Monte Carlo Project

Group Member: Jichen Wu, Maggie Wen Liu

**Objective**: Price Arithmetic Asian Call using standard Monte Carlo method, Control Variates Monte Carlo method and randomized Quasi Monte Carlo method. Then investigate the efficiency and compare the performances of three methods for fixed dimension increasing sample size, and fixed sample size for increasing dimension.

**Work Division:**

Maggie wen Liu wrote the code for Question 1 and Question 3. Jichen Wu searched for the sobol sequence generator online and wrote code for Question 2. Then we discussed together and run our code, and get relevant data. Maggie Wen Liu wrote the analysis for Control Variates and standard Monte Carlo Method. Jichen Wu wrote the analysis for Quasi Monte Carlo method. Then we combined our finding result and Maggie Wen Liu compiled the final paper.

**Structure of Paper**:

We separate our paper into three parts. In the first part, we price arithmetic Asian Call using Monte Carlo method and Control Variate method, In the second part, we price arithmetic Asian call using Quasi Monte Carlo method. In the third part, we investigate the impact of dimension m chosen for the control variate approach and randomized quasi-Monte Carlo method. In each Part, we briefly state the methods and its relevance to our codes, show the result tables and graphs, investigate the effectiveness and compare performance.

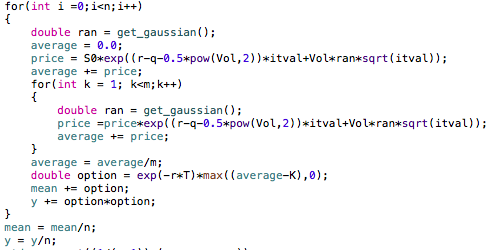
***Part I***

We develop Monte Carlo simulation C++ program to price arithmetic Asian call options in the Black-Scholes-Merton Model:

*Asian\_Call\_MC(K,T,S,,r,q,n,m)*

Where the parameters strike Price *K* =100, time to maturity *T* = 1 years, current asset price =100, risk free rate *r* = 10%, the volatility of the asset = 20%, continuous yield q=0, discrete monitoring m=50, and n is number of sample paths generated. So the length of each time interval is =1/m = 0.02, =, for i=1,2, …,m. We simulate 50 times price for each sample path.

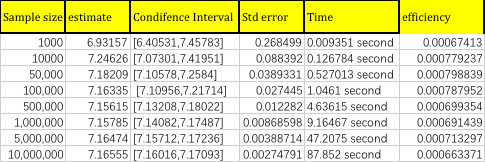
exp((r-q-)+), where ~N(0,1)



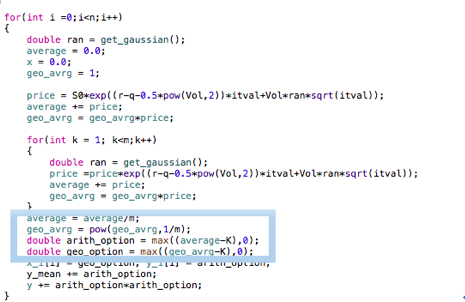
Then we use those 50 prices to calculate = and . the estimate of the arithmetic Asian option price is the the average of n payoff simulated.

We look at standard error, total computational time, and efficiency with an increasing sequence of sample sizes.

***Result***:



*control variate method*: we use Geometric Asian Call as the control variables. We simulate n times the arithmetic Asian option payoffs = and n times geometric Asian option payoff for each sample path generated = for i= 1,2,…,n, and m=1,2,…50.



Then calculate *E(X)*, the expected geometric Asian call, from black Scholes:

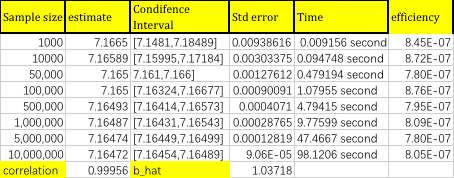


where r is interest rate, =is volatility, is continuous yield,

is maturity, strike K, initial asset price . Then the estimate price of the arithmetic Asian Call is:

, where b=

***Result***:



It is shown from the results that for the same sample size the standard error is smaller for

control variates than without control. The efficiency for control variate method is much smaller. It means that control variates method is much efficient than without control.

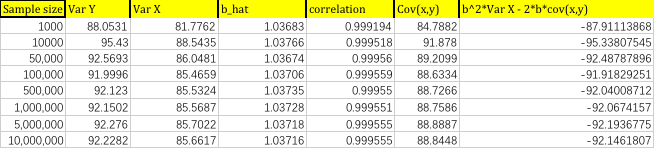
**Investigation of Effectiveness:**

The efficiency for control variate is much smaller than without control which means that control variates works better.

Zoom in this part

It is shown from the plots that the standard error for the control variates methods is much smaller than without control. The standard error for without control method is Var() = . The standard error for control variates is Var() = +). So if <0, control variate method is efficient than without control. The higher the correlation, the better the performance. So we check whether is significant smaller than 0

Here is the table for variance, covariance, correlation for control variates:



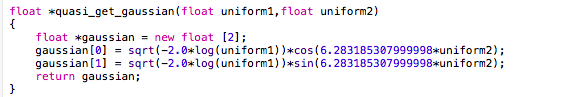
It is shown that the correlations between x and y are really consistently high. So is very small which reduces standard error significantly. Control Variates method is more efficient than without control.

***Part II***

We use quasi Monte Carlo Method to price arithmetic Asian call via Sobol Sequence generator which was written by S.Joe and F.Y.Kuo, <http://web.maths.unsw.edu.au/~fkuo/sobol/>

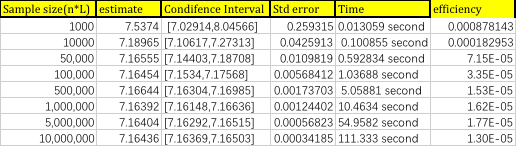
*Asian\_Call\_MC(K,T,S,,r,q,n,L,m)*. #We set the batch L as 10.

Differing from Monte Carlo, for every time we use Quasi Monte Carlo we make a random shift. We generate a 50 uniform numbers , …,. For each kth entry of each of make a shift mod 1. In order to generate Gaussian variates, we utilize Box Muller Algorithm by using two sobol numbers to get two Gaussian variates. X = , y= , x and y are independent normally distribute variables with mean 0 and variance 1.



Then we used those Gaussian variables to simulate the price changes as before and obtain estimates. Confidence interval and standard error are calculated by those L=10 estimates

***Result and effectiveness***

******

we graph the standard error:

Zoom in this part

Differing from the Monte Carlo method, Quasi Monte Carlo are predetermined.

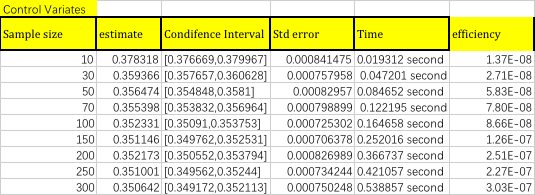
We Make a random shift for each batch for sobol sequence. It is shown in the graphs that the standard errors of Quasi Monte Carlo Method are bounded by Standard Monte Carlo approach and control variates approach. When the n\*L is small, the standard error of QMC is close to MC and far from Control Variates. But when n\*L gets larger and larger, the standard error of QMC is close to Control Variates and far from MC.

It is shown from that when sample size is small, the the efficiency for Quasi Monte Carlo is similar to the standard approach and higher than control variates. When sample size gets larger, the efficiency for Quasi Monte Carlo gets closer and closer to control variates and far from standard approach, but does not actually reach the level of control variates. So the Quasi Monte Carlo method works better than standard Monte Carlo and worse than control variates method.

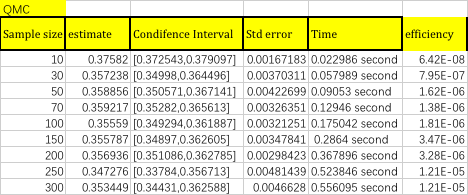
***Part III***

Now we fix sample size n = 10,000, and use different number of dimension m to run of our code before. We change the parameters of Asian Call, time to maturity T= 2, risk free interest rate r = 0.05, the volatility of the asset = 50%, , Strike price K = 2, initial Stock price S = 2, continuous yield q=0. We also set the batch L =10 for quasi Monte Carlo method

***Result***



For the increasing dimensions, the standard error is quite consistent for control variates.



But for Quasi Monte Carlo Method, the standard error oscillates when sample size increases.

When dimensional is small, the efficiency of quasi Monte Carlo method and control variate method is close. But when dimensional increases, the efficiency of quasi Monte Carlo gradually deviate from control variates. Therefore, for large dimension, Quasi Monte Carlo method is not good as Control Variates method.

***Conclusion:***

When fixed dimension m, for an increasing sample size Control Variates Monte Carlo method is consistent better than Standard Monte Carlo approach. When sample size(n\*L) is small, randomized quasi-Monte Carlo Method’s efficiency is close to the standard Monte Carlo approach. When sample size increases, randomized quasi-Monte Carlo method’s efficiency is getting closer to the Variate Control Monte Carlo method and far from standard Monte Carlo approach. Therefore, Quasi Monte Carlo Method works better for large sample size and worse for small sample size. Control Variates approach is the best and standard Monte Carlo approach is the worst.

When fixed sample size and small dimension, the efficiency of randomized quasi-Monte Carlo method’s efficiency is close to the Control Variate Monte Carlo approach. But when dimension increases, the efficiency of randomized quasi-Monte Carlo method is get far from the Control Variate Monte Carlo approach, which means that for large dimension, Control Variate Monte Carlo approach works better than randomized quasi-Monte Carlo method. Randomized quasi-Monte Carlo method gets worse when dimension increases.

***Appendix: C++ files***

***Header files***

#ifndef sobol\_h

#define sobol\_h

// Frances Y. Kuo

//

// Email: <f.kuo@unsw.edu.au>

// School of Mathematics and Statistics

// University of New South Wales

// Sydney NSW 2052, Australia

//

// Last updated: 21 October 2008

//

// You may incorporate this source code into your own program

// provided that you

// 1) acknowledge the copyright owner in your program and publication

// 2) notify the copyright owner by email

// 3) offer feedback regarding your experience with different direction numbers

//

//

// -----------------------------------------------------------------------------

// Licence pertaining to sobol.cc and the accompanying sets of direction numbers

// -----------------------------------------------------------------------------

// Copyright (c) 2008, Frances Y. Kuo and Stephen Joe

// All rights reserved.

//

// Redistribution and use in source and binary forms, with or without

// modification, are permitted provided that the following conditions are met:

//

// \* Redistributions of source code must retain the above copyright

// notice, this list of conditions and the following disclaimer.

//

// \* Redistributions in binary form must reproduce the above copyright

// notice, this list of conditions and the following disclaimer in the

// documentation and/or other materials provided with the distribution.

//

// \* Neither the names of the copyright holders nor the names of the

// University of New South Wales and the University of Waikato

// and its contributors may be used to endorse or promote products derived

// from this software without specific prior written permission.

//

// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS ``AS IS'' AND ANY

// EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED

// WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE

// DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDERS BE LIABLE FOR ANY

// DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES

// (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;

// LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND

// ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT

// (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS

// SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

// -----------------------------------------------------------------------------

#include <cstdlib> // \*\*\* Thanks to Leonhard Gruenschloss and Mike Giles \*\*\*

#include <cmath> // \*\*\* for pointing out the change in new g++ compilers \*\*\*

#include <iostream>

#include <iomanip>

#include <fstream>

using namespace std;

// ----- SOBOL POINTS GENERATOR BASED ON GRAYCODE ORDER -----------------

// INPUT:

// N number of points (cannot be greater than 2^32)

// D dimension (make sure that the data file contains enough data!!)

// dir\_file the input file containing direction numbers

//

// OUTPUT:

// A 2-dimensional array POINTS, where

//

// POINTS[i][j] = the jth component of the ith point,

//

// with i indexed from 0 to N-1 and j indexed from 0 to D-1

//

// ----------------------------------------------------------------------

double \*\*sobol\_points(unsigned N, unsigned D, char \*dir\_file)

{

ifstream infile(dir\_file,ios::in);

if (!infile) {

cout << "Input file containing direction numbers cannot be found!\n";

exit(1);

}

char buffer[1000];

infile.getline(buffer,1000,'\n');

// L = max number of bits needed

unsigned L = (unsigned)ceil(log((double)N)/log(2.0));

// C[i] = index from the right of the first zero bit of i

unsigned \*C = new unsigned [N];

C[0] = 1;

for (unsigned i=1;i<=N-1;i++) {

C[i] = 1;

unsigned value = i;

while (value & 1) {

value >>= 1;

C[i]++;

}

}

// POINTS[i][j] = the jth component of the ith point

// with i indexed from 0 to N-1 and j indexed from 0 to D-1

double \*\*POINTS = new double \* [N];

for (unsigned i=0;i<=N-1;i++) POINTS[i] = new double [D];

for (unsigned j=0;j<=D-1;j++) POINTS[0][j] = 0;

// ----- Compute the first dimension -----

// Compute direction numbers V[1] to V[L], scaled by pow(2,32)

unsigned \*V = new unsigned [L+1];

for (unsigned i=1;i<=L;i++) V[i] = 1 << (32-i); // all m's = 1

// Evalulate X[0] to X[N-1], scaled by pow(2,32)

unsigned \*X = new unsigned [N];

X[0] = 0;

for (unsigned i=1;i<=N-1;i++) {

X[i] = X[i-1] ^ V[C[i-1]];

POINTS[i][0] = (double)X[i]/pow(2.0,32); // \*\*\* the actual points

// ^ 0 for first dimension

}

// Clean up

delete [] V;

delete [] X;

// ----- Compute the remaining dimensions -----

for (unsigned j=1;j<=D-1;j++) {

// Read in parameters from file

unsigned d, s;

unsigned a;

infile >> d >> s >> a;

unsigned \*m = new unsigned [s+1];

for (unsigned i=1;i<=s;i++) infile >> m[i];

// Compute direction numbers V[1] to V[L], scaled by pow(2,32)

unsigned \*V = new unsigned [L+1];

if (L <= s) {

for (unsigned i=1;i<=L;i++) V[i] = m[i] << (32-i);

}

else {

for (unsigned i=1;i<=s;i++) V[i] = m[i] << (32-i);

for (unsigned i=s+1;i<=L;i++) {

V[i] = V[i-s] ^ (V[i-s] >> s);

for (unsigned k=1;k<=s-1;k++)

V[i] ^= (((a >> (s-1-k)) & 1) \* V[i-k]);

}

}

// Evalulate X[0] to X[N-1], scaled by pow(2,32)

unsigned \*X = new unsigned [N];

X[0] = 0;

for (unsigned i=1;i<=N-1;i++) {

X[i] = X[i-1] ^ V[C[i-1]];

POINTS[i][j] = (double)X[i]/pow(2.0,32); // \*\*\* the actual points

// ^ j for dimension (j+1)

}

// Clean up

delete [] m;

delete [] V;

delete [] X;

}

delete [] C;

return POINTS;

}

#endif /\* sobol\_h \*/

***Main.Cpp***

#include <iostream>

#include <iomanip>

#include <stdlib.h>

#include <cmath>

#include <vector>

#include <time.h>

#include <fstream>

#include "sobol.h"

#define max(a, b) (((a) > (b)) ? (a) : (b))

#define min(a, b) (((a) < (b)) ? (a) : (b))

using namespace std;

double mean, price, y,std\_err,average,x,x\_mean,geo\_avrg;

double \*x\_i,\*y\_i;

double CDF(double x);

double CDF(double x)

{

if (x > 15.0) { return 1.0; };

if (x < -15.0) { return 0.0; };

double v[16]={0,1.253314137315500,0.6556795424187985,0.4213692292880545,0.3045902987101033,0.2366523829135607,0.1928081047153158,0.1623776608968675, 0.1401041834530502,0.1231319632579329, 0.1097872825783083,0.09902859647173193, 0.09017567550106468,0.08276628650136917,0.0764757610162485,0.07106958053885211};

double c = 0.918938533204672;

int j = floor(min(abs(x)+0.5,14));

int z = j;

double h = abs(x)-z;

double a = v[j+1];

double b = z\*a-1;

double q =1;

double s = a+ h \*b;

for (int i = 2;i<=24-j;i = i+2)

{

a = (a+z\*b)/i;

b = (b+z\*a)/(i+1);

q = q\*h\*h;

s = s+q\*(a+h\*b);

}

double y = s\*exp(-0.5\*x\*x-c);

if(x>0)

y = 1-y;

return y;

}

float get\_uniform()

{

return (((float) random())/(pow(2.0, 31.0)-1.0));

}

float get\_gaussian()

{

return (sqrt(-2.0\*log(get\_uniform()))\*cos(6.283185307999998\*get\_uniform()));

}

float \*quasi\_get\_gaussian(float uniform1,float uniform2)

{

float \*gaussian = new float [2];

gaussian[0] = sqrt(-2.0\*log(uniform1))\*cos(6.283185307999998\*uniform2);

gaussian[1] = sqrt(-2.0\*log(uniform1))\*sin(6.283185307999998\*uniform2);

return gaussian;

}

float \*box\_muller()

{

float \*gaussian = new float [2];

gaussian[0] = sqrt(-2.0\*log(get\_uniform()))\*cos(6.283185307999998\*get\_uniform());

gaussian[1] = sqrt(-2.0\*log(get\_uniform()))\*sin(6.283185307999998\*get\_uniform());

return gaussian;

}

double geometric\_black\_scholes\_call(const double& S, const double& K, const double& r, const double& sigma, const double& time, const double& q)// this function is used to compute option price by Black Scholes fomula

{

double time\_sqrt = sqrt(time);

double d1 = (log(S/K)+(r-q)\*time)/(sigma\*time\_sqrt)+0.5\*sigma\*time\_sqrt;

double d2 = d1-(sigma\*time\_sqrt);

return S\*exp(-q\*time)\*CDF(d1) - K\*exp(-r\*time)\*CDF(d2);

}

void simulation(double K, double T, double S0, double Vol, double r, double q, double n,double m)

{

// initialize vector to calculate the change of price

cout << "simulation without control"<<endl;

clock\_t start\_time=clock();

mean = 0.0;

y = 0.0;

double itval = T/m;

for(int i =0;i<n;i++)

{

double ran = get\_gaussian();

average = 0.0;

price = S0\*exp((r-q-0.5\*pow(Vol,2))\*itval+Vol\*ran\*sqrt(itval));

average += price;

for(int k = 1; k<m;k++)

{

double ran = get\_gaussian();

price =price\*exp((r-q-0.5\*pow(Vol,2))\*itval+Vol\*ran\*sqrt(itval));

average += price;

}

average = average/m;

double option = exp(-r\*T)\*max((average-K),0);

mean += option;

y += option\*option;

}

mean = mean/n;

y = y/n;

std\_err=sqrt((1/(n-1))\*(y-mean\*mean));

clock\_t end\_time=clock();

double time = static\_cast<double>(end\_time-start\_time)/CLOCKS\_PER\_SEC;

cout << "estimate = "<< mean << endl<<"standard error = "<<std\_err<<endl;

double upper\_bound = mean+1.96\*std\_err;

double lower\_bound = mean-1.96\*std\_err;

cout <<"95% confidence Interval = "<< "["<<lower\_bound<<","<<upper\_bound<<"]"<<endl;

cout<< "total computation time is: "<<time<<" second"<<endl;

cout<< "efficiency is: "<< std\_err\*std\_err\*time<<endl;

}

void control(double K, double T, double S0, double Vol, double r, double q, double n,double m)

{

cout << "-----------------------------------------"<<endl;

cout << "simulation with control"<<endl;

x\_i = new double[(int)n];

y\_i = new double[(int)n];

for(int i =0; i<n;i++)

{

x\_i[i]=0;

y\_i[i]=0;

}

clock\_t start\_time=clock();

double y\_mean = 0.0;

y = 0.0;

double itval = T/m;

for(int i =0;i<n;i++)

{

double ran = get\_gaussian();

average = 0.0;

x = 0.0;

geo\_avrg = 1;

price = S0\*exp((r-q-0.5\*pow(Vol,2))\*itval+Vol\*ran\*sqrt(itval));

average += price;

geo\_avrg = geo\_avrg\*price;

for(int k = 1; k<m;k++)

{

double ran = get\_gaussian();

price =price\*exp((r-q-0.5\*pow(Vol,2))\*itval+Vol\*ran\*sqrt(itval));

average += price;

geo\_avrg = geo\_avrg\*price;

}

average = average/m;

geo\_avrg = pow(geo\_avrg,1/m);

double arith\_option = max((average-K),0);

double geo\_option = max((geo\_avrg-K),0);

x\_i[i] = geo\_option; y\_i[i] = arith\_option;

y\_mean += arith\_option;

y += arith\_option\*arith\_option;

}

y\_mean = y\_mean/n;

double T\_til =(0.5\*(T+itval));

double sigma\_til = (2\*m+1)\*Vol\*Vol/(3\*m);

double q\_til = q+0.5\*Vol\*Vol-0.5\*sigma\_til;

x\_mean =exp(r\*T\_til)\*geometric\_black\_scholes\_call(S0,K,r,sqrt(sigma\_til),T\_til,q\_til);

double above = 0.0;

double below = 0.0;

double var\_x = 0.0;

double var\_y = 0.0;

for(int i =0 ; i<n ; i++)

{

var\_x += (x\_i[i]-x\_mean)\*(x\_i[i]-x\_mean);

var\_y += (y\_i[i]-y\_mean)\*(y\_i[i]-y\_mean);

above += (x\_i[i]-x\_mean)\*(y\_i[i]-y\_mean);

below += (x\_i[i]-x\_mean)\*(x\_i[i]-x\_mean);

}

var\_y = var\_y/(n-1);

var\_x = var\_x/(n-1);

double covar = above /(n-1);

double pxy = covar/(sqrt(var\_x)\*sqrt(var\_y));

double b\_hat = above/below;

cout <<"variance y "<<var\_y<<endl;

cout <<"variance x "<<var\_x<<endl;

cout <<"b\_hat "<<b\_hat<<endl;

cout << "correlation "<<pxy<<endl;

cout <<"covariance "<<covar<<endl;

double yb = 0.0;

for(int i =0;i<n;i++)

{

yb += exp(-r\*T)\*(y\_i[i]+b\_hat\*(x\_mean-x\_i[i]));

}

yb = yb/n;

std\_err=sqrt((1-pxy\*pxy)\*(var\_y/n));

clock\_t end\_time=clock();

cout << "estimate = "<< yb << endl<<"standard error = "<<std\_err<<endl;

double upper\_bound = yb+1.96\*std\_err;

double lower\_bound = yb-1.96\*std\_err;

cout <<"95% confidence Interval = "<< "["<<lower\_bound<<","<<upper\_bound<<"]"<<endl;

double time = static\_cast<double>(end\_time-start\_time)/CLOCKS\_PER\_SEC;

cout<< "total computation time is: "<<time<<" second"<<endl;

cout<< "efficiency is: "<< std\_err\*std\_err\*time<<endl;

}

void quasi\_monte(double K, double T, double S0, double Vol, double r, double q, double n, double L,double m)

{

cout << "-----------------------------------------"<<endl;

cout << "quasi monte carlo simulation"<<endl;

clock\_t start\_time=clock();

int N = n;//atoi(argv[1]);

int D = m;//atoi(argv[2]);

char \*file = "sobol.txt";

double quasi\_mean = 0.0;

double quasi\_sqaure = 0.0;

double ab[10];

for(int t=0;t<L;t++)

{

double \*\*P = sobol\_points(N,D,file);

double shift[(int)m];

mean = 0.0;

y = 0.0;

for(int i = 0; i<m;i++)

{

shift[i] = get\_uniform();

}

// use quasi monte carlo simulate gaussian variables

for( int i = 0; i<n; i++)

{

for(int k=0;k<m;k++)

{

P[i][k]=P[i][k]+shift[k];

if(P[i][k]>=1) P[i][k] = P[i][k]-1;

if(k%2==1)

{

float \*temp\_gaussian = quasi\_get\_gaussian(P[i][k-1],P[i][k]);

//double a,b;

//a =sqrt(-2.0\*log(P[k-1][i]))\*cos(6.283185307999998\*P[k][i]);

//b =sqrt(-2.0\*log(P[k-1][i]))\*sin(6.283185307999998\*P[k][i]);

//P[k-1][i] = a;P[k][i]=b;

P[i][k-1] = temp\_gaussian[0];

P[i][k] = temp\_gaussian[1];

}

}

}

double itval = T/m;

for(int i =0;i<n;i++)

{

average = 0.0;

price = S0\*exp((r-q-0.5\*pow(Vol,2))\*itval+Vol\*P[i][0]\*sqrt(itval));

average += price;

for(int k = 1; k<m;k++)

{

price =price\*exp((r-q-0.5\*pow(Vol,2))\*itval+Vol\*P[i][k] \*sqrt(itval));

average += price;

}

average = average/m;

double option = exp(-r\*T)\*max((average-K),0);

mean += option;

}

mean = mean/n;

quasi\_mean += mean;

quasi\_sqaure += mean\*mean;

ab[t]=mean;

}

quasi\_mean = quasi\_mean/L;

quasi\_sqaure = quasi\_sqaure/L;

double a=0.0;

for (int i=0;i<L;i++)

{

a+=pow((ab[i]-quasi\_mean),2);

}

std\_err = sqrt(1/L\*a/(L-1));

clock\_t end\_time=clock();

cout << "estimate = "<< quasi\_mean << endl<<"standard error = "<<std\_err<<endl;

double upper\_bound = quasi\_mean+1.96\*std\_err;

double lower\_bound = quasi\_mean-1.96\*std\_err;

cout <<"95% confidence Interval = "<< "["<<lower\_bound<<","<<upper\_bound<<"]"<<endl;

double time = static\_cast<double>(end\_time-start\_time)/CLOCKS\_PER\_SEC;

cout<< "total computation time is: "<<time<<" second"<<endl;

cout<< "efficiency is: "<< std\_err\*std\_err\*time<<endl;

}

int main(int argc, const char \* argv[]) {

double r = 0.1;

double K = 100;

double S0 = 100;

double T = 1;

double m=10;

double Vol = 0.2;

double q = 0;

double no\_of\_trials = 10000;

double L = 10;

T = 2;

r = 0.05;

Vol = 0.5;

S0 = K =2;

q =0;

cout << "The sample size is "<<no\_of\_trials<<endl;

cout << "-----------------------------------------"<<endl;

simulation(K,T,S0,Vol,r,q,no\_of\_trials,m); //Q1

control(K,T,S0,Vol,r,q,no\_of\_trials,m); // Q1

quasi\_monte(K,T,S0,Vol,r,q,no\_of\_trials/L,L,m); // Q2

return 0;

}