Simulation HW3

Ze Yang (zey@andrew.cmu.edu)

February 8, 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
```

In [2]:
```
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

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

def bs_down_in(t, S, K, H, T, sigma, r, div=0):
    tau = T - t
    rexp, dexp = np.exp(-r*tau), np.exp(-div*tau)
    lam = (r-div+0.5*sigma**2)/(sigma**2)
```

```
        y1 = np.log(H**2/(S*K)) / (
            sigma*np.sqrt(tau)) + lam*sigma*np.sqrt(tau)
        y2 = y1-sigma*np.sqrt(tau)
        call = S*dexp*((H/S)**(2*lam))*norm.cdf(y1) - (
            K*rexp*((H/S)**(2*lam-2))*norm.cdf(y2))
        return call
```

In [37]:
```
class MonteCarloEstimator(object):

    @staticmethod
    def vanilla_call(K, tau, r, L=None):
        def __payoff(paths, paths_a=None):
            call = np.fmax(
                0, np.exp(-r*tau)*(paths[-1,:]-K))
            if paths_a is not None:
                call += np.fmax(
                    0, np.exp(-r*tau)*(paths_a[-1,:]-K))
                call /= 2
            if L is not None:
                Phi_L = norm.cdf(L)
                call *= (1-Phi_L)
            return call
        return __payoff

    @staticmethod
    def vanilla_call_parity(K, S0, tau, r):
        def __payoff(paths, paths_a=None):
            put = np.fmax(
                0, np.exp(-r*tau)*(K-paths[-1,:]))
            return put + S0 - K*np.exp(-r*tau)
        return __payoff

    @staticmethod
    def arithmetic_asian_call(K, tau, r):
        def __payoff(paths, paths_a=None):
            call = np.fmax(
                0, np.exp(-r*tau)*(np.mean(paths,axis=0)-K))
            if paths_a is not None:
                call += np.fmax(0, np.exp(-r*tau)*(
                    np.mean(paths_a,axis=0)-K))
                call /= 2
            return call
        return __payoff

    @staticmethod
    def down_in_digital(K, H, tau, r,
                        stop=False,
                        sigma=None, div=0):
```

```
    def __payoff(paths):
        pay = 10000.0*(paths[-1,:] > K)*(
            np.min(paths,axis=0) < H)
        pay *= np.exp(-r*tau)
        return pay

    def __conditional_mc_payoff(paths,
                                stopping_times,
                                rn_derivatives=None):
        n_steps, size = paths.shape
        dt = tau/n_steps
        pay = np.zeros(size)
        for j, stop in enumerate(stopping_times):
            if stop < 0:
                continue
            else:
                t = stop*dt
                pay[j] = np.exp(-r*t)*bs_digital(
                    0, paths[stop-1, j],
                    K, tau-t, sigma, r ,div=0)[0]*10000.0
        return pay
    if stop: return __conditional_mc_payoff
    return __payoff

@staticmethod
def down_in_call(K, H, tau, r,
                 stop=False,
                 sigma=None,
                 div=0, theta=0):
    def __payoff(paths):
        pay = np.fmax(
            0, (paths[-1,:]-K))*(
            np.min(paths,axis=0) < H)
        pay *= np.exp(-r*tau)
        return pay

    def __conditional_mc_payoff(paths,
                                stopping_times,
                                rn_derivatives=None):
        n_steps, size = paths.shape
        dt = tau/n_steps
        pay = np.zeros(size)
        for j, stop in enumerate(stopping_times):
            if stop < 0:
                continue
            else:
                t = stop*dt
                pay[j] = np.exp(-r*t)*bs(
```

```
                        0, paths[stop-1, j],
                        K, tau-t, sigma, r ,div=0)[0]
                    if rn_derivatives is not None:
                        pay[j] *= rn_derivatives[j]
            return pay
        if stop: return __conditional_mc_payoff
        return __payoff

    @staticmethod
    def gbm(S0, T, r, sigma, div=0,
            antithetic=False, truncate=None,
            stop=None, theta=0):
        def __gbm(n_steps, size):
            dt = T/n_steps
            print("[gbm]: Initializing grids...")
            time.sleep(0.5)
            S = np.random.normal(size=(n_steps, size))
            bar = ProgressBar()
            for j in bar(range(size)):
                z = S[:,j]
                logr = np.cumsum(sigma*np.sqrt(dt)*z + (
                    r-div-0.5*sigma**2)*dt)
                S[:,j] = S0*np.exp(logr)
            return S,

        def __antithetic_gbm(n_steps, size):
            dt = T/n_steps
            print("[antithetic gbm]: Initializing grids...")
            time.sleep(0.5)
            S = np.random.normal(size=(n_steps, size))
            S_a = -S
            bar = ProgressBar()
            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)
            return S, S_a,

        def __truncated_gbm(n_steps, size):
            # Importance sampling
            # truncate = K
            dt = T/n_steps
            L = (np.log(truncate/S0) - (
                r-div-0.5*sigma**2)*T) / (sigma*np.sqrt(T))
```

```
        Phi_L = norm.cdf(L)
        print("[truncated gbm]: warning ",
              "only final prices generated...")
        X = np.random.uniform(
            size=(2, size))*(1-Phi_L) + Phi_L
        X = norm.ppf(X)
        x = X[-1,:]
        logr = sigma*np.sqrt(T)*x + (
            r-div-0.5*sigma**2)*T
        X[-1,:] = S0*np.exp(logr)
        return X,

    def __stopped_gbm(n_steps, size):
        dt = T/n_steps
        print("[stopped gbm]: Initializing grids...")
        time.sleep(0.5)
        stopping_times = -np.ones(size, dtype=np.int8)
        rn_derivatives = np.ones(size)
        S = np.zeros((n_steps+1, size))
        S[0,:] = np.ones(size)*S0
        bar = ProgressBar()
        for j in bar(range(size)):
            for t in range(1,n_steps+1):
                z = np.random.normal(
                    loc=theta)
                logr = sigma*np.sqrt(dt)*z + (
                    r-div-0.5*sigma**2)*dt
                S[t,j] = S[t-1,j] * np.exp(logr)
                if stop(t, S[t,j]):
                    stopping_times[j] = t
                    if theta != 0:
                        rn_derivatives[j] *= np.exp(
                            (-theta*z)+(0.5*theta**2))
                    break
        return S[1:,:], stopping_times, rn_derivatives

    if stop is not None: return __stopped_gbm
    if truncate is not None: return __truncated_gbm
    return __antithetic_gbm if antithetic else __gbm

@staticmethod
def control_S_T(S0, tau, r):
    def __control(paths, paths_a=None):
        control = paths[-1,:]
        if paths_a is not None:
            control += paths_a[-1,:]
            control /= 2
        gbm_mean = S0*np.exp(r*tau)
```

```
            adj = np.mean(control) - gbm_mean
            return control, adj
        return __control

    @staticmethod
    def control_geometric_asian_call(
        S0, K, tau, sigma, r, div=0, n_steps=None):
        def __control(paths, paths_a=None):
            control = np.fmax(
                0, np.exp(-r*tau)*(
                stats.mstats.gmean(paths,axis=0)-K))
            if paths_a is not None:
                control += np.fmax(
                    0, np.exp(-r*tau)*(
                    stats.mstats.gmean(paths,axis=0)-K))
                control /= 2
            if n_steps is None:
                # asymptotic result
                sigma_star = sigma / np.sqrt(3)
                div_star = ((r+div)/2) + (sigma**2/12)
            else:
                N = n_steps
                sigma_star = sigma*np.sqrt((N+1)*(2*N+1)/(6*N**2))
                div_star = r*((N-1)/(2*N)) + div*((N+1)/(2*N)) + (
                    sigma**2 * ((N+1)*(N-1)/(12*N**2)))
            geom_asian_mean = bs(
                0, S0, K, tau, sigma_star, r, div_star)[0]
            adj = np.mean(control) - geom_asian_mean
            return control, adj
        return __control

    def __init__(self, sampler=None, payoff=None, control=None):
        self.path_sampler = sampler
        self.payoff = payoff
        self.control = control

    def estimate(self, n_steps, n_size):
        assert (self.path_sampler and self.payoff)
        sde_paths = self.path_sampler(n_steps, n_size)
        sample = self.payoff(*sde_paths)
        sample_mean, se = np.mean(sample), stats.sem(sample, ddof=0)
        if self.control is not None:
            control, adj = self.control(*sde_paths)
            cov_xy = np.cov(control, sample)
            rho = np.corrcoef(control, sample)[0,1]
            a_hat = -cov_xy[0,1]/cov_xy[0,0]
            sample_mean += a_hat * adj
            se *= np.sqrt(1-rho**2)
```

```
                return sample, sample_mean, se

            def reset_models(self, sampler=None, payoff=None, control=None):
                if sampler:
                    self.path_sampler = sampler
                if payoff:
                    self.payoff = payoff
                if control:
                    self.control= control
```

1 Antithetic Variables

1.1 Vanilla Call Standard MC

```
In [16]: table = []

In [17]: mc = MonteCarloEstimator(
             sampler = MonteCarloEstimator.gbm(
                 S0=100, T=1, r=0.05, sigma=0.1))

         for i, strike in enumerate([95, 100, 105]):
             mc.reset_models(
                 payoff = MonteCarloEstimator.vanilla_call(
                     K=strike, tau=1, r=0.05))
             _, price, se = mc.estimate(1000, 1000)
             table.append([strike, price, se])
             print('''Vanila call K={}:\n  Price = {}
             SE = {}\n'''.format(strike, price, se))
```

```
[gbm]: Initializing grids...


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


Vanila call K=95:
  Price = 10.644390155763 82
    SE = 0.27573932311576904

[gbm]: Initializing grids...


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


Vanila call K=100:
  Price = 6.764201837681756
```

```
    SE = 0.2468540780718818

[gbm]: Initializing grids...


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


Vanila call K=105:
  Price = 3.912480745292385
    SE = 0.19666380834128136
```

1.2 Vanilla Call Antithetic

```
In [18]: mc = MonteCarloEstimator(
             sampler = MonteCarloEstimator.gbm(
                 S0=100, T=1, r=0.05, sigma=0.1, antithetic=1))

         for i, strike in enumerate([95, 100, 105]):
             time.sleep(0.5)
             mc.reset_models(
                 payoff = MonteCarloEstimator.vanilla_call(
                     K=strike, tau=1, r=0.05))
             _, price, se = mc.estimate(1000, 500)
             table.append([strike, price, se])
             print('''Vanila call K={}:\n  Price = {}
             SE = {}\n'''.format(strike, price, se))
```

```
[antithetic gbm]: Initializing grids...


100% (500 of 500) |######################| Elapsed Time: 0:00:00 Time: 0:00:00


Vanila call K=95:
  Price = 10.364790598532108
    SE = 0.09406948852336663

[antithetic gbm]: Initializing grids...


100% (500 of 500) |######################| Elapsed Time: 0:00:00 Time: 0:00:00


Vanila call K=100:
  Price = 6.58884626385324
    SE = 0.1301069737385625
```

```
[antithetic gbm]: Initializing grids...


100% (500 of 500) |######################| Elapsed Time: 0:00:00 Time: 0:00:00


Vanila call K=105:
  Price = 4.12154917739265
    SE = 0.1540737723237456
```

```
In [23]: index = [['Standard MC', 'Antithetic Variables'], [1,2,3]]
         index = pd.MultiIndex.from_product(index, names=['Method', 'case'])
         summary = pd.DataFrame(table, columns=[
             'Strike', 'Price', 'SE'], index=index)
         summary
```

Out[23]:

Method	case	Strike	Price	SE
Standard MC	1	95	10.644390	0.275739
	2	100	6.764202	0.246854
	3	105	3.912481	0.196664
Antithetic Variables	1	95	10.364791	0.094069
	2	100	6.588846	0.130107
	3	105	4.121549	0.154074

Clearly, the Antithetic Variables method has lower standard error in all cases, even if the sample size get halved (n=500) to ensure same computation cost.

2 Arithmetic Asian option

2.1 Arithmetic Asian Call Standard MC

```
In [24]: table = []
         mc = MonteCarloEstimator(
             sampler = MonteCarloEstimator.gbm(
                 S0=100, T=1, r=0.05, sigma=0.1))
         mc.reset_models(
             payoff = MonteCarloEstimator.arithmetic_asian_call(
                 K=100, tau=1, r=0.05))
         time.sleep(0.5)
         _, price, se = mc.estimate(52, 1000)
         table.append(['Base MC', price, se])
         print('''Arithmetic Asian Call K={}:\n  Price = {}
           SE = {}\n'''.format(100, price, se))
```

```
[gbm]: Initializing grids...
```

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


Arithmetic Asian Call K=100:
  Price = 3.5908349161560364
  SE = 0.13356079852260308
```

2.2 Arithmetic Asian Call Antithetic

```
In [25]: mc = MonteCarloEstimator(
             sampler = MonteCarloEstimator.gbm(
                 S0=100, T=1, r=0.05, sigma=0.1, antithetic=1))
         mc.reset_models(
             payoff = MonteCarloEstimator.arithmetic_asian_call(
                 K=100, tau=1, r=0.05))
         time.sleep(0.5)
         _, price, se = mc.estimate(52, 500)
         table.append(['Antithetic Variables', price, se])
         print('''Arithmetic Asian Call K={}:\n  Price = {}
           SE = {}\n'''.format(100, price, se))
```

```
[antithetic gbm]: Initializing grids...


100% (500 of 500) |######################| Elapsed Time: 0:00:00 Time: 0:00:00


Arithmetic Asian Call K=100:
  Price = 3.5805053875720962
  SE = 0.07258388090957411
```

2.3 Arithmetic Asian Call, S_T Controlled

```
In [26]: mc = MonteCarloEstimator(
             sampler = MonteCarloEstimator.gbm(
                 S0=100, T=1, r=0.05, sigma=0.1),
             payoff = MonteCarloEstimator.arithmetic_asian_call(
                 K=100, tau=1, r=0.05),
             control = MonteCarloEstimator.control_S_T(
                 S0=100, tau=1, r=0.05)
         )

         _, price, se = mc.estimate(52, 1000)
         table.append(['ST Controlled', price, se])
         print('''Arithmetic Asian Call K={}:\n  Price = {}
           SE = {}\n'''.format(100, price, se))
```

```
[gbm]: Initializing grids...


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


Arithmetic Asian Call K=100:
  Price = 3.7479105355126965
  SE = 0.07804828110527139
```

2.4 Arithmetic Asian Call, Geometric Asian Call Controlled (Asymptotic Parameters)

```
In [27]: mc = MonteCarloEstimator(
             sampler = MonteCarloEstimator.gbm(
                 S0=100, T=1, r=0.05, sigma=0.1, antithetic=1),
             payoff = MonteCarloEstimator.arithmetic_asian_call(
                 K=100, tau=1, r=0.05),
             control = MonteCarloEstimator.control_geometric_asian_call(
                 S0=100, K=100, tau=1, sigma=0.1, r=0.05)
         )

         _, price, se = mc.estimate(52, 1000)
         table.append(['Geom.Asian Call Controlled (Asymptotic Formula)',
                       price, se])
         print('''Arithmetic Asian Call K={}:\n  Price = {}
           SE = {}\n'''.format(100, price, se))
```

```
[antithetic gbm]: Initializing grids...


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


Arithmetic Asian Call K=100:
  Price = 3.6697697761644346
  SE = 0.04916007093518221
```

2.5 Arithmetic Asian Call, Geometric Asian Call Controlled (True Parameters)

```
In [28]: mc = MonteCarloEstimator(
             sampler = MonteCarloEstimator.gbm(
                 S0=100, T=1, r=0.05, sigma=0.1, antithetic=1),
             payoff = MonteCarloEstimator.arithmetic_asian_call(
                 K=100, tau=1, r=0.05),
```

```
    control = MonteCarloEstimator.control_geometric_asian_call(
        S0=100, K=100, tau=1, sigma=0.1, r=0.05, n_steps=52)
)

_, price, se = mc.estimate(52, 1000)
table.append(['Geom.Asian Call Controlled (True Formula)',
              price, se])
print('''Arithmetic Asian Call K={}:\n  Price = {}
  SE = {}\n'''.format(100, price, se))
```

```
[antithetic gbm]: Initializing grids...


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


Arithmetic Asian Call K=100:
  Price = 3.6215009915348864
  SE = 0.04835164879616438
```

In [29]:
```
summary = pd.DataFrame(table, columns=[
    'Method', 'Price', 'SE'])
summary
```

Out[29]:

	Method	Price	SE
0	Base MC	3.590835	0.133561
1	Antithetic Variables	3.580505	0.072584
2	ST Controlled	3.747911	0.078048
3	Geom.Asian Call Controlled (Asymptotic Formula)	3.669770	0.049160
4	Geom.Asian Call Controlled (True Formula)	3.621501	0.048352

The summary is provided above. As we can see, using geometric Asian call option as the control variable is the most effective variance reduction method. They achieved a standard error ~0.05 for 1000 samples. Using S_T as control variable is about as effective as antithetic variable method, in that they reduced standard error from ~0.13 to ~0.08.

3 Control Variables and Importance Sampling

3.1 Standard MC

In [30]:
```
table = []
mc = MonteCarloEstimator(
    sampler = MonteCarloEstimator.gbm(
        S0=100, T=1, r=0.05, sigma=0.2))

for i, strike in enumerate([120, 140, 160]):
    mc.reset_models(
```

```
                payoff = MonteCarloEstimator.vanilla_call(
                    K=strike, tau=1, r=0.05))
            _, price, se = mc.estimate(100, 10000)
            table.append([strike, price, se])
            print('''Vanila call K={}:\n  Price = {}
            SE = {}\n'''.format(strike, price, se))
```

```
[gbm]: Initializing grids...


100% (10000 of 10000) |###################| Elapsed Time: 0:00:00 Time: 0:00:00


Vanila call K=120:
  Price = 3.2247931596217136
    SE = 0.08527096304049593

[gbm]: Initializing grids...


100% (10000 of 10000) |###################| Elapsed Time: 0:00:00 Time: 0:00:00


Vanila call K=140:
  Price = 0.8250017838247197
    SE = 0.04359696086769234

[gbm]: Initializing grids...


100% (10000 of 10000) |###################| Elapsed Time: 0:00:00 Time: 0:00:00


Vanila call K=160:
  Price = 0.1350313379765521
    SE = 0.015399145719972753
```

3.2 Put-Call Parity

```
In [31]: mc = MonteCarloEstimator(
            sampler = MonteCarloEstimator.gbm(
                S0=100, T=1, r=0.05, sigma=0.2))

        for i, strike in enumerate([120, 140, 160]):
            mc.reset_models(
                payoff = MonteCarloEstimator.vanilla_call_parity(
                    K=strike, S0=100, tau=1, r=0.05))
```

```
        _, price, se = mc.estimate(100, 10000)
        table.append([strike, price, se])
        print('''Vanila call K={}:\n  Price = {}
        SE = {}\n'''.format(strike, price, se))
```

```
[gbm]: Initializing grids...


100% (10000 of 10000) |###################| Elapsed Time: 0:00:00 Time: 0:00:00


Vanila call K=120:
  Price = 3.044323974221644
    SE = 0.14629904121306525

[gbm]: Initializing grids...


100% (10000 of 10000) |###################| Elapsed Time: 0:00:00 Time: 0:00:00


Vanila call K=140:
  Price = 0.7565303104723016
    SE = 0.18306090971823263

[gbm]: Initializing grids...


100% (10000 of 10000) |###################| Elapsed Time: 0:00:00 Time: 0:00:00


Vanila call K=160:
  Price = 0.479728696655923
    SE = 0.1957533174577639
```

3.3 Put-Call Parity with Control

```
In [32]: mc = MonteCarloEstimator(
             sampler = MonteCarloEstimator.gbm(
                 S0=100, T=1, r=0.05, sigma=0.2),
             control = MonteCarloEstimator.control_S_T(
                 S0=100, tau=1, r=0.05)
         )

         for i, strike in enumerate([120, 140, 160]):
             mc.reset_models(
                 payoff = MonteCarloEstimator.vanilla_call_parity(
```

```
                K=strike, S0=100, tau=1, r=0.05))
        _, price, se = mc.estimate(100, 10000)
        table.append([strike, price, se])
        print('''Vanila call K={}:\n  Price = {}
        SE = {}\n'''.format(strike, price, se))
```

```
[gbm]: Initializing grids...


100% (10000 of 10000) |###################| Elapsed Time: 0:00:00 Time: 0:00:00


Vanila call K=120:
  Price = 3.277791627801777
    SE = 0.055764034136161975

[gbm]: Initializing grids...


100% (10000 of 10000) |###################| Elapsed Time: 0:00:00 Time: 0:00:00


Vanila call K=140:
  Price = 0.8257040298499927
    SE = 0.03656413201552896

[gbm]: Initializing grids...


100% (10000 of 10000) |###################| Elapsed Time: 0:00:00 Time: 0:00:00


Vanila call K=160:
  Price = 0.1544671655280099
    SE = 0.017664413806660905
```

3.4 Importance Sampling: Option Pays off with Certainty

```
In [33]: for i, strike in enumerate([120, 140, 160]):
             mc = MonteCarloEstimator(
                 sampler = MonteCarloEstimator.gbm(
                     S0=100, T=1, r=0.05,
                     sigma=0.2, truncate=strike
                 ),
                 payoff = MonteCarloEstimator.vanilla_call(
                     K=strike, tau=1,
                     r=0.05,
```

```
                L=(np.log(strike/100) - (
                    0.05-0-0.5*0.2**2)*1) / (
                    0.2*np.sqrt(1))
            )
        )
        _, price, se = mc.estimate(100, 10000)
        table.append([strike, price, se])
        print('''Vanila call K={}:\n  Price = {}
        SE = {}\n'''.format(strike, price, se))
```

```
[truncated gbm]: warning  only final prices generated...
Vanila call K=120:
  Price = 3.220183859691146
    SE = 0.02920918865832503

[truncated gbm]: warning  only final prices generated...
Vanila call K=140:
  Price = 0.7713834526898626
    SE = 0.007315384607125251

[truncated gbm]: warning  only final prices generated...
Vanila call K=160:
  Price = 0.1568889896801693
    SE = 0.001504411777608655
```

In [34]:
```
index = [['Standard MC',
            'Put-Call Parity',
            'Parity & Control Variable',
            'Importance Sampling'], [1,2,3]]
index = pd.MultiIndex.from_product(index, names=['Method', 'Case'])
summary = pd.DataFrame(table, columns=[
    'Strike', 'Price', 'SE'], index=index)
summary
```

Out[34]:

Method	Case	Strike	Price	SE
Standard MC	1	120	3.224793	0.085271
	2	140	0.825002	0.043597
	3	160	0.135031	0.015399
Put-Call Parity	1	120	3.044324	0.146299
	2	140	0.756530	0.183061
	3	160	0.479729	0.195753
Parity & Control Variable	1	120	3.277792	0.055764
	2	140	0.825704	0.036564
	3	160	0.154467	0.017664
Importance Sampling	1	120	3.220184	0.029209

```
2         140  0.771383   0.007315
3         160  0.156889   0.001504
```

Using put-call parity itself doesn't reduce variance at all, but when combined with control variable S_T, the variance of the first two cases is fairly reduced, since S_T is highly correlated with the put price. Importance sampling is the most effective method to reduce variance for this problem. It brings down the standard error in all three cases very significantly.

4 Digital Payoff Barrier Options

4.1 Standard Monte Carlo

```
In [35]: table = []
         for i, param in enumerate([(94,96) ,(90,96), (85,96), (90,106)]):
             H, K = param
             mc = MonteCarloEstimator(
                 sampler = MonteCarloEstimator.gbm(
                     S0=95, T=0.25, r=0.05,
                     sigma=0.15),
                 payoff = MonteCarloEstimator.down_in_digital(
                     K=K, H=H, tau=0.25, r=0.05)
             )
             _, price, se = mc.estimate(50, 100000)
             table.append([K, H, price, se])
             print('''Down&In Digital K={}, H={}:\n  Price = {}
             SE = {}\n'''.format(K, H, price, se))
```

```
[gbm]: Initializing grids...


100% (100000 of 100000) |#################| Elapsed Time: 0:00:01 Time: 0:00:01


Down&In Digital K=96, H=94:
  Price = 3038.5793765595754
    SE = 14.413663915061488

[gbm]: Initializing grids...


100% (100000 of 100000) |#################| Elapsed Time: 0:00:01 Time: 0:00:01


Down&In Digital K=96, H=90:
  Price = 420.11559633009716
    SE = 6.302754359396163

[gbm]: Initializing grids...
```

```
100% (100000 of 100000) |################| Elapsed Time: 0:00:01 Time: 0:00:01


Down&In Digital K=96, H=85:
  Price = 6.814286823407782
    SE = 0.8200606649818247

[gbm]: Initializing grids...


100% (100000 of 100000) |################| Elapsed Time: 0:00:01 Time: 0:00:01


Down&In Digital K=106, H=90:
  Price = 13.727331426864954
    SE = 1.1635275593806762
```

4.2 Conditional Monte Carlo

```
In [38]: for i, param in enumerate([(94,96) ,(90,96), (85,96), (90,106)]):
             H, K = param
             mc = MonteCarloEstimator(
                 sampler = MonteCarloEstimator.gbm(
                     S0=95, T=0.25, r=0.05,
                     sigma=0.15,
                     stop=lambda t,S: S<H),
                 payoff = MonteCarloEstimator.down_in_digital(
                     K=K, H=H, tau=0.25, r=0.05,
                     stop=True, sigma=0.15
                 )
             )
             payoffs, price, se = mc.estimate(50, 100000)
             table.append([K, H, price, se])
             print('''Down&In Digital K={}, H={}:\n  Price = {}
             SE = {}\n'''.format(K, H, price, se))
```

```
[stopped gbm]: Initializing grids...


100% (100000 of 100000) |################| Elapsed Time: 0:00:13 Time: 0:00:13
/Users/zed/anaconda/lib/python3.6/site-packages/ipykernel_launcher.py:15: RuntimeWarning: divide
  from ipykernel import kernelapp as app


Down&In Digital K=96, H=94:
  Price = 3017.656775621864
    SE = 5.060121107567195
```

```
[stopped gbm]: Initializing grids...


100% (100000 of 100000) |#################| Elapsed Time: 0:00:35 Time: 0:00:35


Down&In Digital K=96, H=90:
  Price = 428.94245397284635
    SE = 2.1080537490729907

[stopped gbm]: Initializing grids...


100% (100000 of 100000) |#################| Elapsed Time: 0:00:44 Time: 0:00:44


Down&In Digital K=96, H=85:
  Price = 5.5628905430268905
    SE = 0.09876890008477911

[stopped gbm]: Initializing grids...


100% (100000 of 100000) |#################| Elapsed Time: 0:00:33 Time: 0:00:33


Down&In Digital K=106, H=90:
  Price = 13.346183147268253
    SE = 0.09276963553590656
```

```
In [58]: index = [['Standard MC',
                   'Conditional MC'], [1,2,3,4]]
         index = pd.MultiIndex.from_product(index, names=['Method', 'Case'])
         summary = pd.DataFrame(table, columns=[
             'Strike', 'Barrier', 'Price', 'SE'], index=index)
         ratio = list(summary['SE']['Standard MC']/summary['SE']['Conditional MC'])
         summary['Variance Ratio'] = [np.nan]*4+[x**2 for x in ratio]
         summary
```

Out[58]:

Method	Case	Strike	Barrier	Price	SE	Variance Ratio
Standard MC	1	96	94	3038.579377	14.413664	NaN
	2	96	90	420.115596	6.302754	NaN
	3	96	85	6.814287	0.820061	NaN
	4	106	90	13.727331	1.163528	NaN
Conditional MC	1	96	94	3017.656776	5.060121	8.113850

2	96	90	428.942454	2.108054	8.939174
3	96	85	5.562891	0.098769	68.936865
4	106	90	13.346183	0.092770	157.304682

The digital call closed form formula is derived by

$$\begin{aligned} c(t,x) &= \tilde{E}[e^{-r(T-t)}\mathbb{1}_{\{S_T>K\}}|S_t = x] \\ &= e^{-r(T-t)}\mathbb{P}(S_T > K|S_t = x) \\ &= e^{-r(T-t)}\mathbb{P}\left(S_t \exp\left((r-\frac{1}{2}\sigma^2)(T-t) + \sigma(\tilde{W}_T - \tilde{W}_t)\right) > K\middle|S_t = x\right) \\ &= e^{-r(T-t)}\mathbb{P}\left(Z > \frac{\log\frac{x}{K} + (r-\frac{1}{2}\sigma^2)(T-t)}{\sigma\sqrt{T-t}}\right) \end{aligned}$$

Where $\tilde{W}_T - \tilde{W}_t \sim N(0, T-t)$, $Z \sim N(0,1)$. Therefore,

$$c(t,x) = e^{-r(T-t)}N(d_2)$$

Where

$$d_2 = \frac{\log\frac{x}{K} + (r-\frac{1}{2}\sigma^2)(T-t)}{\sigma\sqrt{T-t}}; \qquad S_t = x$$

The standard error comparison is given in the summary table above.

5 Discrete V.S. Continuous Pricing

5.1 Standard Monte Carlo

```
In [59]: table = []
         mc = MonteCarloEstimator(
             sampler = MonteCarloEstimator.gbm(
                 S0=100, T=0.2, r=0.1,
                 sigma=0.3),
             payoff = MonteCarloEstimator.down_in_call(
                 K=100, H=95, tau=0.2, r=0.1)
         )

         for i, N in enumerate([25, 50, 100]):
             _, price, se = mc.estimate(N, 100000)
             table.append([N, price, se])
             print('''Down&In Call K={}, H={}:\n  Price = {}
             SE = {}\n'''.format(100, 95, price, se))
```

```
[gbm]: Initializing grids...


100% (100000 of 100000) |#################| Elapsed Time: 0:00:01 Time: 0:00:01
```

```
Down&In Call K=100, H=95:
  Price = 1.2548986811091567
    SE = 0.012415750337300278

[gbm]: Initializing grids...


100% (100000 of 100000) |################| Elapsed Time: 0:00:01 Time: 0:00:01


Down&In Call K=100, H=95:
  Price = 1.4103013976243566
    SE = 0.013286104100301455

[gbm]: Initializing grids...


100% (100000 of 100000) |################| Elapsed Time: 0:00:01 Time: 0:00:01


Down&In Call K=100, H=95:
  Price = 1.5758218133375175
    SE = 0.014047002929688102
```

5.2 Conditional Monte Carlo

```
In [60]: mc = MonteCarloEstimator(
             sampler = MonteCarloEstimator.gbm(
                 S0=100, T=0.2, r=0.1,
                 sigma=0.3,
                 stop=lambda t,S: S<95),
             payoff = MonteCarloEstimator.down_in_call(
                 K=100, H=95, tau=0.2, r=0.1,
                 stop=True, sigma=0.3)
         )

         for i, N in enumerate([25, 50, 100]):
             _, price, se = mc.estimate(N, 100000)
             table.append([N, price, se])
             print('''Down&In Call K={}, H={}:\n  Price = {}
             SE = {}\n'''.format(100, 95, price, se))
```

```
[stopped gbm]: Initializing grids...


100% (100000 of 100000) |################| Elapsed Time: 0:00:12 Time: 0:00:12
/Users/zed/anaconda/lib/python3.6/site-packages/ipykernel_launcher.py:5: RuntimeWarning: divide
  """
```

```
Down&In Call K=100, H=95:
  Price = 1.2564675105671148
    SE = 0.004006626066561751

[stopped gbm]: Initializing grids...


100% (100000 of 100000) |#################| Elapsed Time: 0:00:25 Time: 0:00:25


Down&In Call K=100, H=95:
  Price = 1.4396341513291386
    SE = 0.004247912439035337

[stopped gbm]: Initializing grids...


100% (100000 of 100000) |#################| Elapsed Time: 0:00:46 Time: 0:00:46


Down&In Call K=100, H=95:
  Price = 1.5718324927881755
    SE = 0.004414490720001968
```

5.3 Conditional MonteCarlo with Importance Sampling

```
In [61]: mc = MonteCarloEstimator(
             sampler = MonteCarloEstimator.gbm(
                 S0=100, T=0.2, r=0.1,
                 sigma=0.3,
                 stop=lambda t,S: S<95,
                 theta=-0.45),
             payoff = MonteCarloEstimator.down_in_call(
                 K=100, H=95, tau=0.2, r=0.1,
                 stop=True, sigma=0.3)
         )

         _, price, se = mc.estimate(25, 100000)
         table.append([N, price, se])
         print('''Down&In Call K={}, H={}:\n  Price = {}
         SE = {}\n'''.format(100, 95, price, se))
```

```
[stopped gbm]: Initializing grids...


100% (100000 of 100000) |#################| Elapsed Time: 0:00:05 Time: 0:00:05
/Users/zed/anaconda/lib/python3.6/site-packages/ipykernel_launcher.py:5: RuntimeWarning: divide
  """
```

```
Down&In Call K=100, H=95:
  Price = 1.390315773227041
SE = 0.0021916865816488597
```

```
In [62]: mc = MonteCarloEstimator(
             sampler = MonteCarloEstimator.gbm(
                 S0=100, T=0.2, r=0.1,
                 sigma=0.3,
                 stop=lambda t,S: S<95,
                 theta=-0.3),
             payoff = MonteCarloEstimator.down_in_call(
                 K=100, H=95, tau=0.2, r=0.1,
                 stop=True, sigma=0.3)
         )
         for N in [50, 100]:
             _, price, se = mc.estimate(N, 100000)
             table.append([N, price, se])
             print('''Down&In Call K={}, H={}:\n  Price = {}
             SE = {}\n'''.format(100, 95, price, se))
```

```
[stopped gbm]: Initializing grids...


100% (100000 of 100000) |#################| Elapsed Time: 0:00:11 Time: 0:00:11
/Users/zed/anaconda/lib/python3.6/site-packages/ipykernel_launcher.py:5: RuntimeWarning: divide
  """


Down&In Call K=100, H=95:
  Price = 1.7715332800748584
    SE = 0.0022090943425283538

[stopped gbm]: Initializing grids...


100% (100000 of 100000) |#################| Elapsed Time: 0:00:13 Time: 0:00:13


Down&In Call K=100, H=95:
  Price = 2.0193625330494527
    SE = 0.001907217981900248
```

$N \to \infty$

```
In [64]: bs_down_in(0, 100, 100, 95, 0.2, 0.3, 0.1)
```

Out[64]: 1.9466109033299226

In the continous case ($N \to \infty$), the price is given by Hull's formula, which is 1.9466 in this setting. Simulation summary is presented in the table below. Note that conditional MC reduced the standard error by about 60%, from ~0.13 to ~0.0045. And the change of measure method further reduce about half of the standard error on that basis.

As N increase, the simulated price approaches the limit case given by the exact formula.

In [63]:
```
index = [['Standard MC',
          'Conditional MC',
          'Conditional MC with Change of Measure'], [1,2,3]]
index = pd.MultiIndex.from_product(index, names=['Method', 'Case'])
summary = pd.DataFrame(table, columns=[
    '# Evaluation Points', 'Price', 'SE'], index=index)
summary
```

Out[63]:
```
                                           # Evaluation Points     Price  \
Method                                Case
Standard MC                           1                     25  1.254899
                                      2                     50  1.410301
                                      3                    100  1.575822
Conditional MC                        1                     25  1.256468
                                      2                     50  1.439634
                                      3                    100  1.571832
Conditional MC with Change of Measure 1                    100  1.390316
                                      2                     50  1.771533
                                      3                    100  2.019363


                                                  SE
Method                                Case
Standard MC                           1     0.012416
                                      2     0.013286
                                      3     0.014047
Conditional MC                        1     0.004007
                                      2     0.004248
                                      3     0.004414
Conditional MC with Change of Measure 1     0.002192
                                      2     0.002209
                                      3     0.001907
```

In [ ]: