jgiebas_HW5

February 22, 2018

1 46-932, Simulation Methods for Option Pricing: Homework 5

Author: Jordan Giebas *Due Date*: Feb. 22, 2018

1.1 Question 1: *Practice on Stratification*

Let there be two assets following GBMs,

$$dS_i(t) = rS_i(t)dt + \sigma_i S_i(t)dW_i(t), \;\; i \in \{1, 2\}$$

driven by independent brownian motions.
We would like to price a call option on the average of the two stock prices,

$$payoff = \left(\frac{S_1(T) + S_1(T)}{2} - K \right)^+$$

The parameters are given as,
$S_i(0) = K = 100$
$\sigma_i = 0.20$
$T = 1$
$r = 0.05$
$n = 10000$

1.1.1 Part (a)

Price the option using standard Monte Carlo simulation, drawing $n = 10000$ pais of independent uniform random variables and converting them to standard normals when simulating the price process. Get the discounted payoff, and provide an estimate and standard error for the simulation.

```
In [52]: # import needed packages
         import pandas as pd
         import numpy as np
         import numpy.random as npr
         import scipy
         from scipy.stats import norm


         ## Define Problem 1 parameters
```

```
S_1_init = S_2_init = K = 100
sig_1 = sig_2 = 0.20
T = 1
r = 0.05
nopaths = 10000
```

```
In [60]: # Initialize S_1 dataframe
         S1_pp = pd.DataFrame(columns=pd.RangeIndex(start=1,stop=10001))
         S1_pp.loc[0,:] = S_1_init
         # Populate using transition
         S1_pp.loc[1,:] = S1_pp.loc[0,:]*np.exp( (r-0.5*sig_1**2)*T + sig_1*np.sqrt(T)*norm.ppf
         S1_T = np.array(S1_pp.loc[1,:])

         # Initialize S_2 dataframe
         S2_pp = pd.DataFrame(columns=pd.RangeIndex(start=1,stop=10001))
         S2_pp.loc[0,:] = S_2_init
         # Populate using transition
         S2_pp.loc[1,:] = S2_pp.loc[0,:]*np.exp( (r-0.5*sig_1**2)*T + sig_1*np.sqrt(T)*norm.ppf
         S2_T = np.array(S2_pp.loc[1,:])

         # Get scenario payoffs
         c_i = np.maximum(np.add(S1_T,S2_T)/2.0 - K, 0)

         # Print results
         print("The estimate is: %f,\nStd. Error: %f" % (np.mean(c_i), np.std(c_i)/np.sqrt(1000
```

```
The estimate is: 8.566763,
Std. Error: 0.110228
```

1.1.2 Part (b):

```
In [61]: ## Make a function to get an estimate in each cell.
         def get_ci( x, y ):

             # Store vectors of initial prices
             S1_0 = np.full(10,100.0)
             S2_0 = np.full(10,100.0)

             # Generate standard normals
             Z_s1 = norm.ppf(npr.uniform((x-1)/10.0, x/10.0, size=10))
             Z_s2 = norm.ppf(npr.uniform((y-1)/10.0, y/10.0, size=10))

             # Iterate to T
             S1_T = [s_0*np.exp( (r-0.5*sig_1**2)*T + sig_1*np.sqrt(T)*z )  for s_0, z in zip(S
             S2_T = [s_0*np.exp( (r-0.5*sig_1**2)*T + sig_1*np.sqrt(T)*z )  for s_0, z in zip(S

             # Add them element wise
```

```
        c_i = np.maximum(np.add(S1_T,S2_T)/2.0 - K,0)

        return np.mean(c_i), np.var(c_i)
```

In [62]:
```
# Create a dataframe for this
unit_rec = pd.DataFrame(index=pd.RangeIndex(start=1,stop=11), columns=pd.RangeIndex(s
var_rec = pd.DataFrame(index=pd.RangeIndex(start=1,stop=11), columns=pd.RangeIndex(sta
```

In [63]:
```
for i in unit_rec.index:
    for j in unit_rec.columns:

        mean, var = get_ci(i,j)
        unit_rec.loc[i,j] = mean
        var_rec.loc[i,j] = var
```

In [64]:
```
unit_rec
```

Out[64]:

	1	2	3	4	5	6	7 \
1	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0.754349
4	0	0	0	0	0	0.651867	3.10217
5	0	0	0	0	0.781699	3.38827	6.04422
6	0	0	0.0149295	0.984532	3.5769	5.57144	8.76887
7	0	0.00452322	0.842135	3.27453	5.26103	8.9632	11.2254
8	0	1.52392	4.04221	6.58375	9.4063	11.8268	14.2017
9	0.446264	6.23509	7.28875	10.5232	14.0641	17.3817	19.6627
10	6.65056	18.2529	15.8395	18.3531	23.1908	26.1456	31.9829

	8	9	10
1	0	1.24438	14.51
2	1.17814	4.97125	17.3585
3	4.04898	8.18091	18.1828
4	6.52364	12.2022	19.248
5	9.65187	13.3329	22.3706
6	11.6524	17.702	23.7194
7	14.5424	19.2239	28.8876
8	17.6979	22.8627	32.8763
9	22.6776	27.6966	35.7037
10	35.268	38.9812	47.1638

In [65]:
```
print("Estimated Option Price: %f" % (unit_rec.values.sum()/100.0))
print("Standard Error: %f" % (np.sqrt(var_rec.values.sum()/100)/np.sqrt(1000)))
```

```
Estimated Option Price: 8.764716
Standard Error: 0.090049
```

In [66]:
```
var_rec
```

Out[66]:

	1	2	3	4	5	6	7
1	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0.330925
4	0	0	0	0	0	0.498215	0.700517
5	0	0	0	0	0.722359	0.79529	0.909509
6	0	0	0.00109086	0.469828	1.02801	1.84679	1.15383
7	0	0.000184135	0.426851	0.357485	1.44612	1.67727	1.21439
8	0	1.06593	2.81048	1.30827	1.46094	1.81873	2.76787
9	0.549008	1.84083	1.79802	1.53255	2.80554	3.74244	3.53676
10	26.3616	48.6244	7.79761	15.6637	20.9906	33.0347	172.84

	8	9	10
1	0	3.05666	55.4658
2	1.23337	3.65016	42.4682
3	2.48679	3.90336	43.6117
4	1.17343	4.16599	19.0486
5	1.98087	1.43973	8.13457
6	0.597593	0.870188	7.18029
7	2.58077	3.73287	34.323
8	2.27553	2.86014	65.8581
9	1.58911	5.87939	10.7714
10	30.0246	66.8275	17.7714

1.1.3 Part (c)

Now we condition on a projection. Divide the unit interval into 250 equiprobable bins. Draw four random uniforms with boundaries of the bin, and convert into standard (conditional) normals using the Probability Integral Transform. Follow the procedure from the notes and Glasserman, with

$$\nu^T = \frac{1}{\sqrt{2}}(1,1)$$

In [67]:
```
# Getting the payoffs using stratification about a projection
def pr1c_ci( ix ):

    # Define bounds to get correct Uniform
    low  = (ix-1)/B
    high = ix/B

    # Get a random uniform vector of dimension 4
    U_vec = npr.uniform(low, high, size=4)

    # Probability integral transform
    Z_vec = norm.ppf(U_vec)

    # For each of these variables, we conditon and use the glasserman formula
    C_vec = list()
```

```
        for z in Z_vec:

            # Define parameters for multivariatenormal dist
            nu_nuT = 0.5*np.ones(shape=(2,2))
            mu_vec     = 1/np.sqrt(2)*z*np.array([1,1])
            cov_matrix = np.eye(2,2) - nu_nuT

            # Glasserman formula for conditional distribution
            Z_1, Z_2 = npr.multivariate_normal(mu_vec, cov_matrix)

            # Generate S_1, S_2
            S1_T = S_1_init*np.exp((r-0.5*sig_2**2)*T + sig_2*np.sqrt(T)*Z_1)
            S2_T = S_2_init*np.exp((r-0.5*sig_2**2)*T + sig_2*np.sqrt(T)*Z_2)

            # Get discounted payoff
            c_i = np.exp(-r*T)*np.maximum(np.add(S1_T,S2_T)/2.0 - K,0)
            C_vec.append(c_i)

        return np.mean(C_vec), np.var(C_vec)
```

```
In [68]: # Stratification along the projection conditional on the terminal price
         B = 250
         payoff_list = list()
         var_list = list()
         for i in range(1,B+1):

             mean, var = pr1c_ci(i)
             payoff_list.append( mean )
             var_list.append( var )
```

```
In [71]: print("Estimated Option Price: %f" % (np.mean(payoff_list)))
         print("Standard Error: %f" % (np.mean(var_list)/np.sqrt(1000)))
```

```
Estimated Option Price: 8.299220
Standard Error: 0.035287
```

1.2 Question 2: *Practice on Brownian Bridge Method*

1.2.1 Part (a)

Refer to Table 1 pg. 64 of the paper by Beaglehole, Dybvig, and Zhou "Going to extremes: Correcting simulation bias in exotic option valuation". We are going to replicate some results in the paper using

$T = 0.25$

$N = 30$

$n = 1000$

Also, replicate the corresponding entries for the Brownian Bridge simulation.

```
In [34]: # Global Vars
         T = 0.25
         N = 30
         dt = T/N
         nopaths = 1000
         S_0 = 50
         vol = 0.25
         r = 0.10

         # Low : Previous discrete stock price
         # High: Next discrete stock price
         # Return prob. integral transform
         def sample_bbDist( low, high, delta ):

             # Define b as in paper/notes
             b = (high-low)/(vol*low)

             return ( (b + np.sqrt(b**2 - 2*delta*np.log(npr.uniform())))/2.0 )

         # Low : Previous discrete stock price
         # High: Next discrete stock price
         # Return prob. integral transform
         def Min_bbDist( low, high, delta ):

             # Define b as in paper/notes
             b = (high-low)/(sig_2*low)

             return ( (b - np.sqrt(b**2 - 2*delta*np.log(npr.uniform())))/2.0 )

In [35]: # Construct stock dataframe
         stock_df = pd.DataFrame(index=pd.RangeIndex(0,N+1), columns=pd.RangeIndex(1,nopaths+1

         # t0 - S_0 = 100
         stock_df.loc[0,:] = S_0

         # Iterate using Euler Scheme
         for i in range(1,len(stock_df.index)):
             Z = np.random.standard_normal(size=nopaths)
             stock_df.loc[i,:] = stock_df.loc[i-1,:] + r*stock_df.loc[i-1,:]*dt + vol*stock_df

         for col in stock_df.columns:
             stock_df.loc[31,col] = stock_df[col].max()

In [36]: # Construct a dataframe for modeling in between discrete times
         # Row i corresponds to max S_t \in [t_i, t_{i+1}]
         bbridge = pd.DataFrame(index=pd.RangeIndex(1,N+1), columns=pd.RangeIndex(1,nopaths+1)

         for i in range(1,len(bbridge)+1):
```

```
            for j in range(1,nopaths+1):

                max_value = sample_bbDist( stock_df.loc[i-1,j], stock_df.loc[i,j], dt )
                bbridge.loc[i,j] = stock_df.loc[i-1,j]*(1 + vol*max_value)

        # Populate the last row with the maximums of the appropriate columns
        for k in range(1,nopaths+1):
            bbridge.at[31, k] = bbridge[k].max()
```

```
In [37]: pr2a_payoff_noBB = [ np.maximum(M-S_0, 0) for M in stock_df.loc[31,:] ]

         print("Estimate (w/out BB): ", np.exp(-r*T)*np.mean(pr2a_payoff_noBB))
         print("Standard Deviation: ", np.std(pr2a_payoff_noBB)/np.sqrt(nopaths))
```

```
Estimate (w/out BB):  5.24361538326
Standard Deviation:  0.149604357781
```

```
In [38]: pr2a_payoff_BB = [ np.maximum(elm-S_0, 0) for elm in bbridge.loc[31] ]

         print("Estimate (w/ BB): ", np.exp(-r*T)*np.mean(pr2a_payoff_BB))
         print("Standard Deviation: ", np.std(pr2a_payoff_BB)/np.sqrt(nopaths))
```

```
Estimate (w/ BB):  5.89875165191
Standard Deviation:  0.151519640574
```

1.2.2 Part (b)

We use the same constructs as in part (a), but simply change the strike price to be the terminal stock price

```
In [39]: pr2b_payoff_noBB = [np.maximum(M-S_T, 0) for M, S_T in zip(stock_df.loc[31,:],stock_d

         print("Estimate (w/out BB): ", np.exp(-r*T)*np.mean(pr2b_payoff_noBB))
         print("Standard Deviation: ", np.std(pr2b_payoff_noBB)/np.sqrt(nopaths))
```

```
Estimate (w/out BB):  3.77460746766
Standard Deviation:  0.106387458381
```

```
In [40]: pr2b_payoff_BB = [np.maximum(M-K, 0) for M, K in zip(bbridge.loc[31],stock_df.loc[30,

         print("Estimate: ", np.exp(-r*T)*np.mean(pr2b_payoff_BB))
         print("Standard Deviation: ", np.std(pr2b_payoff_BB)/np.sqrt(nopaths))
```

```
Estimate:  4.42974373632
Standard Deviation:  0.10641238425
```

1.2.3 Part (c)

Basically redo part (a) and part (b) using $nopaths = 100,000$, but for a Knock-Out option instead.

In [42]:
```
# Update parameters
nopaths=10000
vol = 0.50
r = 0.10
KO = 45

# Construct stock dataframe
stock_df = pd.DataFrame(index=pd.RangeIndex(0,N+1), columns=pd.RangeIndex(1,nopaths+1)

# t0 - S_0 = 100
stock_df.loc[0,:] = S_0

# Iterate using Euler Scheme
for i in range(1,len(stock_df.index)):
    Z = np.random.standard_normal(size=nopaths)
    stock_df.loc[i,:] = stock_df.loc[i-1,:] + r*stock_df.loc[i-1,:]*dt + vol*stock_df

# Populate the 31st row with the prospective option payoffs
stock_df.loc[31,:] = np.exp(-r*T)*np.maximum(stock_df.loc[30,:]-S_0, 0)

# Populate the last row with the maximums of the appropriate columns
for k in range(1,nopaths+1):
    stock_df.at[32, k] = stock_df.loc[1:30, k].min() > KO

# Construct a dataframe for modeling in between discrete times
# Row i corresponds to max S_t \in [t_i, t_{i+1}]
bbridge_2c = pd.DataFrame(index=pd.RangeIndex(1,N+1), columns=pd.RangeIndex(1,nopaths

for i in range(1,len(bbridge_2c)+1):
    for j in range(1,nopaths+1):

        min_value = Min_bbDist( stock_df.loc[i-1,j], stock_df.loc[i,j], dt )
        bbridge_2c.loc[i,j] = stock_df.loc[i-1,j]*(1 + vol*min_value)

# Populate the last row with the maximums of the appropriate columns
for k in range(1,nopaths+1):
    bbridge_2c.at[31, k] = bbridge_2c[k].min() > KO

indicator_BB = np.array(bbridge_2c.loc[31,:])
indicator_noBB = np.array(stock_df.loc[32,:])
dpayout = np.array(stock_df.loc[31,:])
conditional_payoffs_2c_BB = np.multiply(indicator_BB, dpayout)
conditional_payoffs_2c_noBB = np.multiply(indicator_noBB, dpayout)

# Report All Results
```

```
print("==== With Strike = S_0 ====")
print("Estimate (w/ BB): ", np.mean(conditional_payoffs_2c_BB))
print("Standard Deviation: ", np.std(conditional_payoffs_2c_BB)/np.sqrt(nopaths))
print("Estimate (w/out BB): ", np.mean(conditional_payoffs_2c_noBB))
print("Standard Deviation: ", np.std(conditional_payoffs_2c_noBB)/np.sqrt(nopaths))
```

```
==== With Strike = S_0 ====
Estimate (w/ BB):  3.56132865636
Standard Deviation:  0.0825140059925
Estimate (w/out BB):  4.728545376202554
Standard Deviation:  0.0889746963844
```

1.3 Question 3: *Two-Asset Down-and-Out Call Option Pricing*

1.3.1 Part (a):

Consider a discrete pricing problem with N = 50 time steps within [0, T] and use ordinary Monte Carlo simulation to price the option. The paper indicates that the true price is 3.645 for the discrete time option. Do this part without the Brownian Bridge for the continuous time correction on $S_2(T)$.

In [22]:
```
# Define problem parameters
S_1_init = S_2_init = K = 100
r = 0.10
sig_1 = sig_2 = 0.30
rho = 0.5
T = 0.2
N = 50
dt = T/N
H = 95
```

In [23]:
```
# Initialise dataframes
S1_df = pd.DataFrame(index=pd.RangeIndex(0,N+1), columns=pd.RangeIndex(1,1001))
S1_df.loc[0,:] = S_1_init
S2_df = pd.DataFrame(index=pd.RangeIndex(0,N+1), columns=pd.RangeIndex(1,1001))
S2_df.loc[0,:] = S_2_init

# Multivariate Normal Distribution Parameters
mu     = [0,0]
cv_mat = [[1, rho],[rho, 1]]

# Populate Dataframe
for i in range(1,len(S1_df)):

    # Generate Correlated Standard Normals
    Z = npr.multivariate_normal(mu, cv_mat, size=1000)
    Z1 = np.array([z[0] for z in Z])
    Z2 = np.array([z[1] for z in Z])
```

```
        # Iterate Dataframe (closed form)
        S1_df.loc[i,:] = S1_df.loc[i-1,:]*np.exp( (r-0.5*sig_1**2)*dt + sig_1*np.sqrt(dt)
        S2_df.loc[i,:] = S2_df.loc[i-1,:]*np.exp( (r-0.5*sig_2**2)*dt + sig_2*np.sqrt(dt)

    # Put a boolean check to see if S_2(t) > H for all t.
    for col in S2_df.columns:
        S2_df.loc[51,col] = S2_df[col].min() > H

    # Put the *prospective*, discounted payoff in the 51st row of S1_df
    S1_df.loc[51,:] = np.exp(-r*T)*np.maximum(S1_df.loc[50,:] - K, 0)
```

```
In [24]: # Obtain values of interest
         condition = np.array(S2_df.loc[51,:])
         d_payoffs = np.array(S1_df.loc[51,:])
         conditional_payoffs = np.multiply(condition,d_payoffs)
         std_error = np.std(conditional_payoffs)/np.sqrt(10000)
```

```
In [25]: # Report Results
         print("---- Standard MCS ----")
         print("Estimated option price : %f" % np.mean(conditional_payoffs))
         print("Option price std. error: %f" % std_error)
```

```
---- Standard MCS ----
Estimated option price : 3.777255
Option price std. error: 0.079912
```

1.3.2 Part (b):

Use a Brownian Bridge as a continuous time correction for finding the minimum value that $S_2(t)$ takes.

```
In [26]: # Construct a dataframe for modeling in between discrete times
         # Row i corresponds to max S_t \in [t_i, t_{i+1}]
         bbridge_3b = pd.DataFrame(index=pd.RangeIndex(1,N+1), columns=pd.RangeIndex(1,1000+1)

         # Populate last row of Brownian Bridge Dataframe
         for i in range(1,len(bbridge_3b)+1):
             for j in range(1,1000+1):

                 max_value = Min_bbDist( S2_df.loc[i-1,j], S2_df.loc[i,j], dt )
                 bbridge_3b.loc[i,j] = S2_df.loc[i-1,j]*(1 + vol*max_value)

         # Populate the last row with the maximums of the appropriate columns
         for k in range(1,1000+1):
             bbridge_3b.at[51, k] = bbridge_3b[k].min() > H
```

```
In [27]: # Obtain values of interest
         condition_2 = np.array(bbridge_3b.loc[51,:])
```

```
conditional_payoffs_2 = np.multiply(condition_2,d_payoffs)
std_error_2 = np.std(conditional_payoffs_2)/np.sqrt(10000)

# Report Results
print("---- Standard MCS ----")
print("Estimated option price : %f" % np.mean(conditional_payoffs_2))
print("Option price std. error: %f" % std_error_2)
```

```
---- Standard MCS ----
Estimated option price : 2.181014
Option price std. error: 0.062825
```

1.4 Question 4: *Practice on Credit Derivatives and Copulas*

A whole lot of background information. . . Use a specified Guassian copula with covariance matrix Σ to generate five separate default times, each of which has an marginal distribution that is $\exp(\lambda)$. Use this to price each of the possible credit derivatives: FtD, 2tD, . . ., 5tD. Do this for each $\rho \in \{0.0, 0.20, 0.40, 0.60, 0.80, 1.0\}$.

```
In [28]: # Set up covariance matrix for a given rho
         def cov_mat( rho ):

             cov_mat = np.eye(5, dtype=float)
             for x in range(5):
                 for y in range(5):
                     if x!=y:
                         cov_mat[x][y] = rho

             return cov_mat
```

```
In [29]: # Loop through each rho
         rho_list = [i/5.0 for i in range(5)]
         for rho in rho_list:

             # Set up bucket dictionary
             buckets = {k: list() for k in range(1,6)}

             # Define needed parameters to get Copula
             cv = cov_mat(rho)
             A = np.linalg.cholesky(cv)
             lam = 0.01/(1-0.35)
             for i in range(100000):

                 # Copula algo
                 Z = npr.standard_normal(size=5)
                 Y = A.dot(Z)
                 U = norm.cdf(Y)
                 lam = 0.01/(1-0.35)
```

```
            X = (-1/lam)*np.log(1-U)

            # How many of the 5 bonds defaults over [0,5]
            test = X<5
            no_defaults = test.sum()

            for k in range(1,no_defaults+1):
                buckets[k].append( np.exp(-r*T)*(1-0.35) )
            for k in range(no_defaults+1, 6):
                buckets[k].append( 0 )


        # Now take averages
        df = pd.DataFrame.from_dict(buckets)

        print("---- rho = %f ----" % rho)
        for k in range(1,6):
            print("%ith to default price    : $%f" % (k, np.mean(df[k])))
            print("%ith to default std.error: $%f\n" % (k, np.std(df[k])/np.sqrt(100000))
```

```
---- rho = 0.000000 ----
1th to default price    : $0.203837
1th to default std.error: $0.000940

2th to default price    : $0.030697
2th to default std.error: $0.000431

3th to default price    : $0.002217
3th to default std.error: $0.000119

4th to default price    : $0.000070
4th to default std.error: $0.000021

5th to default price    : $0.000000
5th to default std.error: $0.000000

---- rho = 0.200000 ----
1th to default price    : $0.182072
1th to default std.error: $0.000910

2th to default price    : $0.043165
2th to default std.error: $0.000506

3th to default price    : $0.008831
3th to default std.error: $0.000236

4th to default price    : $0.001230
4th to default std.error: $0.000088
```

```
5th to default price    : $0.000076
5th to default std.error: $0.000022

---- rho = 0.400000 ----
1th to default price    : $0.159123
1th to default std.error: $0.000872

2th to default price    : $0.053213
2th to default std.error: $0.000557

3th to default price    : $0.018031
3th to default std.error: $0.000334

4th to default price    : $0.005008
4th to default std.error: $0.000178

5th to default price    : $0.000949
5th to default std.error: $0.000078

---- rho = 0.600000 ----
1th to default price    : $0.133160
1th to default std.error: $0.000819

2th to default price    : $0.058202
2th to default std.error: $0.000580

3th to default price    : $0.028066
3th to default std.error: $0.000413

4th to default price    : $0.012392
4th to default std.error: $0.000278

5th to default price    : $0.003810
5th to default std.error: $0.000155

---- rho = 0.800000 ----
1th to default price    : $0.103693
1th to default std.error: $0.000744

2th to default price    : $0.059215
2th to default std.error: $0.000585

3th to default price    : $0.037221
3th to default std.error: $0.000473

4th to default price    : $0.022988
4th to default std.error: $0.000376
```

```
5th to default price    : $0.011551
5th to default std.error: $0.000269
```

```
In [43]: ## Handle the rho = 1.0 case separately,
         ## Since the covariance matrix isn't PSD

         # Set up bucket dictionary
         buckets = {k: list() for k in range(1,6)}

         # Define needed parameters to get Copula
         lam = 0.01/(1-0.35)
         for i in range(100000):

             # Use PIT to obtain Exponentials
             U = npr.uniform(size=5)
             X = (-1/lam)*np.log(1-U)

             # How many of the 5 bonds defaults over [0,5]
             test = X<5
             no_defaults = test.sum()

             if no_defaults == 0:
                 for k in range(1,6):
                     buckets[k].append( 0 )
             else:
                 for k in range(1,6):
                     buckets[k].append( np.exp(-r*T)*(1-0.35) )



         # Now take averages
         df = pd.DataFrame.from_dict(buckets)

         print("---- rho = %f ----" % 1.0)
         for k in range(1,6):
             print("%ith to default price    : $%f" % (k, np.mean(df[k])))
             print("%ith to default std.error: $%f\n" % (k, np.std(df[k])/np.sqrt(100000)))
```

```
---- rho = 1.000000 ----
1th to default price    : $0.200899
1th to default std.error: $0.000933

2th to default price    : $0.200899
2th to default std.error: $0.000933
```

```
3th to default price    : $0.200899
3th to default std.error: $0.000933

4th to default price    : $0.200899
4th to default std.error: $0.000933

5th to default price    : $0.200899
5th to default std.error: $0.000933
```