IE 525 Hw 2

Maggie Wen Liu (NetID: mliu79)

2. standard method

Sample size	price	Std.error	95% Confidence interval	Computation time	efficiency
10000	30.826	0.468067	[29.9086, 31.7434]	0m0.012s	0.002629
100000	29.8631	0.144129	[29.5806, 30.1456]	0m0.089s	0.0018488
1000000	29.9495	0.0455774	[29.8602, 30.0389]	0m0.837s	0.0017387
10000000	29.9472	0.0144117	[29.9189, 29.9754]	0m8.600s	0.001786
30000000	29.9471	0.00831957	[29.9308, 29.9634]	0m24.995s	0.001730
50000000	29.95	0.00644496	[29.9374, 29.9627]	0m40.613s	0.001687
70000000	29.9535	0.00544782	[29.9428, 29.9642]	0m57.525s	0.0017073
75000000	29.952	0.00526288	[29.9416, 29.9623]	1m3.134s	0.001749
79800000	29.9529	0.0051023	[29.9429, 29.9629]	1m2.850s	0.0016362

Antithetic method

Sample size	price	Std.error	95% Confidence interval	Computation time	efficiency
10000	30.4099	0.244781	[29.1051, 30.8823]	0.011902 second	0.000713139
100000	29.9813	0.076663	[29.831, 30.1316]	0.103764 second	0.000609844
1000000	29.9684	0.0242881	[29.9208, 30.016]	0.909926 second	0.000536777
10000000	29.9537	0.00768135	[29.9387, 29.9688]	8.88981 second	0.000524526
20000000	29.9523	0.0054311	[29.9417, 29.963]	17.5372 second	0.000517292
22000000	29.9535	0.00517861	[29.9433, 29.9636]	19.8676 second	0.00053281
22200000	29.9542	0.00515539	[29.9441, 29.9643]	20.01 second	0.000531827
22600000	29.9543	0.00510946	[29.9443, 29.9643]	20.453 second	0.000533958

3.

For the same trials, the efficiency of antithetic method is smaller than the standard method

Instead of using generated Gaussian variables once and obtaining payoff one time, antithetic method use generated Gaussian variables twice and obtained two payoffs. So, for the same trials, antithetic methods generate more payoffs and can simulate option price better. Also, for the same accuracy for prediction, antithetic methods have fewer trials than standard method.

```
#include <iostream>
#include <cmath>
#include "normdist.h"
#include <vector>
#include <time.h>
#include <algorithm>
using namespace std;


double r_free_rate, strike_price;
double s_zero, time_to_expiration, volatility;
double no_of_trials;
double dividend_rate;
double call_option_monte;
vector<double>mean;
vector<double>mean_square;
vector<double>option_price;

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

float get_uniform()
{
   srand((int)time(0));
   return (((float) random())/(pow(2.0, 31.0)-1.0));
}

float get_gaussian()
{
   return
(sqrt(-2.0*log(get_uniform()))*cos(6.283185307999998*get_unifo
rm()));
```

```
}


int main(int argc, char* argv[])
{
    clock_t start_time=clock();
    //sscanf(argv[1],"%lf", &no_of_trials);
    time_to_expiration=0.01923;
    r_free_rate=0.003866;
    s_zero=1868.99;
    volatility=0.2979;
    strike_price=1870;
    dividend_rate=0.0232;
    call_option_monte = 0.0;
    cout << "Expiration Time (Years) = " << time_to_expiration<<
endl;
    cout << "Risk Free Interest Rate = " << r_free_rate << endl;
    cout << "Volatility (%age of stock value) = " << volatility*100
<< endl;
    cout << "Dividend Rate = "<<dividend_rate<<endl;
    cout << "Initial Stock Price = " << s_zero << endl;
    cout << "Strike Price = " << strike_price << endl;

    no_of_trials=22600000;
    cout << "sample size = "<< no_of_trials<<endl;
    mean.push_back(0);
    mean_square.push_back(0);
    option_price.push_back(0);
    for(int i =0;i<no_of_trials;i++)
    {
        float R = get_gaussian();
        double
price_up=s_zero*exp((r_free_rate-dividend_rate-0.5*pow(volatil
```

```
ity,2))*time_to_expiration+volatility*R*sqrt(time_to_expiratio
n));
        double
price_down=s_zero*exp((r_free_rate-dividend_rate-0.5*pow(volat
ility,2))*time_to_expiration-volatility*R*sqrt(time_to_expirat
ion));
        double temp_option_price=
0.5*exp(-r_free_rate*time_to_expiration)*(max(price_up-strike_
price,0)+max(price_down-strike_price,0));
        call_option_monte += temp_option_price;
        mean.push_back((mean[i]*i+temp_option_price)/(i+1));

mean_square.push_back((mean_square[i]*i+pow(temp_option_price,
2))/(i+1));
    }
    double st_error =
sqrt((1/(no_of_trials-1))*(mean_square[no_of_trials-1]-pow(mea
n[no_of_trials-1],2)));
    cout <<"Monte Carlo European Option Pricing ="
<<call_option_monte/no_of_trials<<endl;
    cout <<"Monte Carlo Standard Error ="<<st_error<<endl;
    double upper_bound =
call_option_monte/no_of_trials+1.96*st_error;
    double lower_bound =
call_option_monte/no_of_trials-1.96*st_error;
    cout <<"95% confidence Interval = "<<
"["<<lower_bound<<","<<upper_bound<<"]"<<endl;
    clock_t end_time=clock();
    double time =
static_cast<double>(end_time-start_time)/CLOCKS_PER_SEC;
    cout<< "Running time is: "<<time<<" second"<<endl;
    cout<< "efficiency is: "<<st_error*st_error*time<<endl;
}
```