampling, K=160:
Price = 0.1587229811302735, SE = 0.0006873805576155507
```

In [103]:
```
summary = pd.DataFrame(table, columns=[
        'Strike', 'm hat', 's hat', 'Capriotti Price', 'SE'])
summary
```

Out[103]:

	Strike	m hat	s hat	Capriotti Price	SE
0	120	1.754983	0.627054	3.242247	0.011498
1	140	2.286233	0.499687	0.780300	0.002993
2	160	2.830673	0.462060	0.158723	0.000687

We can see that the standard error is further reduced when we include s^2 in the non-linear least square optimization.