Introduction to Option Pricing Theory with focus on C++ implementation

Emiliano Papa

Deutsche Bank, London

Date of last change: October 9th 2023

Date of last compilation: May 17, 2024

Reading List

- John Hull, Options, Futures and Other Derivatives, Prentice Hall 2006.
- Steve Shreve, Stochastic Calculus for Finance II, Springer 2004.
- Martin Baxter and Andrew Rennie, Financial Calculus: An Introduction to Derivative Pricing, (CUP 1996).
- Mark Joshi, C++ Design Patterns and Derivative Pricing, (Cambridge University Press, 2004).
- Leif B.G. Andersen, Vladimir Piterbarg, Interest Rate Modeling, Vols. I, II, III, Atlantic Financial Press 2010. (Advanced mathematically.)
- A gentle mathematical intro book on finance: An Introduction to the Mathematics of Financial Derivatives by Ali Hirsa, Salih N. Neftci

Main Points of Lecture

- 1.... Application to CRR model, or Binomial model for pricing Options.
- 2.... Get Started in C++.
- 3.... Entering data, Functions - help build pricer.
- 4.... CRR pricer but procedural approach
- 5.... Pointers
- 6.... Function Pointers

The Binomial Model

It is the simplest and most accessible approach to pricing options. We can see in a simple way the ideas of delta-hedging, risk elimination, and risk-neutral valuation that are important on option pricing.
Another positive point is that the model is numerically fast and can be extended to pricing options with early exercise for which simple closed-form analytic solutions do not exist.
However, keep in mind that the model has also some drawbacks. For instance the model assumes that the stock price can change from today to tomorrow to two possible values, up or down. This is obviously a severe simplification to the model. Add one more state and the results become intractable.
The model is good on its purpose to introduce an easy numerical access to option pricing. We will discuss then Monte-Carlo and PDE's at some future point.

Binomial Model

To illustrate the model take the example of a **call option with one period to maturity**. Assume today's stock price $S_0 = 100$. We also assume that the stock price tomorrow can be in one of the two states $S_1 = 105$ or $S_1 = 95$, with probabilities 0.6, and 0.4, respectively.

$$S_0 = 100 \quad , \quad S_1 = 105(p = 0.6) \quad , \quad S_1 = 95(p' = 0.4) \quad . \tag{1}$$

On this asset an option has been written with strike $K = 100$. A payoff of GBP 5,- is received on the first case, and the option expires out of the money in the second stock state.
What would the option price be?
People could follow a simple argument

$$c = 0.6 \times 5 + 0.4 \times 0 = 3 \quad . \tag{2}$$

However, this is not what the traders think! How is this possible?
Well, they observe something simple! When the stock price increases, also the option payoff increases. When the stock decreases the option payoff decreases or is zero. They have concluded that these assets have similar type of uncertainty, similar risk.

Binomial Model

Then the question is the following, can they build a portfolio which cancels these risks. They have figured this out already from practice, that indeed if they buy one option and hold the right amount of a short position on a stock, then that quantity has no uncertainty the next day. This means that they managed to build a portfolio where the risk is canceled. Let's check this combination

$$\text{Portfolio}_{\text{Day1}}(\text{up}) = \text{OptionPayoff}_1(up) - \frac{1}{2}S_1(up) \quad , \tag{3}$$

$$\text{Portfolio}_{\text{Day1}}(\text{down}) = \text{OptionPayoff}_1(down) - \frac{1}{2}S_1(down) \tag{4}$$

$$\text{Portfolio}_{\text{Day1}}(\text{up}) = 5 - \frac{1}{2}105 = \frac{10-105}{2} \quad , \tag{5}$$

$$\text{Portfolio}_{\text{Day1}}(\text{down}) = 0 - \frac{1}{2}95 \tag{6}$$

Therefore this basket does not change the price no matter what state the stock price ends up being.

Binomial Model

Since we assume zero rates, thus no discounting, the price of this portfolio is the same also the previous day. This means that

$$c - \frac{1}{2}100 = -\frac{1}{2}95 \quad , \quad c = 2.5 \quad . \tag{7}$$

Traders will not pay more than this price for the option, no matter what you think about the probability for the stock price to rise tomorrow. The real world measure probabilities seem to not matter in pricing options. This argument, however, relies on the fact that there is a liquid markets for stocks, where the traders can buy and sell freely the stocks, long or short. Think of an option on rain for instance, can the traders do the same trick? Not in this case, thus the risk-neutral measure does not apply there.

Binomial Model

Removing risk by building a portfolio of an option and a Δ amount of stock is called Delta-hedging. In our case

$$5 - \Delta \times 105 = 0 - \Delta \times 95 \tag{8}$$

resulting in

$$\Delta(105 - 95) = 5 \tag{9}$$

$$\Delta = \frac{5 - 0}{105 - 95} = \frac{1}{2} \tag{10}$$

This can be generalized

$$\Delta = \frac{\text{Range of Option payoff}}{\text{Range of Stock prices}} \sim \frac{\partial c}{\partial S} \tag{11}$$

Binomial Model

What about the probabilities? In risk-neutral measure, investors will ask for interest-free return, $S_1(\text{up}) = (1 + \text{U})\text{S}$ and $S_1(\text{down}) = (1 + D)S$

$$q(1 + U)S + (1 - q)(1 + D)S = S(1 + R) \tag{12}$$

$$q = \frac{(1 + R)S - S_d}{S_u - S_d} = \frac{R - D}{U - D} \tag{13}$$

In our case, $U = 0.05$, $D = -0.05$, $R = 0$, resulting in $q = 0.05/0.1$

$$q = \frac{1}{2} \quad . \tag{14}$$

From previous page

$$c_u - \Delta S_u = c_d - \Delta S_d \quad , \quad \Delta = \frac{c_u - c_d}{S_u - S_d} \tag{15}$$

Substitute $\Delta S = (c_u - c_d)/(u - d)$ in the portolfio exectation

$$c_u - \Delta S_u = (1 + R)(c - \Delta S) \tag{16}$$

Binomial Model

where $\Delta S_u = \Delta S \cdot u$ in the above, we get

$$c = \frac{1}{1+R}\left(\frac{R-D}{U-D}c_u + \frac{U-R}{U-D}c_d\right) \quad . \tag{17}$$

Therefore this risk-neutral probability can be used also to price the option

$$c = \frac{1}{1+R}\Big(qc_u + (1-q)c_d\Big) \quad , \tag{18}$$

$$c = q \times 5 + (1-q) \times 0 = 2.5 \quad , \quad (R=0) \quad , \tag{19}$$

same as previously, where we had $R = 0$.
It looks like real-world probabilities do no matter. The options can be priced with the risk-neutral probabilities. The reason being that stocks and cash can be used to replicate the option, without interference from the real probabilities.

Binomial Model

Use for instance $\Delta \times S$ stocks and $-95/2$ cash today. This gives now portfolio of GBP2.5. Tomorrow this will give $\Delta \times 105 - 95/2 = 5$ in the up scenario or $\Delta \times 95 - 95/2 = 0$ in the down scenario, same as the option's payoff.
This is now the basis for pricing in the binomial tree. The stepping back from the final payoff will be carried out with the risk-neutral probability, simply because this probability gives the correct results.
The Option is partly a stock, and more so when the stock price if further from ATM (from the strike price). Delta tends to increase as the stock price moves higher than the strike. The option is replicated with Δ amount of stock minus some cash. The cash is equal to the option price minus Delta Stock price. In our case $c - \Delta \times S = 2.5 - 1/2 \times 100$.
The option delivers the excess of the stock price over the strike. Hence the option price is expected to be lower than the stock price itself, otherwise one wouldn't go to the trouble of buying options, they would just settle for stocks directly.

Binomial Model

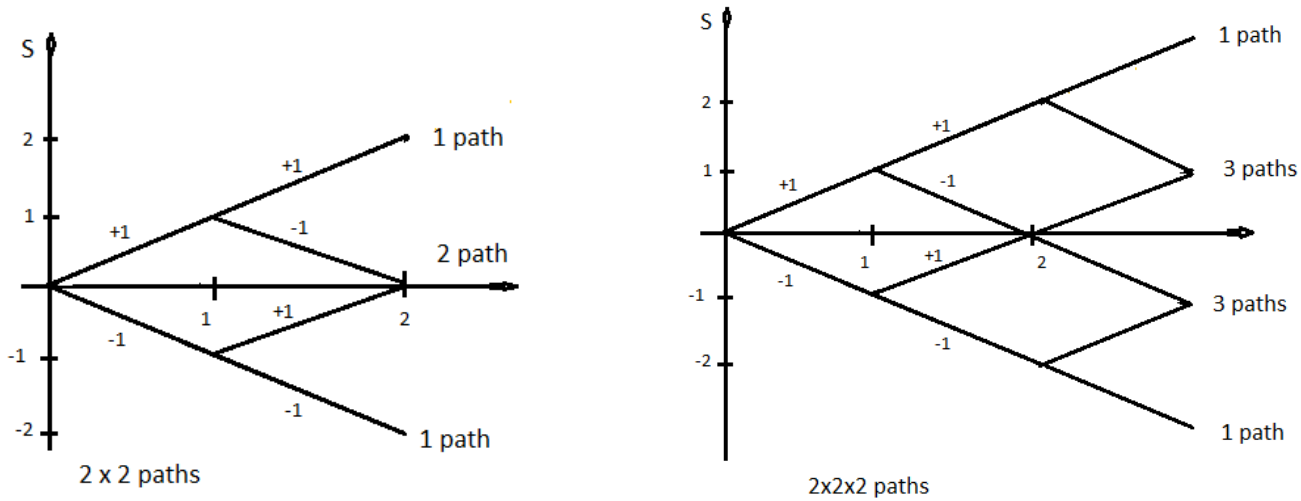

Figure: Schematically a Binomial model (Random walks) for 2 and 3 time steps can be represented as in the figure. Time (or the number of bets) horizontal axis and net profit shown in the vertical axis. Most number of paths end up in the center. Only 1 path is the most extreme up, 1 is the most extreme down.

For the probability per node i, where $i \in [0, N]$, we have

$$PDF[i] = \frac{N!}{i!(N-i)!} q^i (1-q)^{N-i} \tag{20}$$

Binomial Model

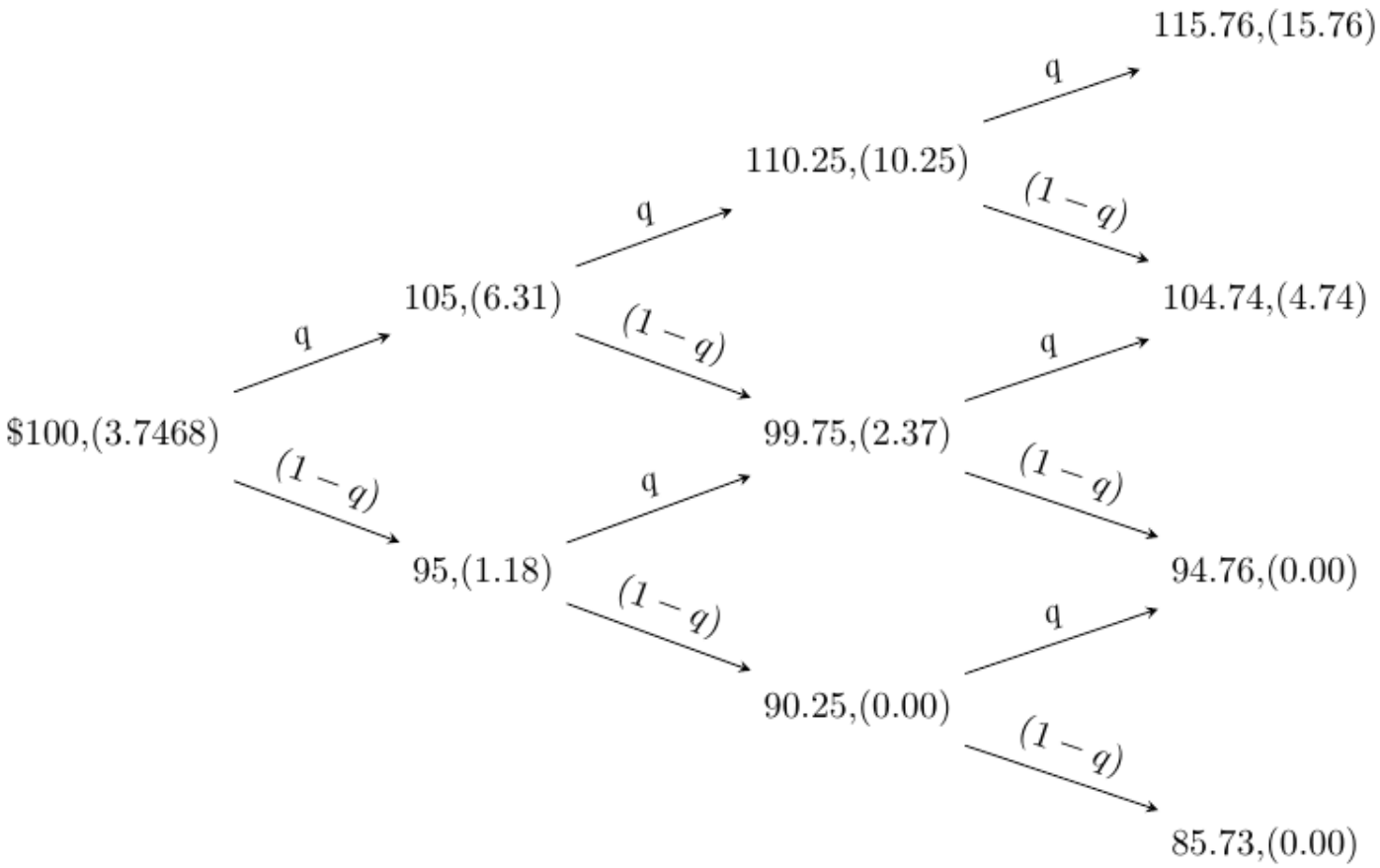

Figure: Binomial tree for 3 time steps. Time horizontal axis and stock price and option prices in the vertical axis. Here $U = -D = 0.05$, $R = 0$, $S_0 = 100$, $K = 100$. Slightly asymmetric tree. Symmetry requires $(1 + U)(1 + D) = 1$

Binomial Model

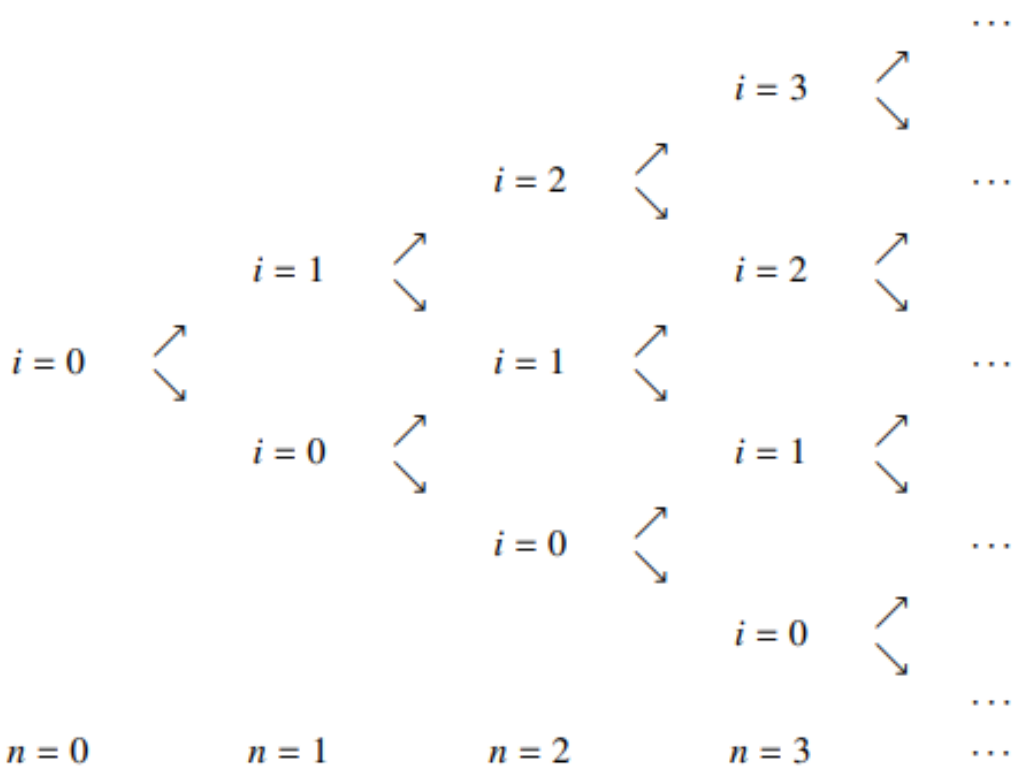

Figure: Price of stock in node (n, i), (refer to $S_{n,i}$ for stock price and $C_{n,i}$ for option price), is $S(n, i) = S(0)(1 + U)^i(1 + D)^{(n-i)}$. Here $n = 0, 1, \ldots$ and $i = 0, 1, \ldots, n$. At the highest point in the tree, $i = n$ and $S(n, n) = S(0)(1 + U)^n$, etc.

Binomial Model

Let us assume the risk-neutral pricing, already discussed to price a call option with payoff $h^{\text{call}}(S) = (S - K)^+$. A positive payoff is paid out only if the stock price end up above the strike.
In the Binomial model we proceed with the backward induction, the price of the option, $C(n, i)$ at time step n and node i, can be computed by taking the risk-neutral expectation

$$C(n, i) = \frac{qC(n+1, i+1) + (1-q)C(n+1, i)}{1+R} \tag{21}$$

At expiry date N,

$$C(N, i) = h(S(N, i)) \quad , \quad i = 0, 1, \ldots N \quad . \tag{22}$$

The risk-neutral probability is

$$q = \frac{R - D}{U - D} \quad . \tag{23}$$

In our example, $R = 0$, $U = 0.05$, and $D = -0.05$.

Binomial Model

At every node on the tree the value of the option is the discounted of the expected values of the options in the later time nodes. For every node before maturity we have

$$c_{i,j} = e^{-r\Delta t}\Big(qc_{i+1,j+1} + (1-q)c_{i+1,j}\Big) \quad . \tag{24}$$

We repeat this backwards starting from the time $N-1$.

For an American option, there is an added comparison. In this case we compare the immediate exercise against what will be my expected profit if I were to continue (the case of the dice game if there is any benefit on continuing to play versus immediate exercise). In the American option case the above equation changes to the following:

$$c_{i,j} = max\Big[e^{-r\Delta t}\big(qc_{i+1,j+1} + (1-q)c_{i+1,j}\big), 0\Big] \quad . \tag{25}$$

Modifications around these will be most of financial engineering and derivatives traded on Interest Rates.

Main IR Models

Main implementation methods in IR are the:

- PDE methods
- Monte-Carlo methods

Where do Binomial and Trinomial Trees fall? The Trinomial trees are very close to the PDEs as they can be obtained by extending the Trinomial trees to the above and below the diagonals, supplied with boundary conditions.

We will proceed step by step on the numerical implementation of these ideas.

Program Shell Code

Main01.cpp

```
#include <iostream>
using namespace std;

int main()
{
        //Display message
        cout << "Hello world!" << endl;

        char x;
        cin >> x;

        return 0;
}
```

Entering Data

Main02.cpp

```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
        double S0, U, D, R;
        //entering data
        cout << "Enter S0: "; cin >> S0; //  100
        cout << "Enter U: "; cin >> U;   //  0.05
        cout << "Enter D: "; cin >> D;   // -0.05
        cout << "Enter R: "; cin >> R;   //  0.1 - Cant be greater than U=0.05!

        // 1. make sure S0>0, (1+U)>0, (1+D)>0, (1+R)>0
        if (S0 <= 0 || U <= -1.0 || D <= -1.0 || R <= -1.0 || U <= D)
        {
                cout << "Illegal data ranges" << endl;
                cout << "Terminating program" << endl;
                return 1;
        }

        //2. Checking for arbitrage D<R<U; q>1, or q<0; q=(R-D)/(U - D)
        if (R <= D || R >= U)
        {
                cout << "Arbitrage exists" << endl;
                cout << "Terminating program" << endl;
                return 1;
        }
        cout << "Input data checked" << endl;
        cout << "There is no Arbitrage" << endl;
```

Entering Data

Main02.cpp (continued)

```
        //compute the risk-neutral probability
        double q = (R - D) / (U - D);
        cout << "Risk-neutral probability q = " << q << endl;

        //compute stock price at n=3, i=2;
        int n = 3; int i = 2;

        cout << "S(" << n << ","<<i<< ")=" <<S0*pow(1+U,i)*pow(1+D,n-i)<<endl;

        char x; cin >> x;

        return 0;
}
```

Functions

Main03.cpp

```
#include <iostream>
#include <cmath>
using namespace std;

int getInputData(double &S0, double &U, double &D, double &R)
{
        //entering data
        cout << "Enter S0: "; cin >> S0; //  100
        cout << "Enter U: "; cin >> U;   //  0.9
        cout << "Enter D: "; cin >> D;   // -0.3
        cout << "Enter R: "; cin >> R;   //  0.1
        // 1. make sure S0>0, (1+U)>0, (1+D)>0, (1+R)>0, U<D,
        otheriwise interchange U <-> D
        if (S0 <= 0 || U <= -1.0 || D <= -1.0 || R <= -1.0 || U <= D)
        {
                cout << "Illegal data ranges" << endl;
                cout << "Terminating program" << endl;
                return 1;
        }
        //2. Checking for arbitrage D < R < U; q > 1, or q < 0; q = (R-D) / (U - D)
        if (R <= D || R >= U)
        {
                cout << "Arbitrage exists" << endl;
                cout << "Terminating program" << endl;
                return 1;
        }
        cout << "Input data checked" << endl;
        cout << "There is no Arbitrage" << endl;
        return 0;
}
```

Functions

Main03.cpp

```
//Computing risk-neutral probability
double riskNeutralProb(double U, double D, double R)
{
        double q = (R - D) / (U - D);
        return q;
}
//computing stock proce and node (n,i)
double S(double S0, double U, double D, int n, int i)
{
        return S0 * pow(1 + U, i)*pow(1 + D, n - i);
}
int main()
{
        double S0, U, D, R;
        //getInput data
        if (getInputData(S0, U, D, R) == 1) return 1;

        // 1. make sure S0>0, (1+U)>0, (1+D)>0, (1+R)>0, U < D,
        otheriwise interchange U <-> D

        //compute the risk-neutral probability
        double q = riskNeutralProb(U, D, R);
        cout << "Risk-neutral probability q = " << q << endl;
        //compute stock price at n=3, i=2;
        int n = 3; int i = 2;
        cout << "S(" << n << "," << i << ")=" << S(S0,U,D,n,i) << endl;

        char x; cin >> x;
        return 0;

}
```

References

It is now useful to mention the usage of references for the address of variables in the function rather than the variables themselves. In the case the same copy of the variable is shared by the function and the calling program. Any changes made by the function to the variables remain available to the calling program even when the function returns control to the main program. The same result can be obtained using pointers, which we will see a but further below.
To get used to references, apply to an exchange function.

Exercise 1-1. Write an *interchange()* function that exchanges the values of two variables.

```
void interchange(double& a, double& b)
{
        double c = a;
        a = b, b = c;
        cout << "a = " << a << endl;
        cout << "b = " << b << endl;
};
```

Sorting Algorithm

Exercise 1-2. Write a sorting algorithm, putting a list of numbers into increasing order, by successively comparing adjacent elements and interchanging them. This is also an interview question.

```
void bubblesort(double a[], int N)
{
   for (int i = 1; i < N; i++)
   {
      for (int j = 1; j <= N-i;  j++)
      {
         if ( a[j] > a[j+1] ) interchange( a[j], a[j+1] );
      }
   }
}
```

Picks the first element and compares it with the next and orders. Then it picks the 2nd element and compares with the 3rd and orders. If the first was the biggest, then that moves consecutively to the 2nd, and then to the 3rd place. This way the algorithm brings the biggest to the very end. In the next iteration i changes to 2, and j still goes from 1 to $N - i$, because the biggest has gone to the end, then the next biggest will go to one before that, that is why j does not go beyond $N - i$. In the last i

Sorting Algorithm

For these types of sorting algorithms, see the book by Cormen et al, "Introduction to Algorithms". See also that the speed of performance on this algo is not very good, growth time is of order $\Theta(N^2)$. However, there is an efficient algorithm called "the merge-sort" algorithm which gives growth time $\Theta(N \cdot \ln N)$.

Seperate Compilation

BinModel01.cpp

```
#include <iostream>
#include <cmath>
using namespace std;
int getInputData(double &S0, double &U, double &D, double &R)
{
        //entering data
        cout << "Enter S0: "; cin >> S0; //  100
        cout << "Enter U: "; cin >> U;   //  0.9
        cout << "Enter D: "; cin >> D;   // -0.3
        cout << "Enter R: "; cin >> R;   //  0.1
      // 1. make sure S0>0, (1+U)>0, (1+D)>0, (1+R)>0, U<D, otheriwise interchange U<->D
        if (S0 <= 0 || U <= -1.0 || D <= -1.0 || R <= -1.0 || U <= D)
        {
                cout << "Illegal data ranges" << endl;
                cout << "Terminating program" << endl;
                return 1;
        }
        //2. Checking for arbitrage D < R < U; q > 1, or q < 0; q = (R-D) / (U - D)
        if (R <= D || R >= U)
        {
                cout << "Arbitrage exists" << endl;
                cout << "Terminating program" << endl;
                return 1;
        }
        cout << "Input data checked" << endl;
        cout << "There is no Arbitrage" << endl;

        return 0;
}
```

Seperate Compilation

BinModel01.cpp (continued)

```
//Computing risk-neutral probability
double riskNeutralProb(double U, double D, double R)
{
        double q = (R - D) / (U - D);
        return q;
}

//computing stock proce and node (n,i)
double S(double S0, double U, double D, int n, int i)
{
        return S0 * pow(1 + U, i)*pow(1 + D, n - i);
}
```

Seperate Compilation

```
BinModel01.h
#ifndef BinModel01_h
#define BinModel01_h

//computing riskNeutralProb(double U, double D, double R)
double riskNeutralProb(double U, double D, double R);

//computing the stockPrice at node (n,i)
double S(double S0, double U, double D, int n, int i);

//inputting, displaying and checking the model data
int getInputData(double &S0, double &U, double &D, double &R);

#endif
```

Seperate Compilation

Main04.cpp

```
#include <iostream>
#include <cmath>
using namespace std;
#include "BinModel01.h"

int main()
{
        double S0, U, D, R;

        //getInput data
        if (getInputData(S0, U, D, R) == 1) return 1;
        // 1. make sure S0>0, (1+U)>0, (1+D)>0, (1+R)>0, U<D,
        otheriwise interchange U <-> D

        //compute the risk-neutral probability
        double q = riskNeutralProb(U,D,R);
        cout << "Risk-neutral probability q = " << q << endl;

        //compute stock price at n=3, i=2;
        int n = 3; int i = 2;

        cout << "S("<<n<<","<<i<<")=" << S(S0, U, D, n, i) << endl;

        char x; cin >> x;

        return 0;
}
```

Binomial Pricer

Options01.h

```
#ifndef Options01_h
#define Options01_h

//inputting and displaying option data
int GetInputData(int& N, double& K);

//pricing European option
double PriceByCRR(double S0, double U, double D,
                  double R, int N, double K);

//computing call payoff
double CallPayoff(double z, double K);

#endif
```

Binomial Pricer

```
Options01.cpp
#include "Options01.h"
#include "BinModel01.h" :commented out <iostream>, <cmath>, std

int GetInputData(int& N, double& K)
{
   cout << "Enter steps to expiry N: "; cin >> N;
   cout << "Enter strike price K: "; cin >> K; cout << endl; return 0;
}
double PriceByCRR(double S0, double U, double D, double R, int N, double K)
{
   double q=RiskNeutProb(U,D,R);
   double Price[N+1];
   for (int i = 0 ; i <= N; i++)
   {
      Price[i]=CallPayoff(S(S0,U,D,N,i),K);
   }
   for (int n = N - 1; n >= 0; n--)
   {
      for (int i = 0; i <= n; i++)
      {
         Price[i]=( q * Price[i+1] + (1 - q) * Price[i] ) / (1+R);
      }
   }
   return Price[0];
}
double CallPayoff(double z, double K)
{
   if (z > K) return z - K;
   return 0.0;
}
```

Binomial Pricer

Main05.cpp

```
#include "BinModel01.h"
#include "Options01.h"
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    double S0,U,D,R;

    if (GetInputData(S0,U,D,R)==1) return 1;

    double K;    //strike price
    int N;       //steps to expiry

    cout << "Enter call option data:" << endl;
    GetInputData(N,K);
    cout << "European call option price = "
         << PriceByCRR(S0,U,D,R,N,K)
         << endl << endl;

    return 0;
}
```

Pointers - page 1

Options02.h

```
#ifndef Options02_h
#define Options02_h

//inputting and displaying option data
int GetInputData(int* PtrN, double* PtrK);

//pricing European option
double PriceByCRR(double S0, double U, double D,
                  double R, int N, double K);

//computing call payoff
double CallPayoff(double z, double K);

#endif
```

Pointers - page 2

```
Options02.cpp
#include "Options02.h"
#include "BinModel01.h" :Commented out <iostream>, <cmath>, std

int GetInputData(int* PtrN, double* PtrK)
{
   cout << "Enter steps to expiry N: "; cin >> *PtrN;
   cout << "Enter strike price K:     "; cin >> *PtrK; cout << endl;    return 0;
}
double PriceByCRR(double S0, double U, double D, double R, int N, double K)
{
   double q=RiskNeutProb(U,D,R);
   double Price[N+1];
   for (int i=0; i<=N; i++)
   {
      *(Price+i)=CallPayoff(S(S0,U,D,N,i),K);
   }
   for (int n=N-1; n>=0; n--)
   {
      for (int i=0; i<=n; i++)
      {
         *(Price+i)=(q*(*(Price+i+1))+(1-q)*(*(Price+i)))/(1+R);
      }
   }
   return *Price;
}
double CallPayoff(double z, double K)
{
   if (z > K) return z - K;
   return 0.0;
}
```

Pointers - page 3

Main06.cpp

```
#include "BinModel01.h"
#include "Options02.h"
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    double S0,U,D,R;

    if (GetInputData(S0,U,D,R)==1) return 1;

    double K;    //strike price
    int N;       //steps to expiry

    cout << "Enter call option data:" << endl;
    GetInputData(&N,&K);
    cout << "European call option price = "
         << PriceByCRR(S0,U,D,R,N,K)
         << endl << endl;

    return 0;
}
```

Home Work - Pointers

Exercise 1-4. Change the GetInputData() in BinModel01.h and BinModel01.cpp to have the parameters passed by pointers rather than by reference. What changes are needed in Main04.cpp to call the modified function.

Function Pointers

Options03.h

```
#ifndef Options03_h
#define Options03_h

//inputting and displaying option data
int GetInputData(int& N, double& K);

//pricing European option
double PriceByCRR(double S0, double U, double D,
                  double R, int N, double K,
                  double (*Payoff)(double z, double K));

//computing call payoff
double CallPayoff(double z, double K);

//computing put payoff
double PutPayoff(double z, double K);

#endif
```

Function Pointers

```
Options03.cpp
#include "Options03.h"
#include "BinModel01.h" : commented out <iostream>, <cmath>, std, Call/Put PayOff

int GetInputData(int& N, double& K)
{
   cout << "Enter steps to expiry N: "; cin >> N;
   cout << "Enter strike price K:    "; cin >> K; cout << endl; return 0;
}
double PriceByCRR(double S0, double U, double D,
                  double R, int N, double K,
                  double (*Payoff)(double z, double K))
{
   double q=RiskNeutProb(U,D,R);
   double Price[N+1];
   for (int i=0; i<=N; i++)
   {
      Price[i]=Payoff(S(S0,U,D,N,i),K);
   }
   for (int n=N-1; n>=0; n--)
   {
      for (int i=0; i<=n; i++)
      {
         Price[i]=(q*Price[i+1]+(1-q)*Price[i])/(1+R);
      }
   }
   return Price[0];
}
```

Function Pointers

Main07.cpp

```
#include "BinModel01.h"
#include "Options03.h"
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    double S0,U,D,R;

    if (GetInputData(S0,U,D,R)==1) return 1;

    double K;    //strike price
    int N;       //steps to expiry

    cout << "Enter call option data:" << endl;
    GetInputData(N,K);
    cout << "European call option price = "
         << PriceByCRR(S0,U,D,R,N,K,CallPayoff)
         << endl << endl;

    cout << "Enter put option data:" << endl;
    GetInputData(N,K);
    cout << "European put option price =  "
         << PriceByCRR(S0,U,D,R,N,K,PutPayoff)
         << endl << endl;

    return 0;
}
```

Home Work - Analytic pricer and comparison with Binomial tree

Exercise 1-5. Compare the Binomial tree price result with the analytic calculation, as obtained by Cox-Ross-Rubinstein (CRR) formula

$$C(0) = \frac{1}{(1+R)^N} \sum_{i=0}^{N} \frac{N!}{i!(N-i)!} q^i (1-q)^{N-i} h(S(N,i)) \quad . \tag{26}$$

Numerically do not calculate the factorials individually, optimize the calculation instead to

$$\frac{N!}{i!(N-i)!} = \frac{N(N-1)\cdots(N-i+1)}{i!} = \frac{N}{i}\frac{N-1}{i-1}\cdots\frac{N-i+1}{1} \quad , \tag{27}$$

as follows:

```
double NewtonSymb(int N, int i)
{
        if (i<0 || i>N) return 0;
        double result = 1;
        for (int k = 1; k <= i; k++) result = result * (N - i + k) / k;
        return result;
};
```

Binomial Pricer - Analytic Pricer function

```
Options01.cpp
#include "Options01.h"
#include "BinModel01.h" :commented out <iostream>, <cmath>, std


double PriceAnalytic(double S0, double U, double D,
        double R, int N, double K)
{
        double q = RiskNeutProb(U, D, R);
        vector<double> PDF(N + 1);
        double PDF_Sum = 0.0;

        double *Price = new double[N + 1];
        double Sum = 0.0;
        for (int i = 0; i <= N; i++)
        {
                Price[i] = CallPayoff(S(S0, U, D, N, i), K);
        }
        for (int i = 0; i <= N; i++)
        {
                PDF[i] = NewtonSymb(N, i)*pow(q, i)*pow(1 - q, N - i);
                PDF_Sum += PDF[i];
                cout << "i = " << i << ", PDF[i] = " << PDF[i] << endl;

                Sum += PDF[i] * Price[i];
        }
        cout << " PDF_Sum = " << PDF_Sum << endl;

        double result = Sum/pow(1.+R,N);
        delete[] Price;
        return result;
```

Home Work - Analytic pricer

Table: European Call Option Prices on Binomial Tree and Analytic for: $S_0 = 100$, $K = 100$, $r = 0\%$, for various expiries N. We take here $U = 0.1$, $D = -0.1$, resulting in $\sigma = 10\%$.

N	BinTree EuroCall	Analytic EuroCall
1	5	5
2	5.25	5.25
5	9.31269	9.31269
10	12.7451	12.7451
15	15.3504	15.3504
100	38.4472	38.4472

Compare this against the BS formula for extracted σ from U, D, R.

Home Work - Analytic pricer

Exercise 1-6. Plot the probabilities PDF[i]. Notice that when $R = 0$ and $U = -D$, then $q = 1/2$ and the probability distribution is symmetric. This corresponds to the stock price value of $S(N, i) = S_0 u^i d^{N-i}$. When you increase U against D, then the probability for up moves decreases and for down moves increases, but the tree nodes increase very much on the larger values of S, while not reaching very low values for S.

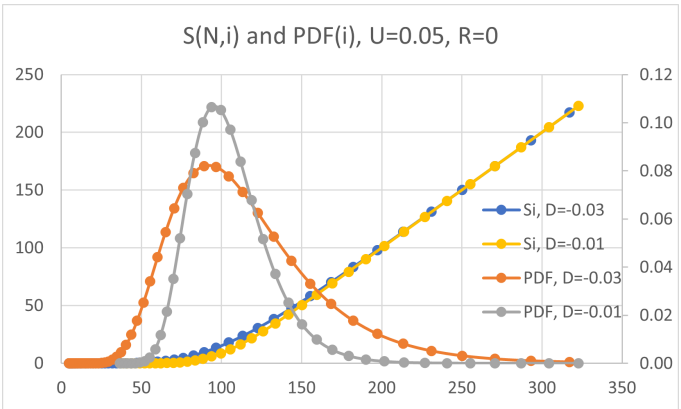

Figure: PDF for various $U = 0.05$, $D = -0.05$, $R = 0$. Here $N = 100$. The down probability increase when $U/D \gg 1$ but the maximum value the stock increases. The process remains martingale, centered around S_0. The two volatilities are not the same, one distribution is thinner. We keep $U = 0.05$, but lower D from -0.03 to -0.01.

Home Work - Continuous time

To move to continuous time, we use

$$1 + R = e^{r\Delta t} \quad , \tag{28}$$

using $\Delta t = T/N$, which will adjust the discounting in the previous tree to $R = e^{rT/N} - 1$. However, one needs to rescale the U and D, so as we wont have huge jumps within the one year ($u^N \sim 1.1^N$ extended over just one year period.)

Later we will generalize to relate U, D with the volatility parameters, but for now we take an example, still keeping with the parameters U and D, which are somewhat unknown and not directly observable.

Take $T = 1$, $S = 100$, $K = 100$, $u = 1.1$, $d = 1/u = 0.9091$, $N = 3$, and the continuous compounded interest rates of $r = 6\%$ per annum

$$\Delta t = \frac{T}{N} = \frac{1}{3} = 0.3333 \quad , \tag{29}$$

$$q = \frac{e^{r\Delta t} - d}{u - d} = \frac{e^{0.06 \times 0.3333} - 0.9091}{1.1 - 0.9091} = 0.5820 \quad , \tag{30}$$

from Eq. (40).

Home Work - Continuous time

The discount factor

$$DF = e^{-r\Delta t} = e^{-0.06\times 0.3333} = 0.9802 \quad . \tag{31}$$

Then we have

$$S_{3,0} = S \times d^N = 100 \times 0.9091^3 = 75.13 \quad , \tag{32}$$

The rest are calculated similarly

$$S_{3,2} = S_{3,1} \times \frac{u}{d} = 90.91 \times \frac{11}{0.9091} = 110.00 \quad . \tag{33}$$

Then the option value for the node $(3,2)$ will be

$$C_{3,2} = \max(S_{3,2} - 100, 0) = 10.00 \quad . \tag{34}$$

Then we perform the discounted expectation back through the tree. For node $(2,2)$, we have

$$\begin{aligned} C_{2,2} &= DF \times \Big(q \times C_{3,3} + (1-q)C_{3,2}\Big) \qquad (35) \\ &= 0.9802 \times (0.5820 \times 33.1000 + (1 - 0.5820) \times 10.0000) = 22.9801 \end{aligned}$$

Home Work - Continuous time

Build a table for European Call Option prices as function of N and see how good the convergence is.

Table: Multiplicative Binomial Tree valuation of European Call. $S = 100$, $K = 100$, $T = 1$, $r = 6\%$, $u = 1.1$, $d = 0.9091$.

$n = 0$ $t = 0$	1 0.3333	2 0.6667	3 1
			133.10, 33.1000
		121.00, 22.9801	
	110.00, 15.4471		110.00, 10.0000
100.00, 10.1457		100.00, 5.7048	
	90.91, 3.2545		90.91, 0.0000
		82.64, 0.0000	
			75.13, 0.0000

Home Work - Continuous time

$n = 0$ $t = 0$	1 $t = 0.3333$	2 t=0.6667	3 t=1
			133.10 33.1000
		121.00 22.9801	
	110.00 15.4471		110.00 10.0000
100.00 10.1457		100.00 5.7048	
	90.91 3.2545		90.91 0.0000
		82.64 0.0000	
			75.13 0.0000

Home Work - Analytic pricer: Matching U, D & σ

Exercise 1-7. Check the relationship between the U, D and the Black-Scholes volatilities, probabilities for up and down movements etc.

As part of the above question, one needs to think on how to match the distributions from BS and the ones from Binomial model. To match the two one needs to match the mean and the standard deviations after the one time step.

$$E^{\text{binomial}}\left[S_{n+1}|\mathcal{F}_n\right] = \left[qS(1+U) + (1-q)S(1+D)\right] = Se^{r\Delta t} \qquad (36)$$

$$E^{\text{bin}}\left[S_{n+1}^2|\mathcal{F}_n\right] = \left[qS^2(1+U)^2 + (1-q)S^2(1+D)^2\right] = S^2 e^{2(r-\frac{1}{2}\sigma^2)\Delta t} E\left[e^{2\sigma W(\Delta t)}\right]$$

$$= \quad S^2 e^{2(r+\frac{1}{2}\sigma^2)\Delta t} \qquad (37)$$

Home Work - Analytic pricer: Matching U, D & σ

Simplifying these equations results in:

$$qu + (1-q)d = e^{r\Delta t} \quad , \quad u = 1 + U \quad , \quad d = 1 + D \quad , \tag{38}$$

$$qu^2 + (1-q)d^2 = e^{(2r+\sigma^2)\Delta t} \quad . \tag{39}$$

Solving Eq.(38), we get

$$q = \frac{e^{r\Delta t} - 1 - D}{U - D} \tag{40}$$

The second equation Eq.(39) provides a link between U, D and the volatility σ. Substituting q in the second equation, we get a relationship between U, D and the volatility.

$$e^{r\Delta t}(u + d) - du = e^{(2r+\sigma^2)\Delta t} \quad . \tag{41}$$

But notice, now we have one equation for two unknowns. Therefore there could be different choices for the relationship between U, D and the volatility σ.

Home Work - Analytic pricer: Matching U, D & σ

There are three popular choices: 1) Cox-Ross-Rubinstein $ud = 1$. 2) Jarrow and Rudd choose equal probabilities for the up and down price movement, $q = 1 - q = 1/2$. A third popular one is assuming $ud = e^{2v\Delta t}$, where v is a scalar number. Then a possible solution is

$$u = e^{v\Delta t + \sigma\sqrt{\Delta t}} \quad , \quad d = e^{v\Delta t - \sigma\sqrt{\Delta t}} \quad , \tag{42}$$

and the probability for the up down movement

$$q = \frac{1}{2} + \frac{1}{2}\left(\frac{\mu - v}{\sigma}\right)\sqrt{\Delta t} \quad , \quad 1 - q = \frac{1}{2} - \frac{1}{2}\left(\frac{\mu - v}{\sigma}\right)\sqrt{\Delta t} \quad , \tag{43}$$

where $\mu = r - \frac{1}{2}\sigma^2$. What we chose here is

$$\mu = r - \frac{1}{2}\sigma^2 \quad , \quad v = r + \frac{1}{2}\sigma^2 \quad , \tag{44}$$

which results in $q = \frac{1}{2} - \frac{1}{2}\sigma\sqrt{\Delta t}$, and $1 - q = \frac{1}{2} + \frac{1}{2}\sigma\sqrt{\Delta t}$.

Home Work - Analytic pricer: Matching U, D & σ

The most important relationships here are ($v = 0$)

$$U = e^{\sigma\sqrt{\Delta t}} - 1 \quad , \quad D = e^{-\sigma\sqrt{\Delta t}} - 1 \quad , \tag{45}$$

and the probability for the up down movement

$$q = \frac{1}{2} + \frac{1}{2}\left(\frac{\mu}{\sigma}\right)\sqrt{\Delta t} \quad , \quad 1 - q = \frac{1}{2} - \frac{1}{2}\left(\frac{\mu}{\sigma}\right)\sqrt{\Delta t} \quad , \tag{46}$$

where $\mu = r - \frac{1}{2}\sigma^2$.

Home Work - Analytic pricer: Matching U, D & σ

Remaining still in the general tree, with $ud = e^{2v\Delta t}$. Another interesting choice for the value of v is $v = -\frac{1}{N\Delta t} \ln \frac{S_0}{K}$, use also $N\Delta t = T$ to write

$$v = -\frac{1}{N\Delta t}\Big[\ln S_0 - \ln K\Big] = -\frac{1}{T}\Big[\ln S_0 - \ln K\Big] \quad . \tag{47}$$

This choice centers the tree around the value of the strike K. For instance take the position of the up node and down node and the ones in the middle $S_0 u^0 d^N = Ke^{-\sigma N\sqrt{\Delta t}}$, and $S_0 u^N d^0 = Ke^{\sigma N\sqrt{\Delta t}}$. The tree is centered in the middle. If we select the number of time steps as odd then the strike K is in the middle of two nodes, and if the number of time steps is even then the middle node will coincide with the strike K.

This choice should remove oscillatory behavior on pricing digitals arising from the positioning of the central node of the tree against the strike K. Also notice that the drift in real world does not change the option price in the risk-neutral measure.

Home Work - Digital Call and Delta

Exercise 1-8. Use BS volatility $\sigma = 20\%, 30\%, 50\%$, and convert to the U, D factors of the Binomial Model, to calculate the price of Digitals, an in particular the greeks, Delta and Vega. PV and Delta are shown below. Reproduce these and also the Vega.

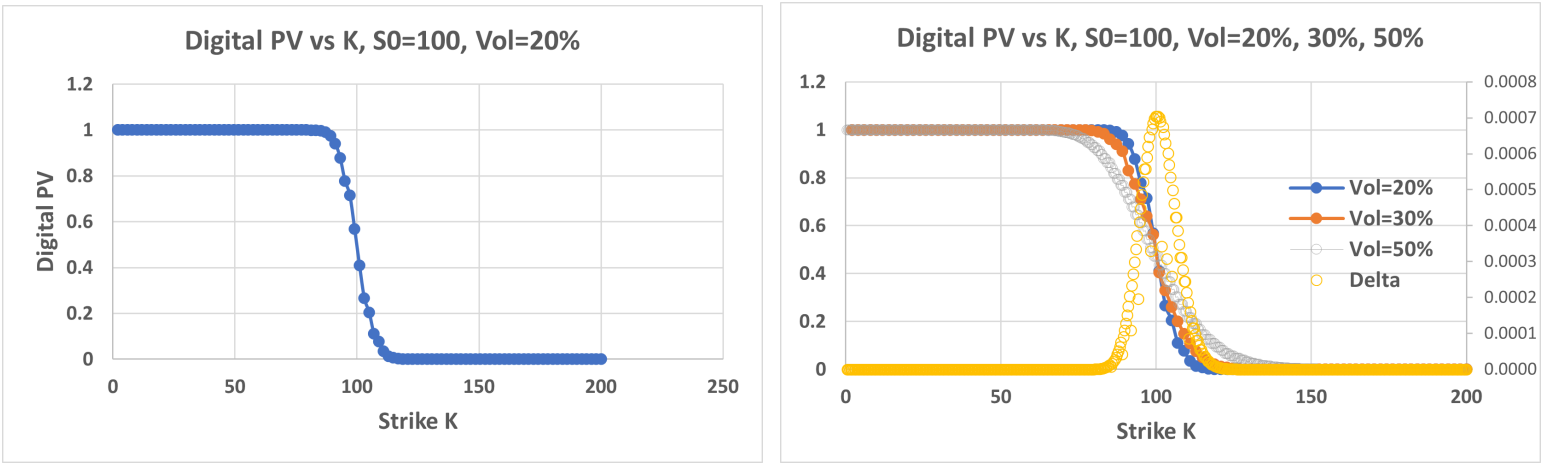

Figure: Digital Option PV versus strike K for different BS volatilities, 20%,30%,50%, respectively. These have been obtained with CRR. The Bump in the middle, secondary axis, shows Delta obtained by using a bump size of $\epsilon = 0.01$.

Home Work - Double Knock Out Digitals

Exercise 1-9. Use BS volatility $\sigma = 20\%, 30\%, 50\%$, and convert to the U, D factors of the Binomial Model, to calculate the price of Double Knock Out products, an in particular the greeks, Delta and Vega.

For the analytically driven people, use the following to calculate the price of Double Knock Outs analytically, as well as numerically.
Group together the different terms as below

$$C(0) = S_0 \sum_{i=k}^{N} \binom{N}{i} \frac{q^i u^i}{(1+R)^i} \frac{(1-q)^{(N-i)} d^{N-i}}{(1+R)^{N-i}} \tag{48}$$

$$-K \sum_{i=k}^{N} \binom{N}{i} \frac{q^i}{(1+R)^i} \frac{(1-q)^{(N-i)}}{(1+R)^{N-i}} \quad . \tag{49}$$

Call

$$p = \frac{qu}{1+R} \quad , \quad \text{then} \quad 1-p = \frac{qd}{1+R} \quad . \tag{50}$$

Home Work - Double-no-Touch Digitals

The discrete Black-Scholes equation becomes

$$C(0) = S_0 \sum_{i=k}^{N} \binom{N}{i} p^i (1-p)^{N-i} - K \frac{1}{(1+R)^N} \sum_{i=k}^{N} \binom{N}{i} q^i (1-q)^{(N-i)} \quad . (51)$$

The sums beginning at k, where $k = k(N)$ is the smallest integer i such that $S_0 u^i d^{N-i} > K$. Use also the following notation for the *incomplete Binomial sum*

$$\Phi(N, k, q) = \sum_{i=k}^{N} \binom{N}{i} q^i (1-q)^{(N-i)} \quad , \qquad (52)$$

to compactify the Call option price to the discrete BS equation

$$C(0) = S_0 \Phi(N, k, q) - K \frac{1}{(1+R)^N} S_0 \Phi(N, k, p) \quad . \qquad (53)$$

Apply these calculations for the Double-no-Touch Digitals out and for Up and Out barriers. Show numerical and analytical comparisons.

Home Work - Double-no-Touch Digitals

Compare this against the BS equation

$$C(0) = S_0 N(d_1) - K e^{-rT} N(d_2) \quad , \tag{54}$$

where $d_1 = \left(\ln \frac{S_0}{K} + (r + \sigma^2/2)T \right) / \sigma\sqrt{T}$,
$d_2 = \left(\ln \frac{S_0}{K} + (r - \sigma^2/2)T \right) / \sigma\sqrt{T}$, and $N(x) = \int_{-\infty}^{x} e^{-y^2/2} \mathrm{d}y / \sqrt{2\pi}$.

Home Work - Double-no-Touch Digitals

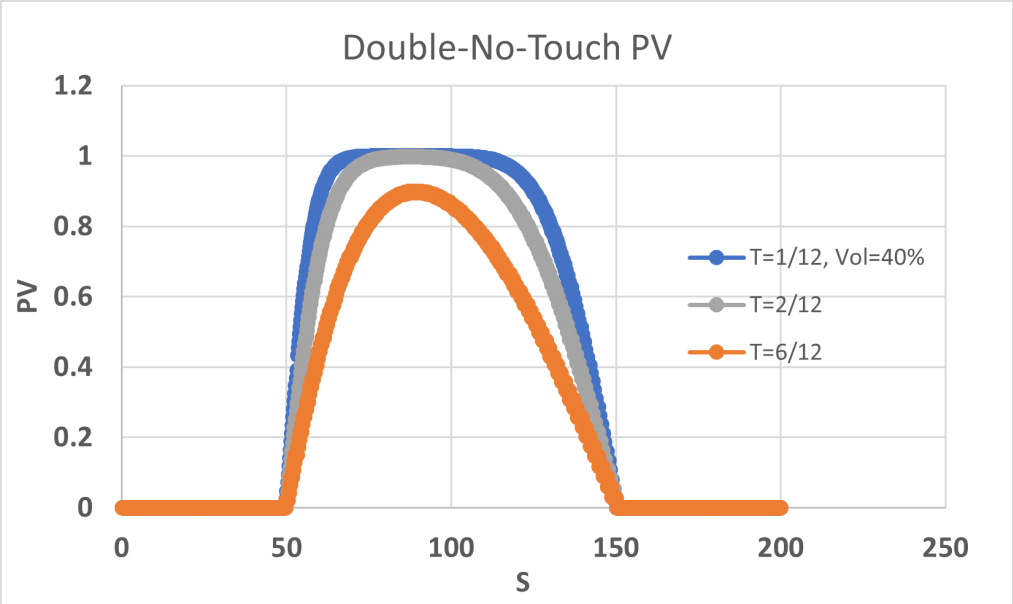

Figure: Double-No-Touch PVs as function of the price S_0, for Low and Upper barriers, 50 and 150, respectively, are shown.

Create a Block-Scholes object with S_0, r, K, T, as private elements, add a BS pricer for Calls, Puts, Digitals, using the BS formula.
Add one more function that has one purpose to convert the volatility σ to BinModel object with $S0$, U, D, R. The BinModel class can do the pricing for Calls, Put, DNTs, Single barriers, etc.

Additive Binomial model

Perhaps the biggest breakthrough in the modeling comes from the following:
The following substitution has proven to have advantages in the modeling of derivatives is to use instead of the price the logarithm $x = \ln S$. Applying Ito's lemma, we get

$$\mathrm{d}x = \left(r - \frac{1}{2}\sigma^2\right)\mathrm{d}t + \sigma \mathrm{d}W(t) \quad , \tag{55}$$

with starting point of $x_0 = \ln S_0$ as the center node, solution to the above $x = x_0 + \nu t + \sigma W(t)$. Then the variable x in a small interval Δt can go either up to the level $x_0 + \Delta x_u$ or down to $x_0 + \Delta x_d$, with probabilities p_u and $p_d = 1 - p_u$, respectively.

Additive Binomial model

We equate the mean and variance of the binomial process for x with the mean and variance of the continuous time process over the time interval Δt. The mean of the continuous process in Eq. (55)

$$E[\Delta x] = p_u \Delta x_u + p_d \Delta x_d = \nu \Delta t \quad , \tag{56}$$

where $\nu = r - \sigma^2/2$, and

$$E[\Delta x^2] = p_u \Delta x_u^2 + p_d \Delta x_d^2 = \sigma^2 \Delta t + \nu^2 \Delta t^2 \quad . \tag{57}$$

The term $\nu^2 \Delta t^2$ was introduced by Trigeorgis and can improve the numerical accuracy in not so small time steps trees. From these two equations, combined with $p_u + p_d = 1$, we have three equations and four unknowns. Therefore we have a free choice for one of the parameters. Two are the obvious choices, the CRR, and the JR, but we focus on the CRR case.

Additive Binomial model

The CRR case, of Equal Jump size case:

$$p_u \Delta x + p_d(-\Delta x) = \nu \Delta t \quad , \tag{58}$$

$$p_u \Delta x^2 + p_d \Delta x^2 = \sigma^2 \Delta t + \nu^2 \Delta t^2 \quad , \tag{59}$$

gives

$$\Delta x \quad = \quad \sqrt{\sigma^2 \Delta t + \nu^2 \Delta t^2} \quad . \tag{60}$$

Additive Binomial model

Then from Eq. (58)

$$p_u \Delta x - (1 - p_u)\Delta x = \nu \Delta t \quad , \quad 2p_u \Delta x - \Delta x = \nu \Delta t \quad , \tag{61}$$

$$p_u \quad = \quad \frac{1}{2} + \frac{1}{2}\frac{\nu \Delta t}{\Delta x} \quad . \tag{62}$$

Then the nodes (i, j) for $i = 0, \ldots, N$ and $j = 0, \ldots, i$, on the tree will be given by

$$S_{i,j} \quad = \quad \exp\left(x_0 + j\Delta x_u + (i - j)\Delta x_d\right) \quad . \tag{63}$$

In the above $e^{x_0} = S_0$, meaning that in the above we have $S_{i,j} = S_0 \exp\left(j\Delta x_u + (i - j)\Delta x_d\right)$.

Additive Binomial model

However, there is a better implementation in the case of equal up and down jumps. We change the structure of the tree, and now use an index j to account for the level of the asset price rather then to account for the number of the up jumps. At the last time slice, N, the level of the index takes values $j = -N, -N+2, \ldots, N-2, N$, and the position in the tree will be given by

$$\begin{aligned} S_{i,j} &= \exp\left(x_0 + j\Delta x\right) \\ &= S_0 \exp\left(j\Delta x\right) \quad , \quad j = -i, \ldots, i \quad , \quad i = 0, \ldots, N \quad . \end{aligned} \tag{64}$$

Lecture 2

Concepts of Classes for Option Pricing

Concepts of Classes for Option Pricing

In the first class we separated the functions into two groups. Ones falling under BinModel.h, and another set of functions falling under Options.h and the corresponding cpp files, which contained their definition.

* The BinModel.h contained elements relevant for the Binomial Model, like the calculation of the risk-neutral probability $q = (R - D)/(U - D)$, using the function RiskNeutrProb(S0,U,D,R). Then the other function, GetInput(S0,U,D,R), the input function that sets the parameters for the code and the functions.

* The Options.h needed details of the Option, like Expiry=N, and strike=K. Then it needed a function to price the Option, PriceByCRR(S0, U, D, R, N, K, (*PayOff)(S,K))

Concepts of Classes for Option Pricing

What could be the critiques of the previous code?

- In Options.h, what if we needed a payoff that contained two strikes, like a double-digital, that pays 1, if $K_1 < S < K_2$? Then we need to modify PriceByCRR() to take a payoff with two strikes.
- Notice also the long notation: PriceByCRR(S0, U, D, R, N, K, (*PayOff)(S,K))

The extend-ability shortcoming is a major issue. The non-streamlined writing with long list of parameters is also another issue.

Now we introduce a Class object **BinModel**, which contains a group of elements, and a set of functions, that are unique for a model.

Binomial Model Class

What is unique about a Binomial Model class **BinModel**?

- It needs to build a binomial tree. Therefore it needs the starting point, first node of the tree, S_0. It needs the factors U and D to move from one time slice to the next. It also needs the discounting rate, R, to go from one time slice backwards to the initial node point.
- Then it needs two functions that make use of these data. The first function that is needed from a Binomial Model is the RiskNeutral function. It is needed in pricing of the options.

  The second function is the one that produces the stock price in every node (n, i), $S(n, i)$.

  The other auxiliary functions are the function that sets the data values, GetInputData, which is like a constructor of the class. And the function getR(), that is needed for the discounting in the Option pricing.

This is the totality of the functionality of the Binomial Model Class.

First Class: Binomial Model

BinModel02.h

```
#ifndef BinModel02_h
#define BinModel02_h

class BinModel
{
   private:
      double S0;
      double U;
      double D;
      double R;

   public:
      //computing risk-neutral probability
      double RiskNeutProb();

      //computing the stock price at node n,i
      double S(int n, int i);

      //inputting, displaying and checking model data
      int GetInputData();

      double GetR();
};

#endif
```

First Class: Binomial Model

BinModel02.cpp

```
#include "BinModel02.h"
#include <iostream>
#include <cmath>
using namespace std;

double BinModel::RiskNeutProb()
{
   return (R-D)/(U-D);
}

double BinModel::S(int n, int i)
{
   return S0*pow(1+U,i)*pow(1+D,n-i);
}

int BinModel::GetInputData()
{
   //entering data
   cout << "Enter S0: "; cin >> S0;
   cout << "Enter U: "; cin >> U;
   cout << "Enter D: "; cin >> D;
   cout << "Enter R: "; cin >> R;
   cout << endl;
```

First Class: Binomial Model

BinModel02.cpp (cont.)

```
   //making sure that 0 < S0, -1 < D < U, -1 < R
   if (S0 <= 0.0 || U <= -1.0 || D <=-1.0 || U <= D || R <= -1.0)
   {
      cout << "Illegal data ranges" << endl;
      cout << "Terminating program" << endl;
      return 1;
   }

   //checking for arbitrage
   if (R>=U || R<=D)
   {
      cout << "Arbitrage exists" << endl;
      cout << "Terminating program" << endl;
      return 1;
   }

   cout << "Input data checked" << endl;
   cout << "There is no arbitrage" << endl << endl;

   return 0;
}

double BinModel::GetR()
{
   return R;
}
```

Binomial Model - question?

Why do we not use the variables in the function like in Class 1?

```
int BinModel::GetInputData(S0, U, D, R) -> int BinModel::GetInputData()
```

The reason for this is that the class knows its private elements. There is no need to call these elements explicitly in the class functions because they have access to them.
The BinModel object has 4 private data, and it has a number of functions that it manipulates them. This is a closed box, and the only way to access this box is by using the classes of the box. This type of object is more intricate than the simple "int" or "double".

Now we modify the Options03.h $->$ Options04.h to make use of the BinModel class.

Modified Options.h to use Class BinModel

```
Options04.h
#ifndef Options04_h
#define Options04_h

#include "BinModel02.h"

//inputting and displaying option data
int GetInputData(int& N, double& K);

//pricing European option
double PriceByCRR(BinModel Model, int N, double K,
                  double (*Payoff)(double z, double K));

//computing call payoff
double CallPayoff(double z, double K);

//computing put payoff
double PutPayoff(double z, double K);

#endif
```

Notice the difference introduced here:

```
Options03.h
double PriceByCRR(double S0, double U, double D, double R, int N, double K,
                  double (*Payoff)(double z, double K));
->
Options04.h
double PriceByCRR(BinModel Model, int N, double K,
                  double (*Payoff)(double z, double K))
```

Modified Options.cpp to use Class BinModel

```
Options04.cpp

#include "Options04.h"
#include "BinModel02.h"
#include <iostream>
#include <cmath>
using namespace std;

int GetInputData(int& N, double& K)
{
   cout << "Enter steps to expiry N: "; cin >> N;
   cout << "Enter strike price K:    "; cin >> K;
   cout << endl;
   return 0;
}

double CallPayoff(double z, double K)
{
   if (z > K) return z-K;
   return 0.0;
}

double PutPayoff(double z, double K)
{
   if (z < K) return K-z;
   return 0.0;
}
```

Modified Options.cpp to use Class BinModel

```
Options04.cpp (cont.)

double PriceByCRR(BinModel Model, int N, double K,
                  double (*Payoff)(double z, double K))
{

   double q=Model.RiskNeutProb();
   double Price[N+1];

   for (int i=0; i<=N; i++)
   {
      Price[i]=Payoff(Model.S(N,i),K);
   }

   for (int n=N-1; n>=0; n--)
   {

      for (int i=0; i<=n; i++)
      {
         Price[i]=(q*Price[i+1]+(1-q)*Price[i])
                  /(1+Model.GetR());
      }

   }
   return Price[0];
}
```

Modified Main.cpp to use Class BinModel

Main08.cpp

```
#include "BinModel02.h"
#include "Options04.h"
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    BinModel Model;

    if (Model.GetInputData()==1) return 1;

    double K;    //strike price
    int N;       //steps to expiry

    cout << "Enter call option data:" << endl;
    GetInputData(N,K);
    cout << "European call option price = "
         << PriceByCRR(Model,N,K,CallPayoff)
         << endl << endl;

    cout << "Enter put option data:" << endl;
    GetInputData(N,K);
    cout << "European put option price =  "
         << PriceByCRR(Model,N,K,PutPayoff)
         << endl << endl;

    return 0;
}
```

BinModel Cass - Changes on Options03.h/cpp

The change in the code that we mentioned previously, to accommodate the introduction of the class Model:

```
Options03.h
double PriceByCRR(double S0, double U, double D,
                  double R, int N, double K,
                  double (*Payoff)(double z, double K));
->

Options04.h
double PriceByCRR(BinModel Model, int N, double K,
                  double (*Payoff)(double z, double K))
```

Everything is the same apart from reducing the number of arguments. The reason being that the "Model" knows its own private elements. They are inside the Class.

BinModel Cass - Changes on Options03.h/cpp

```
Options03.cpp
    for (int i=0; i<=N; i++)
    {
        Price[i]=Payoff(S(S0,U,D,N,i),K);
    }
    for (int n=N-1; n>=0; n--)
    {
        for (int i=0; i<=n; i++)
        {
            Price[i]=(q*Price[i+1]+(1-q)*Price[i])/(1+R);
        }
    }
->
Options04.cpp
    for (int i=0; i<=N; i++)
    {
        Price[i]=Payoff(Model.S(N,i),K);
    }
    for (int n=N-1; n>=0; n--)
    {
        for (int i=0; i<=n; i++)
        {
            Price[i]=(q*Price[i+1]+(1-q)*Price[i]) / (1+Model.GetR());
        }
    }
```

The calculation of the stock price at nodes (n, i), $S(n, i)$, is done through the function of the class, Model.S(N,i), in $Price[i] = Payoff(Model.S(N, i), K)$. Similarly, the discount rate R, is obtained from the class using *Model.getR()*

BinModel Cass - Changes in Main07 to Main09

```
Main07.cpp
PriceByCRR(S0,U,D,R,N,K,CallPayoff);

* Here we did not have any class at all

->

Main08.cpp
PriceByCRR(Model,N,K,CallPayoff)

* Here we have BinModel Model class object but not a Call/Put or EurOption
 object, we will see next

->

Main09.cpp
Option1.PriceByCRR(Model,Option1.get(K))

* Here we have both a BinModel Model Object and Call class Option1 object
```

Options Class - Inheritance

Now we introduce a new and important class, the EurOption Class. Then we will inherit two classes based on it, namely, the Call and Put subclasses.
The idea is that Calls are a subset of European Options. Puts are another subsets. Another subset would be the Digitals with two strikes, K_1, and K_2 etc.

Options Class EurOption - Inheritance

Options05.h

```
#ifndef Options05_h
#define Options05_h

#include "BinModel02.h"

class EurOption
{
   private:
      //steps to expiry
      int N;
      //pointer to payoff function
      double (*Payoff)(double z, double K);
   public:
      void SetN(int N_){N=N_;}
      void SetPayoff
         (double (*Payoff_)(double z, double K))
         {Payoff=Payoff_;}
      //pricing European option
      double PriceByCRR(BinModel Model, double K);
};

//computing call payoff
double CallPayoff(double z, double K);
```

Options Class - Inheritance

Options05.h (continued)

```
class Call: public EurOption
{
   private:
      double K; //strike price
   public:
      Call(){SetPayoff(CallPayoff);}
      double GetK(){return K;}
      int GetInputData();
};

//computing put payoff
double PutPayoff(double z, double K);

class Put: public EurOption
{
   private:
      double K; //strike price
   public:
      Put(){SetPayoff(PutPayoff);}
      double GetK(){return K;}
      int GetInputData();
};

#endif
```

EurOption Class, Call and Put

What is unique about a European Option Model class **EurOption** and its inherited classes **Call** and **Put**?

- The pricing happens in the base class function PriceByCRR(Model, K). The Call and Put functions use this function because Call and Puts are also EurOption objects, as they are inherited from it.
- The base class EurOption has two private elements: 1. the Expiry=N, and 2. a pointer to a function that takes two double arguments, and returns a double. Obviously this is a Payoff type function. The pointer does not point to a specific payoff yet. This is the idea of extend-ability. The subclasses will point the pointer to the proper payoffs.
- The base class EurOption, provides two Set() type functions, for the Expiry, SetN(N), and the Pointer, SetPayoff(CallPayoff). These are to be used by the subclasses to set the expiry, and to point the pointers. Without them, the subclasses cannot set their values as they are private elements of EurOption

Options Class - Inheritance

Options05.cpp

```
#include "Options05.h"
#include "BinModel02.h"
#include <iostream> , #include <cmath>,  using namespace std;

double EurOption::PriceByCRR(BinModel Model, double K)
{
   double q=Model.RiskNeutProb();
   double Price[N+1];
   for (int i=0; i<=N; i++)
   {
      Price[i]=Payoff(Model.S(N,i),K);
   }
   for (int n=N-1; n>=0; n--)
   {
      for (int i=0; i<=n; i++)
      {
         Price[i]=(q*Price[i+1]+(1-q)*Price[i])
            /(1+Model.GetR());
      }
   }
   return Price[0];
}

double CallPayoff(double z, double K)
{
   if (z > K) return z-K;
   return 0.0;
}
```

Options Class - Inheritance

Options05.cpp (continued)

```
int Call::GetInputData()
{
   cout << "Enter call option data:" << endl;
   int N;
   cout << "Enter steps to expiry N: "; cin >> N;
   SetN(N);
   cout << "Enter strike price K:    "; cin >> K;
   cout << endl;
   return 0;
}

double PutPayoff(double z, double K)
{
   if (z < K) return K-z;
   return 0.0;
}

int Put::GetInputData()
{
   cout << "Enter put option data:" << endl;
   int N;
   cout << "Enter steps to expiry N: "; cin >> N;
   SetN(N);
   cout << "Enter strike price K:    "; cin >> K;
   cout << endl;
   return 0;
}
```

Modeified Main - incorporate BinModel and Options Class

Main09.cpp

```
#include "BinModel02.h"
#include "Options05.h"
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    BinModel Model;

    if (Model.GetInputData()==1) return 1;

    Call Option1;
    Option1.GetInputData();
    cout << "European call option price = "
         << Option1.PriceByCRR(Model,Option1.GetK())
         << endl << endl;

    Put Option2;
    Option2.GetInputData();
    cout << "European put option price = "
         << Option2.PriceByCRR(Model,Option2.GetK())
         << endl << endl;

    return 0;
}
```

Options Class - Inheritance

What is the structure of the Options class and its sub-classes Call and Put?

The base class contains two Private data, the Expiry N, and a pointer to a double function with two arguments. This is a payoff-type function, that will be set in the sub-classes to point to the appropriate payoffs, allowing also extension to other types of payoffs with one strike.

The base class seems to be doing all the work. It has been equipped with one pricing function of CRR-type, and two other functions to set the value of the Expiry, and to make the pointer point to the right payoff.

The base class allows the sub-classes to use these functions.

*However, there is one limitation still of these three classes. The pointer to a payoff function in the base class takes two double arguments, one for the value of the stock at expiry, and another for the strike K. But this is still only one strike. What if we needed two strikes, K_1 and K_2 or even something more generic?

This shortcoming will be corrected in the modification provided below in

Use Virtual Functions in Options.h class

Options06.h

```
#ifndef Options06_h
#define Options06_h

#include "BinModel02.h"

class EurOption
{
   private:
      int N; //steps to expiry
   public:
      void SetN(int N_){N=N_;}
      //Payoff defined to return 0.0
      //for pedagogical purposes.
      //To use a pure virtual function replace by
      //virtual double Payoff(double z)=0;
      virtual double Payoff(double z){return 0.0;}
      //pricing European option
      double PriceByCRR(BinModel Model);
};
```

Use Virtual Functions in Options.h class

Options06.h (continued)

```
class Call: public EurOption
{
   private:
      double K; //strike price
   public:
      void SetK(double K_){K=K_;}
      int GetInputData();

      virtual double Payoff(double z);
};

class Put: public EurOption
{
   private:
      double K; //strike price
   public:
      void SetK(double K_){K=K_;}
      int GetInputData();

      virtual double Payoff(double z);
};

#endif
```

Use Virtual Functions in Options.h class

Options06.cpp

```
#include "Options06.h"
#include "BinModel02.h", #include <iostream>, #include <cmath>, using namespace std;

double EurOption::PriceByCRR(BinModel Model)
{
    double q=Model.RiskNeutProb();
    double Price[N+1];
    for (int i=0; i<=N; i++)
    {
        Price[i]=Payoff(Model.S(N,i));
    }
    for (int n=N-1; n>=0; n--)
    {
        for (int i=0; i<=n; i++)
        {
            Price[i]=(q*Price[i+1]+(1-q)*Price[i])
                /(1+Model.GetR());
        }
    }
    return Price[0];
}

double Put::Payoff(double z)
{
    if (z < K) return K-z;
    return 0.0;
}
```

Use Virtual Functions in Options.h class

Options06.cpp (continued)

```
int Call::GetInputData()
{
    cout << "Enter call option data:" << endl;
    int N;
    cout << "Enter steps to expiry N: "; cin >> N;
    SetN(N);
    cout << "Enter strike price K:    "; cin >> K;
    cout << endl;
    return 0;
}

double Call::Payoff(double z)
{
    if (z > K) return z - K;
    return 0.0;
}

int Put::GetInputData()
{
    cout << "Enter put option data:" << endl;
    int N;
    cout << "Enter steps to expiry N: "; cin >> N;
    SetN(N);
    cout << "Enter strike price K:    "; cin >> K;
    cout << endl;
    return 0;
}
```

Use Virtual Functions Options.h class in Main

Main10.cpp (continued)

```
#include "BinModel02.h"
#include "Options06.h", #include <iostream>, #include <cmath>, using namespace std;

int main()
{
    BinModel Model;

    if (Model.GetInputData()==1) return 1;

    Call Option1;
    Option1.GetInputData();
    cout << "European call option price = "
         << Option1.PriceByCRR(Model)
         << endl << endl;

    Put Option2;
    Option2.GetInputData();
    cout << "European put option price = "
         << Option2.PriceByCRR(Model)
         << endl << endl;

    return 0;
}
```

Virtual Functions in Options.h

1. Important change in Options06.h has been the introduction of the virtual function payoff. That means that the compiler will select at run time which payoff to use. The base function Payoff(z) contains just the Stock price, no mention of strike K.

2. The strike K has been removed from the base class EurOption with the aim of making the class more generic, to allow for extension to sub-classes with more strikes.

Exercise 2.1 As an exercise, inherit a DoubleDigital Option from the EurOption class, and extend the code to price, using the vrtual function provided in Option06.h

$$h^{\text{doubDigit}}(z) = \begin{cases} 1, & \text{if } K_1 < z < K_2 \\ 0, & \text{otherwise} \quad . \end{cases} \tag{65}$$

Lecture 3

American Options

American Options

Binomial trees are very well suited for pricing options with callability features like American options. These options do not have a single expiry date but can be called at any of the time slices on the binomial tree.
The price of the American option at a node we denote by $H(n,i)$ and the payoff $h(n,i) = (S(n,i) - K)^+$
The American option, at Expiry, the last time slice, is given by

$$H(N,i) = h(N,i) = (S(N,i) - K)^+ \tag{66}$$

At previous times, it is given by

$$H(n,i) = \max\left(\frac{qH(n+1,i+1) + (1-q)H(n+1,i)}{1+R}, h(S(n,i))\right) \tag{67}$$

The American option price will be the one of the first node $H(0)$.

American Options

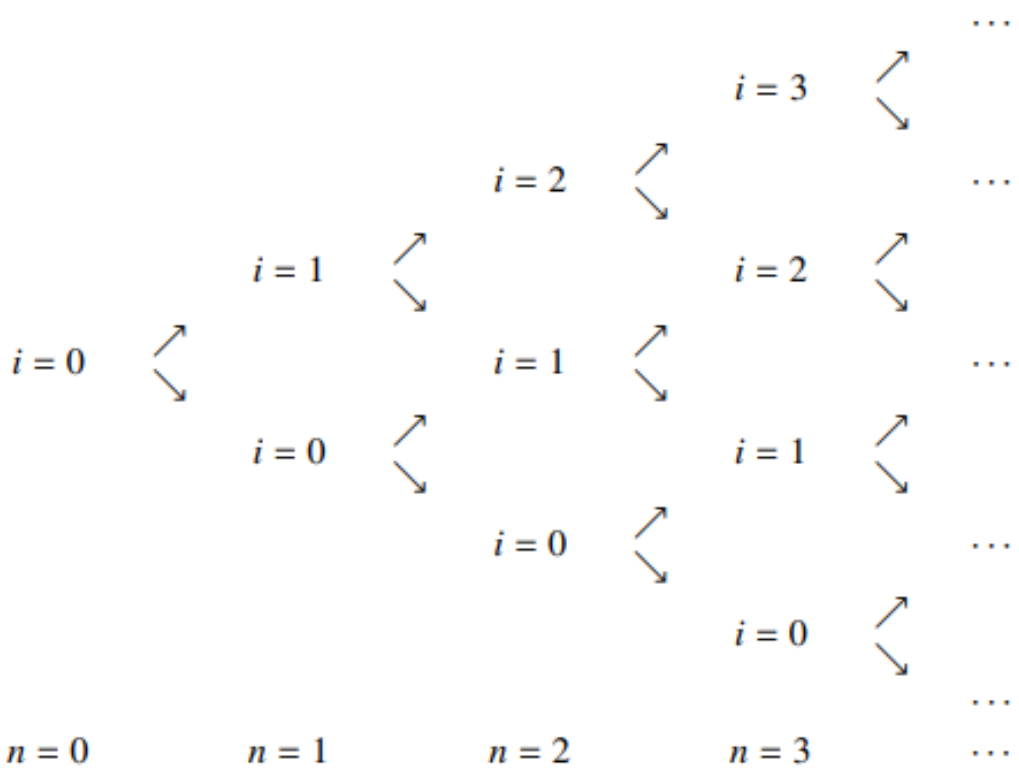

Figure: Binomial tree node structure. Price of stock in node (n, i) is $S(n,i) = S(0)(1+U)^i(1+D)^{(n-i)}$. Here $n = 0, 1, \ldots$ and $i = 0, 1, \ldots, n$.

American Options

Options07.h

```
#ifndef Options07_h
#define Options07_h

#include "BinModel02.h"

class EurOption
{
   private:
      int N; //steps to expiry
   public:
      void SetN(int N_){N=N_;}
      virtual double Payoff(double z)=0;
      //pricing European option
      double PriceByCRR(BinModel Model);
};

class AmOption
{
   private:
      int N; //steps to expiry
   public:
      void SetN(int N_){N=N_;}
      virtual double Payoff(double z)=0;
      //pricing American option
      double PriceBySnell(BinModel Model);
};
```

American Options

Options07.h (cont.)

```
Class Call: public EurOption, public AmOption
{
   private:
      double K; //strike price
   public:
      void SetK(double K_){K=K_;}
      int GetInputData();
      double Payoff(double z);
};

class Put: public EurOption, public AmOption
{
   private:
      double K; //strike price
   public:
      void SetK(double K_){K=K_;}
      int GetInputData();
      double Payoff(double z);
};

#endif
```

American Options

Options07.cpp

```
#include "Options07.h"
#include "BinModel02.h"
#include <iostream>
#include <cmath>
#include <vector>
using namespace std;

double EurOption::PriceByCRR(BinModel Model)
{
    double q=Model.RiskNeutProb();
    vector<double> Price(N+1);
    for (int i=0; i<=N; i++)
    {
        Price[i]=Payoff(Model.S(N,i));
    }
    for (int n=N-1; n>=0; n--)
    {
        for (int i=0; i<=n; i++)
        {
            Price[i]=(q*Price[i+1]+(1-q)*Price[i])
                /(1+Model.GetR());
        }
    }
    return Price[0];
}
```

American Options

Options07.cpp (cont. 1)

```
double AmOption::PriceBySnell(BinModel Model)
{
   double q=Model.RiskNeutProb();
   vector<double> Price(N+1);
   double ContVal;
   for (int i=0; i<=N; i++)
   {
      Price[i]=Payoff(Model.S(N,i));
   }
   for (int n=N-1; n>=0; n--)
   {
      for (int i=0; i<=n; i++)
      {
         ContVal=(q*Price[i+1]+(1-q)*Price[i])
                                  /(1+Model.GetR());
         Price[i]=Payoff(Model.S(n,i));
         if (ContVal>Price[i]) Price[i]=ContVal;
      }
   }
   return Price[0];
}
```

American Options

Options07.cpp (cont. 2)

```
int Call::GetInputData()
{
    cout << "Enter call option data:" << endl;
    int N;
    cout << "Enter steps to expiry N: "; cin >> N;
    EurOption::SetN(N); AmOption::SetN(N);
    cout << "Enter strike price K:    "; cin >> K;
    cout << endl;
    return 0;
}
```

American Options

Options07.cpp (cont. 3)

```
double Call::Payoff(double z)
{
   if (z > K) return z-K;
   return 0.0;
}

int Put::GetInputData()
{
   cout << "Enter put option data:" << endl;
   int N;
   cout << "Enter steps to expiry N: "; cin >> N;
   EurOption::SetN(N); AmOption::SetN(N);
   cout << "Enter strike price K:    "; cin >> K;
   cout << endl;
   return 0;
}

double Put::Payoff(double z)
{
   if (z < K) return K-z;
   return 0.0;
}
```

American Options

Main12.cpp

```
#include "BinModel02.h" #include "Options07.h"
#include <iostream> using namespace std;

int main()
{
    BinModel Model;

    if (Model.GetInputData()==1) return 1;

    Call Option1;
    Option1.GetInputData();
    cout << "European call option price = "
         << Option1.PriceByCRR(Model)
         << endl;
    cout << "American call option price = "
         << Option1.PriceBySnell(Model)
         << endl << endl;

    Put Option2;
    Option2.GetInputData();
    cout << "European put option price = "
         << Option2.PriceByCRR(Model)
         << endl;
    cout << "American put option price = "
         << Option2.PriceBySnell(Model)
         << endl << endl;

    return 0;
}
```

American Options

Options08.h

```
#ifndef Options08_h
#define Options08_h

#include "BinModel02.h"

class Option
{
   private:
      int N; //steps to expiry
   public:
      void SetN(int N_){N=N_;}
      int GetN(){return N;}
      virtual double Payoff(double z)=0;
};

class EurOption: public virtual Option
{
   public:
      //pricing European option
      double PriceByCRR(BinModel Model);
};

class AmOption: public virtual Option
{
   public:
      //pricing American option
      double PriceBySnell(BinModel Model);
};
```

American Options

Options08.h (cont. 1)

```
class Call: public EurOption, public AmOption
{
   private:
      double K; //strike price
   public:
      void SetK(double K_){K=K_;}
      int GetInputData();
      double Payoff(double z);
};

class Put: public EurOption, public AmOption
{
   private:
      double K; //strike price
   public:
      void SetK(double K_){K=K_;}
      int GetInputData();
      double Payoff(double z);
};

#endif
```

American Options

Options08.cpp

```
#include "Options08.h"
#include "BinModel02.h"
#include <iostream>
#include <cmath>
#include <vector>
using namespace std;

double EurOption::PriceByCRR(BinModel Model)
{
    double q=Model.RiskNeutProb();
    int N=GetN();
    vector<double> Price(N+1);
    for (int i=0; i<=N; i++)
    {
        Price[i]=Payoff(Model.S(N,i));
    }
    for (int n=N-1; n>=0; n--)
    {
        for (int i=0; i<=n; i++)
        {
            Price[i]=(q*Price[i+1]+(1-q)*Price[i])
                /(1+Model.GetR());
        }
    }
    return Price[0];
}
```

American Options

Options08.cpp (cont. 1)

```
double AmOption::PriceBySnell(BinModel Model)
{
    double q=Model.RiskNeutProb();
    int N=GetN();
    vector<double> Price(N+1);
    double ContVal;
    for (int i=0; i<=N; i++)
    {
        Price[i]=Payoff(Model.S(N,i));
    }
    for (int n=N-1; n>=0; n--)
    {
        for (int i=0; i<=n; i++)
        {
            ContVal=(q*Price[i+1]+(1-q)*Price[i])
                                    /(1+Model.GetR());
            Price[i]=Payoff(Model.S(n,i));
            if (ContVal>Price[i]) Price[i]=ContVal;
        }
    }
    return Price[0];
}
```

American Options

Options08.cpp (cont. 2)

```
int Call::GetInputData()
{
    cout << "Enter call option data:" << endl;
    int N;
    cout << "Enter steps to expiry N: "; cin >> N;
    SetN(N);
    cout << "Enter strike price K:    "; cin >> K;
    cout << endl;
    return 0;
}

double Call::Payoff(double z)
{
    if (z>K) return z-K;
    return 0.0;
}

int Put::GetInputData()
{
    cout << "Enter put option data:" << endl;
    int N;
    cout << "Enter steps to expiry N: "; cin >> N;
    SetN(N);
    cout << "Enter strike price K:    "; cin >> K;
    cout << endl;
    return 0;
}
```

American Options: Class templates

It is useful to store the values of the American option not only at the very first $(0,0)$ node, but also at intermediate (n,i) ones.
It is important to store also information about the exercise strategy, whether you should exercise immediately of proceed. This is characterized by the condition

$$(S(n,i)-K)^{+}>0 \quad . \tag{68}$$

We build two BinLattice type classes, one of type double and another type bool, to record when the immediate exercise is of advantage.

American Options

BinLattice01.h

```
#ifndef BinLattice01_h
#define BinLattice01_h #include <iostream>#include <iomanip>#include <vector>
using namespace std;

class BinLattice
{
   private:

      int N;
      vector< vector<double> > Lattice;

   public:

      void SetN(int N_)
      {
         N=N_;
         Lattice.resize(N+1);
         for(int n=0; n<=N; n++) Lattice[n].resize(n+1);
      }

      void SetNode(int n, int i, double x)
         {Lattice[n][i]=x;}

      double GetNode(int n, int i)
         {return Lattice[n][i];}
```

American Options

BinLattice01.h (cont.)

```
        void Display()
        {
            cout << setiosflags(ios::fixed)
                 << setprecision(3);
            for(int n=0; n<=N; n++)
            {
                for(int i=0; i<=n; i++)
                    cout << setw(7) << GetNode(n,i);
                cout << endl;
            }
            cout << endl;
        }
};

#endif
```

American Options: template BinLattice

BinLattice02.h

```
#ifndef BinLattice02_h
#define BinLattice02_h
#include here!...

template<typename Type> class BinLattice
{
   private:
      int N;
      vector< vector<Type> > Lattice;

   public:
      void SetN(int N_)
      {

         N=N_;
         Lattice.resize(N+1);
         for(int n=0; n<=N; n++) Lattice[n].resize(n+1);
      }

      void SetNode(int n, int i, Type x)
         {Lattice[n][i]=x;}

      Type GetNode(int n, int i)
         {return Lattice[n][i];}
```

American Options: template BinLattice

BinLattice02.h (cont.)

```
        void Display()
        {
            cout << setiosflags(ios::fixed)
                 << setprecision(3);
            for(int n=0; n<=N; n++)
            {
                for(int i=0; i<=n; i++)
                    cout << setw(7) << GetNode(n,i);
                cout << endl;
            }
            cout << endl;
        }
};

#endif
```

American Options: template BinLattice

Options09.h

```
#ifndef Options09_h
#define Options09_h

#include "BinLattice02.h"
#include "BinModel02.h"

class Option
{
   private:
      int N; //steps to expiry

   public:
      void SetN(int N_){N=N_;}
      int GetN(){return N;}
      virtual double Payoff(double z)=0;
};

class EurOption: public virtual Option
{
   public:
      //pricing European option
      double PriceByCRR(BinModel Model);
};
```

American Options: template BinLattice

Options09.h (cont.)

```
class AmOption: public virtual Option
{
   public:
      //pricing American option
      double PriceBySnell(BinModel Model,
         BinLattice<double>& PriceTree,
         BinLattice<bool>& StoppingTree);
};

class Call: public EurOption, public AmOption
{
   private:
      double K; //strike price

   public:
      void SetK(double K_){K=K_;}
      int GetInputData();
      double Payoff(double z);
};
```

American Options: template BinLattice

```
Options09.h (cont.)

class Put: public EurOption, public AmOption
{
   private:
      double K; //strike price

   public:
      void SetK(double K_){K=K_;}
      int GetInputData();
      double Payoff(double z);
};

#endif
```

American Options: template BinLattice

Options09.cpp

```
#include "Options09.h" #include "BinModel02.h" #include "BinLattice02.h"
#include <iostream> #include <cmath> using namespace std;

double EurOption::PriceByCRR(BinModel Model)
{
    double q=Model.RiskNeutProb();
    int N=GetN();
    vector<double> Price(N+1);
    for (int i=0; i<=N; i++)
    {
        Price[i]=Payoff(Model.S(N,i));
    }
    for (int n=N-1; n>=0; n--)
    {
        for (int i=0; i<=n; i++)
        {
            Price[i]=(q*Price[i+1]+(1-q)*Price[i])
                /(1+Model.GetR());
        }
    }
    return Price[0];
}
```

American Options: template BinLattice

Options09.cpp (cont.)

```
double AmOption::PriceBySnell(BinModel Model,
   BinLattice<double>& PriceTree,
   BinLattice<bool>& StoppingTree)
{
   double q=Model.RiskNeutProb();
   int N=GetN();
   PriceTree.SetN(N);
   StoppingTree.SetN(N);
   double ContVal;
   for (int i=0; i<=N; i++)
   {
      PriceTree.SetNode(N,i,Payoff(Model.S(N,i)));
      StoppingTree.SetNode(N,i,1);
   }
   for (int n=N-1; n>=0; n--)
   {
      for (int i=0; i<=n; i++)
      {

         ContVal=(q*PriceTree.GetNode(n+1,i+1)
            +(1-q)*PriceTree.GetNode(n+1,i))
            /(1+Model.GetR());

         PriceTree.SetNode(n,i,Payoff(Model.S(n,i)));

         StoppingTree.SetNode(n,i,1);
```

American Options: template BinLattice

Options09.cpp (cont.)

```
            if (ContVal>PriceTree.GetNode(n,i))
            {
                PriceTree.SetNode(n,i,ContVal);
                StoppingTree.SetNode(n,i,0);
            }
            else if (PriceTree.GetNode(n,i)==0.0)
            {
                StoppingTree.SetNode(n,i,0);
            }
        }
    }
    return PriceTree.GetNode(0,0);
}

int Call::GetInputData()
{
    cout << "Enter call option data:" << endl;
    int N;
    cout << "Enter steps to expiry N: "; cin >> N;
    SetN(N);
    cout << "Enter strike price K:    "; cin >> K;
    cout << endl;
    return 0;
}
double Call::Payoff(double z)
{
    if (z > K) return z - K;
    return 0.0;
}
```

American Options: template BinLattice

Main14.cpp

```
#include "BinLattice02.h"
#include "BinModel02.h"
#include "Options09.h"
#include <iostream>
using namespace std;


int main()
{
    BinModel Model;

    if (Model.GetInputData()==1) return 1;

    Put Option;
    Option.GetInputData();
    BinLattice<double> PriceTree;
    BinLattice<bool> StoppingTree;
    Option.PriceBySnell(Model,PriceTree,StoppingTree);
    cout << "American put prices:" << endl << endl;
    PriceTree.Display();
    cout << "American put exercise policy:"
         << endl << endl;
    StoppingTree.Display();
    return 0;
}
```

American Options: template BinLattice

Exercise 3-1: Modify *PriceByCRR*() function and *Options09.cpp* to compute the replicating strategy for a European Option in the Binomial tree model, using the *BinLattice<>* class template to store stock and money market positions in the replicating strategy at the nodes of the binomial tree. The portfolio belonging to the replicating strategy created at time $n-1$, node i and held during the nth step, that is, until time n, consists of stock and money market account positions:

$$\Delta(n,i) = \frac{H(n+1,i+1) - H(n+1,i)}{S(n+1,i+1) - S(n+1,i)} \quad , \tag{69}$$

and the cash position of

$$B(n,i) = \frac{H(n+1,i) - \Delta(n,i)S(n+1,i)}{(1+R)} \quad . \tag{70}$$

At the very last time nodes replace $\Delta(N,i)$ with 1 for the in-the-money stock price, and 0 for the out of the money stock

American Options: American and European Barrier/Knock-Out Option

Exercise 3-2: Price a European and American knock-out (knock-in) option. Based on reflection principle the American Option will be double the price of the European.
Take strike 115, knock-out any time up to expiry. Take expiry to be 1y. You receive GBP1,- if you cross above, or GBP1,- if you do not cross above.
Check also the hedging with Call-Spreads and the Greeks.

Lecture 4

Path Dependent Options - Monte Carlo Methods

Lecture 4: Monte Carlo

In finance a large number of products need to be priced with Monte Carlo methods. The binomial model and PDE methods are not always able to capture the various features of the payoffs. Even less so analytical methods.
One example of a model where MC methods are used are path-dependent options like the Arithmetic Asian options. (The Geometric case below is only used as an example here to illustrate the methodology, since this option can be priced analytically).
As a simple example we take a stock on the Black-Scholes world:

$$S(t_k) = S(0)e^{\sigma W(t_k) - \frac{1}{2}\sigma^2 t_k} \tag{71}$$

An Asian option is one where the stock price is looked at a number of dates t_1, $t_2, \ldots,$ t_m, the so-called "observation points". Usually these are taken to be equidistant, of the form $t_k = \frac{k}{m}T$, say daily, monthly or annually.

Lecture 4: Monte Carlo

The Brownian motion has independent increments of the form $W(t) - W(s) \sim N(0, t-s)$ from one time point to the next. We call these independent samples of random variables, and take out the time distance, to re-scale them to standard Gaussian type distributions, $Z_1, Z_2, \ldots, Z_m$ and define

$$S(t_1) = S(0)e^{\sigma\sqrt{t_1}Z_1 - \frac{1}{2}\sigma^2 t_1} \tag{72}$$

$$S(t_k) = S(t_{k-1})e^{\sigma\sqrt{t_k - t_{k-1}}Z_k - \frac{1}{2}\sigma^2(t_k - t_{k-1})} \quad , \quad k = 1, \ldots, m, \tag{73}$$

where $Z_k \sim N(0,1)$, with no correlation between the different k's.

Lecture 4: Monte Carlo

BinModel:

$$S(n,i) = S_0(1+U)^i(1+D)^{n-i} \quad , \tag{74}$$

Black-Scholes model:

$$S(t_i) = S_0 \exp\left\{\sigma\sqrt{t_i}\tilde{Z}_i - \frac{1}{2}\sigma^2 t_i\right\} \quad . \tag{75}$$

We have here

$$S(t_0), \quad S(t_1),\ S(t_2),\ \ldots S(t_i), \ldots \quad S(t_m) = S(T) \tag{76}$$

Lecture 4: Monte Carlo

However, these random numbers $\tilde{Z}_i$, that take each S_i from S_0 to $S(t_i)$, are correlated as these Brownian motions extend in the intervals $(0, t_{i-1})$ and $(0, t_i)$ with overlap in the interval $(0, t_{i-1})$.
We prefer to convert to the following

$$S(t_i) = S_{i-1} \exp\left\{\sigma\sqrt{t_i - t_{i-1}}Z_i - \frac{1}{2}\sigma^2(t_i - t_{i-1})\right\} \quad , \qquad (77)$$

where the Z_i's are uncorrelated. We will use this equation in the path generation.

Lecture 4: Monte Carlo

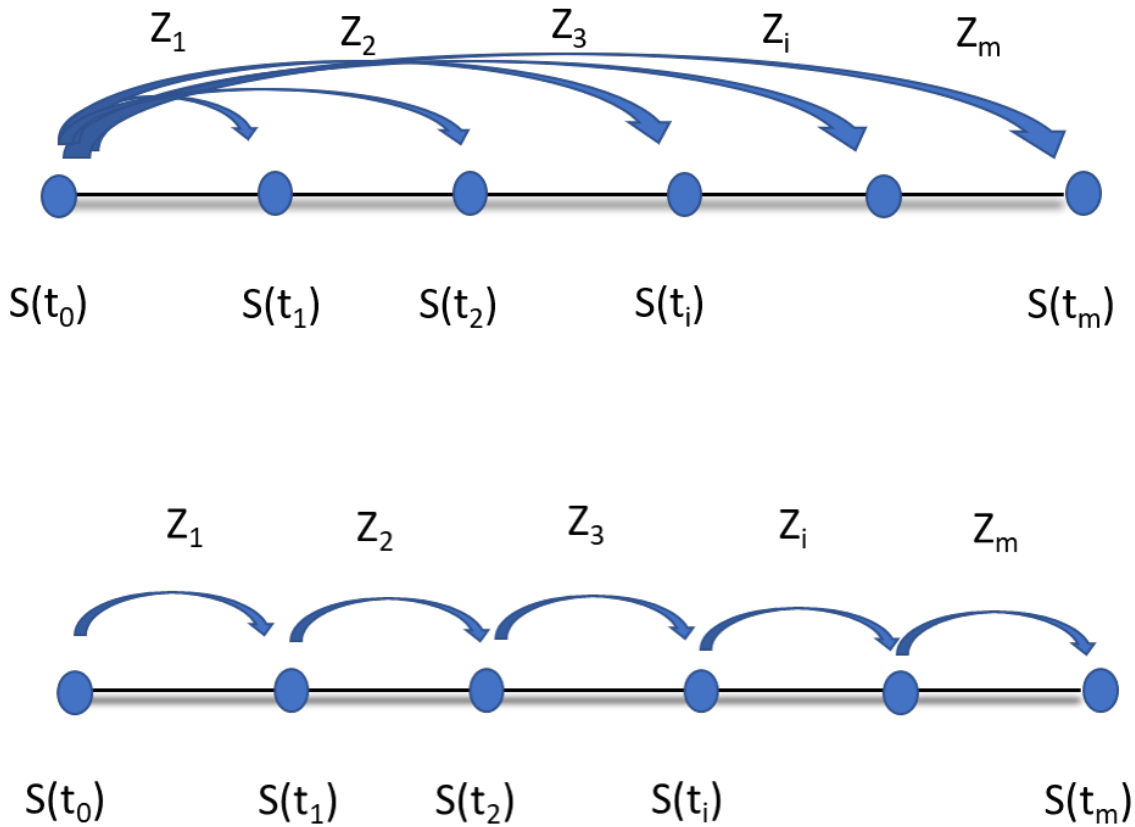

Figure: Two kinds of jumps from the initial position of stock price. The first, is done with correlated random variables, t he second with independent ones, different in each time interval.

Lecture 4: Monte Carlo

How do we select the standard Gaussian Numbers in C++?

in Excel we use =Norm.Inv(Rand(), Mean, StdDev)

Rand() produces uniform random variables in the interval $(0, 1)$.

Aletrnatively:
=SQRT(-2*LN(RAND()))*COS(2*PI()*RAND())*StdDev+Mean

where U1= Rand(), and U2=Rand().

Lecture 4: Monte Carlo

How do we get independent standard Gaussian random variable?
It is known that if I have two uniformly distributed random variables, U_1 and U_2 in the interval $(0, 1)$, then we can build two standard Gaussian random variables in turn, if i combine U_1 and U_2 as follows

$$Z = \cos 2\pi U_1 \sqrt{-2 \ln U_2} \quad , \quad \hat{Z} = \sin 2\pi U_1 \sqrt{-2 \ln U_2} \quad . \tag{78}$$

We need to avoid using values zero for U_1 and U_2. C++ provides the *rand*() function which is uniformly distributed between $(0, RAND\_MAX)$. We build

$$U_1 = \frac{rand() + 1}{RAND\_MAX + 1} \quad , \quad U_2 = \frac{rand() + 1}{RAND\_MAX + 1} \quad . \tag{79}$$

the 1 in the numerator is added to avoid the zero, and the 1 in the denominator is there to restrict the maximum value to 1.

Monte Carlo methods

```
double Gauss()
{
    double U1 = (rand()+1.0)/(RAND_MAX+1.0);

    double U2 = (rand()+1.0)/(RAND_MAX+1.0);



    return sqrt(-2.0*log(U1)) * cos(2.0*pi*U2);
}
```

As simple as this! This will be incorporated inside the new BS Monte-Carlo model which will replace the BinModel Class.

Lecture 4: Monte Carlo

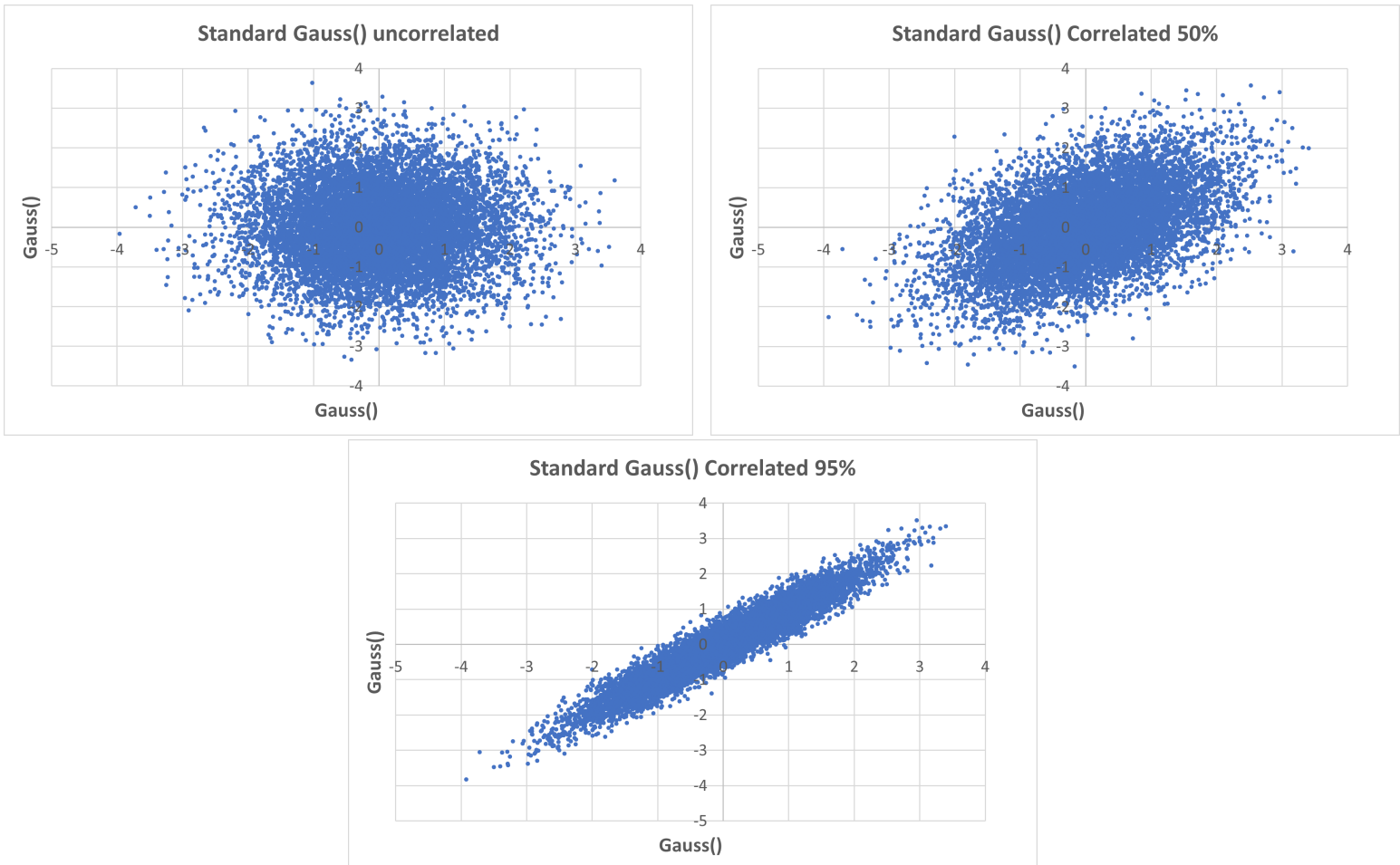

Figure: Example of uncorrelated, $\rho = 50\%$, $\rho = 95\%$, random numbers drawn from the Gauss() function of the previous page, is shown.

Lecture 4: Monte Carlo

The Asian options payoff is of the following form:

$$h^{Asian}(S_1, \ldots, S_m) = \left(\frac{1}{m} \sum_{k=1}^{m} S_k - K \right)^+ \tag{80}$$

The random numbers are obtained using the Box-Mueller method.

Lecture 4: Monte Carlo

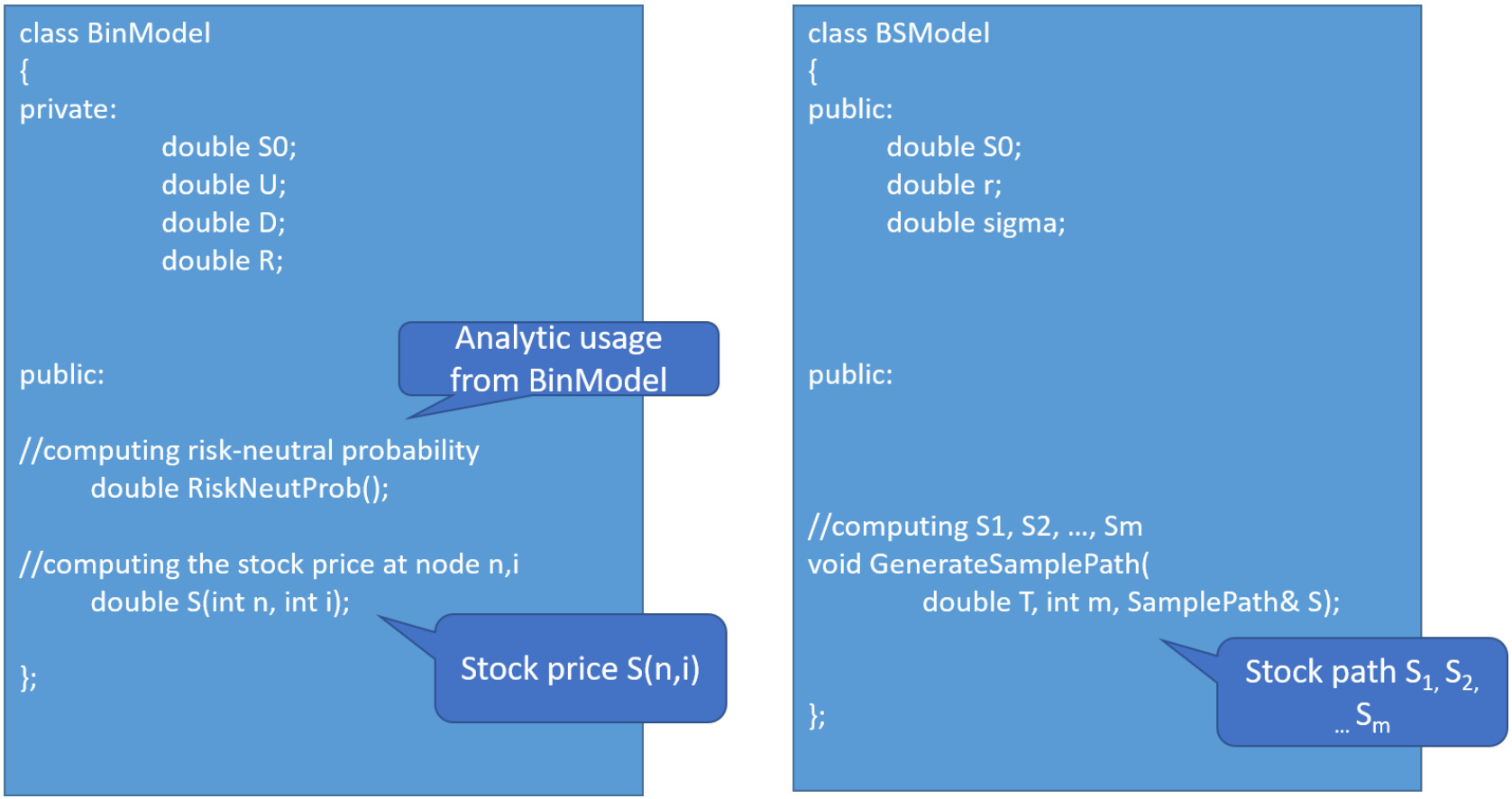

Figure: Compare the semi-analytic BinModel class with the MC BSModel class. Both need to produce the stock price along the way, nodes (n, i) and times t_i, $S_1, S_2, \ldots S(t_m)$, respectively.

Monte Carlo methods

BSModel01.h

```
#ifndef BSModel01_h
#define BSModel01_h

#include <vector> #include <cstdlib>
#include <ctime>  using namespace std;

typedef vector<double> SamplePath;

class BSModel
{
   public:

      double S0;
      double r;
      double sigma;

      BSModel(double S0_, double r_, double sigma_)
      :                                                    //constructor!
      {S0 = S0_; r = r_; sigma = sigma_; srand(time(NULL));}

      void GenerateSamplePath(double T, int m, SamplePath& S);
};

#endif
```

Monte Carlo methods

BSModel01.cpp

```
#include "BSModel01.h"
#include <cmath>

const double pi=4.0*atan(1.0);

double Gauss()
{
    double U1 = (rand()+1.0)/(RAND_MAX+1.0);
    double U2 = (rand()+1.0)/(RAND_MAX+1.0);

    return sqrt(-2.0*log(U1)) * cos(2.0*pi*U2);
}

void BSModel::GenerateSamplePath
              (double T, int m, SamplePath& S)
{
   double St = S0;
   for(int k=0; k<m; k++)
   {
      S[k]= St*exp((r-sigma*sigma*0.5)*(T/m)
                   +sigma*sqrt(T/m)*Gauss());
      St=S[k];
   }
}
```

vector<double> S has been provided to the GenerateSamplePath(), and it fills it up with the values of the Stock price along the path.

Lecture 4: Monte Carlo

Now we need a class design for the Arithmetic Asian Option. We need to build a base class which contains the pricing function for a generic path-dependent payoff, that takes as input instead of a stock price, a vector of doubles of stock prices, $S_1, S_2, \ldots, S_m$.
This function is defined as virtual and doesn't really do anything, apart from designing the base class, in preparation for the extension.
The exact shape of the function will be defined in the sub-classes, like the Arithmetic, the Geometric options, etc.

Monte Carlo methods

```
PathDependentOption01.h

#ifndef PathDepOption01_h
#define PathDepOption01_h

#include "BSModel01.h"

class PathDepOption
{
    public:
        double T;
        int m;

        double PriceByMC(BSModel Model, long N);

        virtual double Payoff(SamplePath& S)=0;
};
```

The base class needs the number of points, m, so it knows how many elements along the path it needs to use in the pricing. It also needs the final payoff point T, to do the discounting.

N in the PriceByMC(Model, N), is the number of the Monte Carlo simulations.

The Payoff($S_1, S_2, \ldots, S_m$) will be needed inside the PriceByMC(Model, N) function.

Monte Carlo methods

The ArithmAsianCall class will be inherited from the base class. It will contain the new element K, the strike.

```
PathDependentOption01.h (cont.)

class ArthmAsianCall: public PathDepOption
{
   public:
      double K;

      ArthmAsianCall(double T_, double K_, int m_)
            :                                          //constructor!
            {T=T_; K=K_; m=m_;}

      double Payoff(SamplePath& S);
};

#endif
```

Monte Carlo methods

PathDependentOption01.cpp

```
#include "PathDepOption01.h"
#include <cmath>

double PathDepOption::PriceByMC(BSModel Model, long N)
{
    double H=0.0;
    SamplePath S(m);  // you need to supply the size of the vector of doubles here!

    for(long i=0; i < N; i++)
    {
        Model.GenerateSamplePath(T,m,S);
        H += Payoff(S);
    }

    return exp(-Model.r*T)*H / N ;
}

double ArthmAsianCall::Payoff(SamplePath& S)
{
    double Ave = 0.0;

    for (int k = 0; k < m; k++) Ave = (k*Ave+S[k])/(k+1.0);

    if (Ave < K) return 0.0;

    return Ave - K;
}
```

Monte Carlo methods

Main.cpp

```
#include <iostream>
#include "PathDepOption01.h"

using namespace std;

int main()
{
    double S0=100.0, r=0.03, sigma=0.2;
    BSModel Model(S0,r,sigma);

    double T=1.0/12.0, K=100.0; // Expiry is 1 month.

    int m=30;                   // Daily observations for one month!

    ArthmAsianCall Option(T,K,m);

    long N=30000;

    cout << "Asian Call Price = "
         << Option.PriceByMC(Model,N) << endl;

    return 0;
}
```

Monte Carlo: Barrier Options - Homework

Exercise 1: Extend the code to calculate European Call and Put Options.

Exercise 2: Generate paths for the Black-Scholes equation, the Vasicek model (or Ornstein-Uhlenbeck) process, and the Hull-White process, perhaps its bonds. Plot distributions of the process, the variance etc.

Monte Carlo methods - Pricing Error

If we assume N Monte-Carlo paths, for path i the payoff will be

$$H^i(T) = h(S^i(t_1), \ldots S^i(t_m)), \quad \text{for } i = 1, \ldots, N \quad . \tag{81}$$

Each discounted payoff, $e^{-rT} H^i(T)$ differs from the option price $H(0)$, by the amount $e^{-rT} H^i(T) - H_N(0)$. These differences, if plotted, will produce a distribution. (Draw this distribution as homework!).
The standard error of the MC calculation would be obtained by

$$\sigma_N = \sqrt{\frac{1}{N(N-1)} \sum_{i=1}^{N} \left(e^{-rT} H^i(T) - H_N(0)\right)^2} \quad , \tag{82}$$

$$= \frac{e^{-rT}}{\sqrt{N-1}} \sqrt{\frac{1}{N} \sum_{i=1}^{N} \left(H^i(T)^2 - e^{rT} H_N(0)\right)^2} \quad , \tag{83}$$

where

$$e^{rT} H_N(0) = \frac{1}{N} \sum_{i=1}^{N} H^i(T) \quad . \tag{84}$$

Monte Carlo methods - Pricing Error

$$\frac{1}{N}\sum_{i=1}^{N}\left(H^i(T)^2 - e^{rT}H_N(0)\right)^2 = \frac{1}{N}\sum_{i=1}^{N}\left(H^i(T)\right)^2 - \left(e^{rT}H_N(0)\right)^2 \quad (86)$$

$$\frac{1}{N}\sum_{i=1}^{N}\left(H^i(T)^2 - e^{rT}H_N(0)\right)^2 = \frac{1}{N}\sum_{i=1}^{N}\left(H^i(T)\right)^2 - \left(\frac{1}{N}\sum_{i=1}^{N}H^i(T)\right)^2 \quad (87)$$

The equations to the right represent the expected values $E[H(T)^2] - E[H(T)]^2$. Therefore the standard deviation, of the error distribution function is given be:

$$\sigma_N = \frac{e^{-rT}}{\sqrt{N-1}}\sqrt{E[H(T)^2] - E[H(T)]^2} = \frac{e^{-rT}}{\sqrt{N-1}}\sqrt{\mathrm{Var}[H(T)]} \quad . \quad (88)$$

Monte Carlo methods - Pricing Error

PathDepOption02.h

```
#ifndef PathDepOption02_h
#define PathDepOption02_h

#include "BSModel01.h"

class PathDepOption
{
    public:
        double T, Price, PricingError;    // <- Price and PricingError added here!
        int m;
        double PriceByMC(BSModel Model, long N);
        virtual double Payoff(SamplePath& S)=0;
};

class ArthmAsianCall: public PathDepOption
{
    public:
        double K;
        ArthmAsianCall(double T_, double K_, int m_)
              {T=T_; K=K_; m=m_;}
        double Payoff(SamplePath& S);
};

#endif
```

Monte Carlo methods - Pricing Error

PathDepOption02.cpp

```
#include "PathDepOption02.h"
#include <cmath>

double PathDepOption::PriceByMC(BSModel Model, long N)
{
    double H=0.0, Hsq=0.0;
    SamplePath S(m);
    for(long i=0; i < N; i++)
    {
        Model.GenerateSamplePath(T,m,S);
        H   += Payoff(S);
        Hsq += pow(Payoff(S),2);
    }
    H   = H / N;
    Hsq = Hsq / N;
    Price = exp( - Model.r * T) * H;
    PricingError = exp( - Model.r * T) * sqrt( Hsq - H * H) / sqrt(N-1.0);
    return Price;
}

double ArthmAsianCall::Payoff(SamplePath& S)
{
    double Ave=0.0;
    for (int k=0; k < m; k++) Ave += S[k];
    if (Ave < K) return 0.0;
    return Ave / N - K;
}
```

Monte Carlo methods - Pricing Error - Greeks

We compute here Greeks in the Monte-Carlo implementation. We show here how to implement delta, and the rest to be done as exercises. Notice that

$$H(0) = u(S(0)) \quad . \tag{89}$$

Assuming $u(z)$ is differentiable, we define the Greek parameter δ as

$$\delta = \frac{\mathrm{d}u}{\mathrm{d}z}(S(0)) \quad . \tag{90}$$

To compute delta, for sufficiently small ϵ

$$\delta = \frac{\mathrm{d}u(1+\epsilon)S(0)) - u(S(0))}{\epsilon S(0)} \quad . \tag{91}$$

Monte Carlo methods - Pricing Error - Greeks

Since the price is calculated numerically as follows:

$$\begin{aligned} u(1+\epsilon)S(0)) &= e^{-rT} E^Q \left[h(1+\epsilon))(S(t_1), S(t_2), \ldots, S(t_m))\right] \quad (92) \\ &\approx e^{-rT} \frac{1}{N} \sum_{i=1}^{N} h(1+\epsilon))(S^i(t_1), S^i(t_2), \ldots, S^i(t_m)) \\ &= \hat{H}_{\epsilon N}(0) \quad (93) \end{aligned}$$

We can approximate δ using

$$\delta \approx \hat{\delta} = \frac{\hat{H}_{\epsilon N}(0) - \hat{H}_N(0)}{\epsilon S(0)} \quad (94)$$

See the changes to the "PathDepOption03.h" and ".cpp" and in particular the rescaling function

Monte Carlo methods - Pricing Error - Greeks

PathDepOption03.h

```
#ifndef PathDepOption03_h
#define PathDepOption03_h

#include "BSModel01.h"

class PathDepOption
{
    public:
        double T, Price, PricingError, delta;
        int m;
        virtual double Payoff(SamplePath& S)=0;
        double PriceByMC(BSModel Model, long N,
                                    double epsilon);
};

class ArthmAsianCall: public PathDepOption
{
    public:
        double K;
        ArthmAsianCall
              (double T_, double K_, int m_)
              {T=T_; K=K_; m=m_;}
        double Payoff(SamplePath& S);
};

#endif
```

Monte Carlo methods - Pricing Error

PathDepOption03.cpp (cont.)

```
#include "PathDepOption03.h"
#include <cmath>

void Rescale(SamplePath& S, double x)
{
    int m=S.size();
    for (int j=0; j<m; j++) S[j] = x*S[j];
}

double PathDepOption::PriceByMC(BSModel Model, long N,
                                double epsilon)
{
    double H=0.0, Hsq=0.0, Heps=0.0;
    SamplePath S(m);
    for(long i=0; i<N; i++)
    {
        Model.GenerateSamplePath(T,m,S);
        H = (i*H + Payoff(S))/(i+1.0);
        Hsq = (i*Hsq + pow(Payoff(S),2.0))/(i+1.0);
        Rescale(S,1.0+epsilon);
        Heps = (i*Heps + Payoff(S))/(i+1.0);
    }
    Price = exp(-Model.r*T)*H;
    PricingError = exp(-Model.r*T)*sqrt(Hsq-H*H)/sqrt(N-1.0);
    delta = exp(-Model.r*T)*(Heps-H)/(Model.S0*epsilon);
    return Price;
}
```

Monte Carlo methods - Pricing Error

PathDepOption03.cpp (cont.)

```
double ArthmAsianCall::Payoff(SamplePath& S)
{
    double Ave=0.0;
    for (int k=0; k<m; k++) Ave=(k*Ave+S[k])/(k+1.0);
    if (Ave<K) return 0.0;
    return Ave-K;
}
```

Monte Carlo: Barrier Options - Homework

Exercise 2: Using the fact that

$$\frac{\mathrm{d}^2 u}{\mathrm{d}z^2}(S(0)) \approx \frac{u((1.0+\epsilon)S(0)) - 2u(S(0)) + u((1.0-\epsilon)S(0))}{(\epsilon S(0))^2} \tag{95}$$

expand the code to compute the Greek parameter $\gamma = \frac{\mathrm{d}^2 u}{\mathrm{d}z^2}(S(0))$.
Exercise 3: Expand the code to compute the Greek parameters, Vega, theta, and rho.

Path Dependent Options - Monte Carlo Methods: Variance Reduction Technique

Lecture 4: Monte Carlo

An interesting way to reduce the Monte-Carlo error in numerical simulations is the Variance reduction technique.
Let us assume that we aim to calculate the expected value of variable X but we can calculate analytically the expected value of another variable Y, closely correlated with X. The way we can take advantage of this is by using the following equality

$$E\Big(X\Big) = E\Big(X - Y\Big) + E\Big(Y\Big) \quad . \tag{96}$$

What is the benefit of this? The benefit of this is that the two variables are very close and very correlated, thus if an error is introduced on each path in the variable X, a similar error is introduced in the correlated variable Y. Their difference inside the expectation helps cancel the numerical error.
As an example look into the Taylor expansion as follows:

$$\cos Z \approx 1 - \frac{1}{2}Z^2 \tag{97}$$

$\cos Z$ and $Z^2/2$ are correlated, thus $Y = 1 - \frac{1}{2}Z^2$ can serve as control

Lecture 4: Monte Carlo

Homework: Calculate the correlation between these two variables!

We will use

$$G(T) = g(S(t_1), S(t_2), \ldots, S(t_n)) \tag{98}$$

as control variate for

$$H(T) = h(S(t_1), S(t_2), \ldots, S(t_n)) \tag{99}$$

The $G(0)$ can be calculated analytically, and express $H(0)$ as follows:

$$H(0) = e^{-rT} E^Q [H(T) - G(T)] + G(0) \quad . \tag{100}$$

Lecture 4: Monte Carlo

As an example we will use the Arithmetic Asian option and the Geometric Asian option as its control variate. The geometric Asian call can be calculated analytically

$$g(z_1, z_2, \ldots, z_m) = \left(\sqrt[m]{\prod_{k=1}^{m} z_k} - K \right)^+ \tag{101}$$

It turns out the geometric Asian option payoff can be written as follows:

$$G(T) = \left(a e^{\left(r - \frac{1}{2}b^2\right)T + b\sqrt{T}Z} - K \right)^+ \quad , \tag{102}$$

where $Z \sim N(0,1)$, and the constants a and b are

$$a = e^{-rT} S(0) \exp\left\{ \frac{(m+1)T}{2m} \left(r + \frac{\sigma^2}{2} \left(\frac{(2m+1)}{3m} - 1 \right) \right) \right\} \quad , \tag{103}$$

$$b = \sigma \sqrt{\frac{(m+1)(2m+1)}{6m^2}} \quad . \tag{104}$$

Lecture 4: Monte Carlo

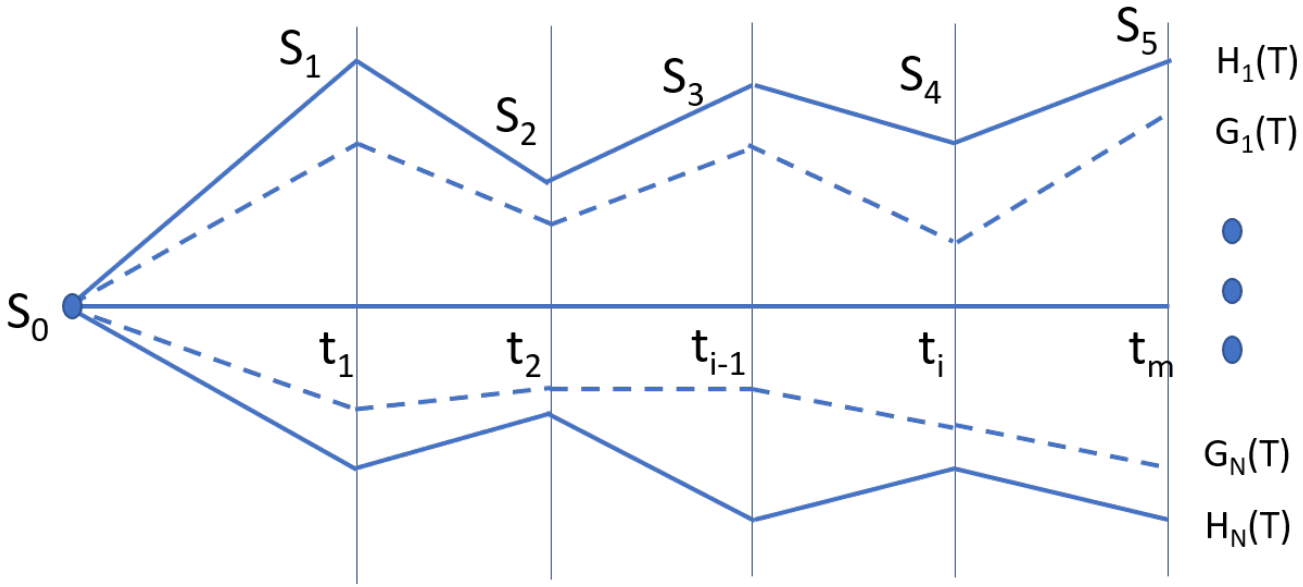

Figure: The paths followed by two correlated variables, X and Y, respectively. They follow each other closely. Errors of one cancels the error of the other. The first produces $H_i(T)$ and the second produces $G_i(T)$.

Lecture 4: Monte Carlo

Let us look into the product:

$$S_1 S_2 \cdots S_m = \prod_{k=1}^{m} \exp\left\{\left(r - \frac{1}{2}\sigma^2\right) t_k + \sigma W(t_k)\right\} \tag{105}$$

Here we group together

$$\sum_{k=1}^{m} \left(r - \frac{1}{2}\sigma^2\right) t_k = \left(r - \frac{1}{2}\sigma^2\right) \frac{T}{m} (1 + 2 + \ldots + m) \tag{106}$$

$$= \left(r - \frac{1}{2}\sigma^2\right) \frac{T}{m} \frac{m(m+1)}{2} \quad . \tag{107}$$

And then

$$\sum_{k=1}^{m} W(t_k) = mW(t_1) + (m-1)W(t_2 - t_1) + \ldots + W(t_m - t_{m-1}) \tag{108}$$

where the Brownian motions are all independent.

Lecture 4: Monte Carlo

Here I would like to remind the following equality about the sum of independent standard Gaussian random variables $X_k \sim N(\mu_k, \sigma_k^2)$

$$\sum_{k=1}^{m} \alpha_k X_k = N\left(\sum_{k=1}^{m} \alpha_k \mu_k, \sum_{k=1}^{m} \alpha_k^2 \sigma_k^2\right) \tag{109}$$

This results in

$$\sum_{k=1}^{m} W(t_k) = N\left(0, \frac{T}{m} \sum_{k=1}^{m} k^2\right) \quad . \tag{110}$$

Now use

$$\sum_{k=1}^{m} \left[(1+k)^3 - k^3\right] = (m+1)^3 - 1 \tag{111}$$

to get

$$\sum_{k=1}^{m} k^2 = \frac{m(m+1)(2m+1)}{6} \tag{112}$$

Lecture 4: Monte Carlo

Resulting in

$$\sigma \sum_{k=1}^{m} W(t_k) = \sigma\sqrt{T}\sqrt{\frac{(m+1)(2m+1)}{6}}N(0,1) \quad . \tag{113}$$

Putting these together we get the results of Eqs. (103) and (104).

Lecture 4: Monte Carlo

Notice that the above equations have similarity with, or rather re-appears in the calculation of the bond price in the Short Rate modeling. There we need the expectation of the exponential of the integral

$$\int_t^T r(s)\mathrm{d}s = r(t)(T-t) + \frac{1}{2}\theta(T-t)^2 + \sigma\int_t^T W(s)\mathrm{d}s - \sigma W(t)(T-t) \quad (114)$$

The integral $\int_t^T W(s)\mathrm{d}s$ appears and needs be calculated.
In similar fashion one can split the sum of the Brownian motions into sum of independent Brownian motions, not unlike the above calculations, getting the following

$$\int_t^T W(s)\mathrm{d}s = (T-t)W(t) + \int_t^T (T-s)\mathrm{d}W(s) \quad , \qquad (115)$$

the first term above being the rectangular area under the path, with height of $W(t)$ and width $(T-t)$. The rest is the area under the path above $W(t)$ point.

Lecture 4: Monte Carlo

The variance of the integral is indeed similar to the one of the Geometric Asian option. There a, couple of differences in the constants,

$$\int_t^T W(s)\mathrm{d}s = W(t_1)(t_2-t_1)+W(t_2)(t_3-t_2)+\ldots \tag{116}$$

$$= W(t_0)(T-t)+mW(t_1-t_0)(t_1-t_0)+(m-1)W(t_2-t_1)(t_2-t_1)+\ldots$$

Here we have the $\Delta t=(T-t)/m$ in front of the Brownians, which give $(\Delta t)^2=(T/m)^2$ in the calculation of the variance, (set $t=t_0=0$ for notation simplicity). If we use also the variance of $W(T_i-T_{i-1})=T/m$, then we get the total variance to be

$$\sigma^2\left(\frac{T}{m}\right)^2\left(\frac{T}{m}\right)\frac{m(m+1)(2m+1)}{6}\to\frac{\sigma^2T^3}{3} \tag{117}$$

This is useful in the case of the Geometric Asian option, as it keeps the variance linear in T. In the case of the short-rates, the exponent will be brought lower by introduction of the mean-reversion parameter.

Lecture 4: Monte Carlo

In the case of the Geometric Asian Eq. (113) we have two circumstances that lowers the exponent to linear. The first being the presence of the factor $(t_i - t_{i-1})$, which gives $(T/m)^2$ in the variance, and second, the presence of $1/m$ from the m-th root. What remains is

$$\sigma^2 \left(\frac{T}{m}\right) \frac{1}{m^2} \frac{m(m+1)(2m+1)}{6} \to \frac{\sigma^2 T}{3} \tag{118}$$

In the Geometric case a term T^2 is missing, as the m-th root contributes only with the factor $1/m$ not T/m.

Monte Carlo methods- Pricing Error

PathDepOption04.h

```
#ifndef PathDepOption04_h
#define PathDepOption04_h

#include "BSModel01.h"

class PathDepOption
{
    public:
        double T, Price, PricingError;
        int m;

        virtual double Payoff(SamplePath& S)=0;

        double PriceByMC(BSModel Model, long N);

        double PriceByVarRedMC
               (BSModel Model, long N, PathDepOption& CVOption);

        virtual double PriceByBSFormula
                       (BSModel Model){return 0.0;}
};
```

Monte Carlo methods- Pricing Error

PathDepOption04.h (cont.)

```
class DifferenceOfOptions: public PathDepOption
{
   public:
      PathDepOption* Ptr1;
      PathDepOption* Ptr2;
      DifferenceOfOptions(double T_, int m_,
                          PathDepOption* Ptr1_,
                          PathDepOption* Ptr2_)
         {T=T_; m=m_; Ptr1=Ptr1_; Ptr2=Ptr2_;}
      double Payoff(SamplePath& S)
         {return Ptr1->Payoff(S)-Ptr2->Payoff(S);}
};

class ArthmAsianCall: public PathDepOption
{
   public:
      double K;
      ArthmAsianCall(double T_, double K_, int m_)
            {T=T_; K=K_; m=m_;}
      double Payoff(SamplePath& S);
};

#endif
```

Monte Carlo methods- Pricing Error

PathDepOption04.cpp

```
#include "PathDepOption04.h"
#include <cmath>

double PathDepOption::PriceByMC(BSModel Model, long N)
{
    double H=0.0, Hsq=0.0;
    SamplePath S(m);
    for(long i=0; i<N; i++)
    {
        Model.GenerateSamplePath(T,m,S);
        H = (i*H + Payoff(S))/(i+1.0);
        Hsq = (i*Hsq + pow(Payoff(S),2.0))/(i+1.0);
    }
    Price = exp(-Model.r*T)*H;
    PricingError = exp(-Model.r*T)*sqrt(Hsq-H*H)/sqrt(N-1.0);
    return Price;
}

double PathDepOption::PriceByVarRedMC
        (BSModel Model, long N, PathDepOption& CVOption)
{
    DifferenceOfOptions VarRedOpt(T,m,this,&CVOption);

    Price = VarRedOpt.PriceByMC(Model,N)
          + CVOption.PriceByBSFormula(Model);

    PricingError = VarRedOpt.PricingError;

    return Price;
```

Monte Carlo methods- Pricing Error

PathDepOption04.cpp (cont.)

```
double ArthmAsianCall::Payoff(SamplePath& S)
{
   double Ave=0.0;
   for (int k=0; k<m; k++) Ave=(k*Ave+S[k])/(k+1.0);
   if (Ave<K) return 0.0;
   return Ave-K;
}
```

Monte Carlo methods- Pricing Error

GmtrAsianCall.h

```
#ifndef GmtrAsianCall_h
#define GmtrAsianCall_h

#include "PathDepOption04.h"

class GmtrAsianCall: public PathDepOption
{
   public:
      double K;
      GmtrAsianCall(double T_, double K_, int m_)
            {T=T_; K=K_; m=m_;}
      double Payoff(SamplePath& S);
      double PriceByBSFormula(BSModel Model);
};

#endif
```

Monte Carlo methods- Pricing Error

GmtrAsianCall.cpp

```
#include <cmath>
#include "GmtrAsianCall.h"
#include "EurCall.h"

double GmtrAsianCall::Payoff(SamplePath& S)
{
    double Prod=1.0;
    for (int i=0; i<m; i++)
    {
        Prod=Prod*S[i];
    }
    if (pow(Prod,1.0/m)<K) return 0.0;
    return pow(Prod,1.0/m)-K;
}

double GmtrAsianCall::PriceByBSFormula(BSModel Model)
{
    double a = exp(-Model.r*T)*Model.S0*exp(
                (m+1.0)*T/(2.0*m)*(Model.r
                +Model.sigma*Model.sigma
                 *((2.0*m+1.0)/(3.0*m)-1.0)/2.0));
    double b = Model.sigma
                    *sqrt((m+1.0)*(2.0*m+1.0)/(6.0*m*m));
    EurCall G(T, K);
    Price = G.PriceByBSFormula(a,b,Model.r);
    return Price;
}
```

Monte Carlo methods - Pricing Error

Main22.cpp

```
#include <iostream>
#include "PathDepOption04.h"
#include "GmtrAsianCall.h"

using namespace std;

int main()
{
    double S0=100.0, r=0.03, sigma=0.2;
    BSModel Model(S0,r,sigma);

    double T =1.0/12.0, K=100.0;
    int m=30;

    ArthmAsianCall Option(T,K,m);
    GmtrAsianCall  CVOption(T,K,m);

    long N=30000;
    Option.PriceByVarRedMC(Model,N,CVOption);
    cout << "Arithmetic call price = " << Option.Price << endl
         << "Error = " << Option.PricingError << endl;

    Option.PriceByMC(Model,N);
    cout << "Price by direct MC = " << Option.Price << endl
         << "MC Error = " << Option.PricingError << endl;

    return 0;
}
```

Monte Carlo methods - Pricing Error

EurCall.h

```
#ifndef EurCall_h
#define EurCall_h

class EurCall
{
   public:
      double T, K;

      EurCall(double T_, double K_){T=T_; K=K_;}

      double d_plus(double S0, double sigma, double r);

      double d_minus(double S0, double sigma, double r);

      double PriceByBSFormula(double S0,  double sigma, double r);
      double VegaByBSFormula(double S0,  double sigma, double r);
};

#endif
```

Monte Carlo methods - Pricing Error

EurCall.cpp

```
#include "EurCall.h"
#include <cmath>

double N(double x)
{
   double gamma = 0.2316419;      double a1 = 0.319381530;
   double a2    =-0.356563782;    double a3 = 1.781477937;
   double a4    =-1.821255978;    double a5 = 1.330274429;
   double pi    = 4.0*atan(1.0); double k  = 1.0/(1.0+gamma*x);
   if (x>=0.0)
   {
      return 1.0-((((a5*k+a4)*k+a3)*k+a2)*k+a1)
                  *k*exp(-x*x/2.0)/sqrt(2.0*pi);
   }
   else return 1.0-N(-x);
}

double EurCall::d_plus(double S0, double sigma, double r)
{
   return (log(S0/K)+
      (r+0.5*pow(sigma,2.0))*T)
      /(sigma*sqrt(T));
}

double EurCall::d_minus(double S0, double sigma, double r)
{
   return d_plus(S0,sigma,r)-sigma*sqrt(T);
}
```

Monte Carlo methods - Pricing Error

EurCall.cpp (cont.)

```
double EurCall::PriceByBSFormula(double S0,
    double sigma, double r)
{
    return S0*N(d_plus(S0,sigma,r))
        -K*exp(-r*T)*N(d_minus(S0,sigma,r));
}


double EurCall::VegaByBSFormula(double S0,
    double sigma, double r)
{
    double pi=4.0*atan(1.0);
    return S0*exp(-d_plus(S0,sigma,r)*d_plus(S0,sigma,r)/2)*sqrt(T)
        /sqrt(2.0*pi);
}
```

Monte Carlo: Barrier Options - Homework

Exercise 4: Price a barrier call option with payoff function

$$h^{\text{barrier call}}(S_1, \ldots, S_m) = \mathbf{1}_{\{\max_{k=1,\ldots,m} z_k \leq L\}} (S_m - K)^+ \quad . \tag{119}$$

Use the European Call option as Control Variate:

$$g(S_1, \ldots, S_m) = (S_m - K)^+ \quad . \tag{120}$$

Monte Carlo: Path-dependent basket options

Path Dependent Options - Monte Carlo Methods: Basket

Monte Carlo: Path-dependent basket options

We consider the case of d stocks,

$$\mathbf{S}(t) = \begin{pmatrix} S_1(t) \\ \vdots \\ S_d(t) \end{pmatrix} \quad . \tag{121}$$

The stock prices follow the risk-neutral dynamics

$$S_j(t) = S_j(0) \exp\left\{ \left(r - \frac{1}{2}\sigma_j^2 \right) t + \sum_{l=1}^{d} c_{jl} W_l(t) \right\} \quad . \tag{122}$$

The volatilities above are

$$\sigma_j = \sqrt{c_{j1}^2 + c_{j2}^2 + \ldots + c_{jd}^2} \quad , \quad \text{for } j = 1, 2, \ldots, d \quad . \tag{123}$$

For now we assume the volatilities are not time dependent.

Monte Carlo: Path-dependent basket options

We need now to evolve a vector of d stocks, from time t_k to time t_{k+1} for $k = 1, \ldots, m$

$$\begin{pmatrix} S_1(0) \\ \vdots \\ S_d(0) \end{pmatrix} \quad , \quad \begin{pmatrix} S_1(t_1) \\ \vdots \\ S_d(t_1) \end{pmatrix} \quad \cdots \quad \begin{pmatrix} S_1(t_m) \\ \vdots \\ S_d(t_m) \end{pmatrix} \quad . \tag{124}$$

We use the a set of d independent Brownian motions

$$\begin{pmatrix} W_1(0) \\ \vdots \\ W_d(0) \end{pmatrix} \quad , \quad \begin{pmatrix} W_1(t_1) \\ \vdots \\ W_d(t_1) \end{pmatrix} \quad \cdots \quad \begin{pmatrix} W_1(t_m) \\ \vdots \\ W_d(t_m) \end{pmatrix} \quad . \tag{125}$$

Each of the stocks uses the following combination of Brownian motions:

$$S_j(t) \sim c_{j1} W_1(t) + c_{j2} W_2(t) + c_{jd} W_d(t) \quad . \tag{126}$$

The stocks are correlated

$$\frac{\mathrm{d}S_j}{S_j} \frac{\mathrm{d}S_k}{S_k} = \sum_{l=1}^{d} c_{jl} c_{kl} \mathrm{d}t \tag{127}$$

Monte Carlo: Path-dependent basket options

An example of the payoff of arithmetic Asian call is the following:

$$H(T) = \left(\sum_{j=1}^{d} \left(\frac{1}{m} \sum_{k=1}^{m} S_j(t_k) \right) - K \right)^+ = \left(\sum_{j=1}^{d} \bar{S}_j - K \right)^+ \qquad (128)$$

Introduce the standard Gaussians $\mathbf{Z}$

$$\mathbf{Z} = \begin{pmatrix} Z_1 \\ \vdots \\ Z_d \end{pmatrix} \quad , \qquad (129)$$

In other words, to evolve the stocks we use a d-dimensional Gaussian Brownian motion, where each component is N(0,1)

$$\mathbf{Z}_1, \mathbf{Z}_2, \ldots \mathbf{Z}_m \quad , \qquad (130)$$

where each $\mathbf{Z}_k$, is a d-dimensional, as shown in Eq.(129).

Monte Carlo: Path-dependent basket options

In terms of notations we use the vector notation to compactify our presentation.

$$\mathbf{vw} = \begin{pmatrix} v_1 w_1 \\ \vdots \\ v_d w_d \end{pmatrix} \quad , \quad \sigma\sigma = \begin{pmatrix} \sigma_1 \sigma_1 \\ \vdots \\ \sigma_d \sigma_d \end{pmatrix} \quad , \quad \exp(\mathbf{v}) = \begin{pmatrix} e^{v_1} \\ \vdots \\ e^{v_d} \end{pmatrix} \quad , \quad (131)$$

The price vector can be written as

$$\mathbf{S}(t_k) = \mathbf{S}(t_{k-1}) \exp\left\{ \left(r - \frac{1}{2}\sigma\sigma \right) (t_k - t_{k-1}) + \sqrt{t_k - t_{k-1}} \mathbf{C}\mathbf{Z}_k \right\} \quad (132)$$

The volatility matrix $\mathbf{C}$ is a square $d \times d$ matrix

$$\mathbf{C}_{d\times d} = c_{ij} \quad , \quad i = 1, \ldots, d \quad , \quad j = 1, \ldots, d \quad . \quad (133)$$

To proceed with the codding we will need to set up the functions that will handle the matrix and vector operations

Monte Carlo methods- Pricing Error

Matrix.h

```
#ifndef Matrix_h
#define Matrix_h

#include <vector>
using namespace std;

typedef vector<double> Vector;
typedef vector<Vector> Matrix;

Vector operator*(const Matrix& C,const Vector& V);
Vector operator*(const double& a,const Vector& V);
Vector operator+(const double& a,const Vector& V);
Vector operator+(const Vector& V,const Vector& W);
Vector operator*(const Vector& V,const Vector& W);
Vector exp(const Vector& V);
double operator^(const Vector& V,const Vector& W);

#endif
```

Monte Carlo methods- Pricing Error

Matrix.cpp

```
#include "Matrix.h"
#include <cmath>

Vector operator*(const Matrix& C,const Vector& V)
{
   int d = C.size();
   Vector W(d);
   for (int j=0; j<d; j++)
   {
      W[j]=0.0;
      for (int I=0; I<d; I++) W[j]=W[j]+C[j][I]*V[I];
   }
   return W;
}

Vector operator+(const Vector& V,const Vector& W)
{
   int d = V.size();
   Vector U(d);
   for (int j=0; j<d; j++) U[j] = V[j] + W[j];
   return U;
}
Vector operator+(const double& a,const Vector& V)
{
   int d = V.size();
   Vector U(d);
   for (int j=0; j<d; j++) U[j] = a + V[j];
   return U;
}
```

Monte Carlo methods- Pricing Error

Matrix.cpp (cont.)

```
Vector operator*(const double& a,const Vector& V)
{
   int d = V.size();
   Vector U(d);
   for (int j=0; j<d; j++) U[j] = a*V[j];
   return U;
}
Vector operator*(const Vector& V,const Vector& W) // returns Vector, elements v[j]*w[j]
{
   int d = V.size();
   Vector U(d);
   for (int j=0; j<d; j++) U[j] = V[j] * W[j];
   return U;
}
Vector exp(const Vector& V)
{
   int d = V.size();
   Vector U(d);
   for (int j=0; j<d; j++) U[j] = exp(V[j]);
   return U;
}
double operator^(const Vector& V,const Vector& W)  //sums the products of the elements,
                                                   // returns double
{
   double sum = 0.0;
   int d = V.size();
   for (int j=0; j<d; j++) sum = sum + V[j]*W[j];
   return sum;
}
```

Monte Carlo methods- Pricing Error

BSModel02.h

```
#ifndef BSModel02_h
#define BSModel02_h

#include "Matrix.h"

typedef vector<Vector> SamplePath;

class BSModel
{
    public:
        Vector S0, sigma;
        Matrix C;
        double r;
        BSModel(Vector S0_, double r_, Matrix C_);
        void GenerateSamplePath(double T, int m, SamplePath& S);
};

#endif
```

Monte Carlo methods- Pricing Error

BSModel02.cpp

```
#include "BSModel02.h"
#include <cmath> #include <cstdlib> #include <ctime>

const double pi=4.0*atan(1.0);

double Gauss()
{
      double U1 = (rand()+1.0)/(RAND_MAX+1.0);
      double U2 = (rand()+1.0)/(RAND_MAX+1.0);
      return sqrt(-2.0*log(U1)) * cos(2.0*pi*U2);
}

Vector Gauss(int d)
{
   Vector Z(d);
   for (int j=0; j<d; j++) Z[j] = Gauss();
   return Z;
}

BSModel::BSModel(Vector S0_, double r_, Matrix C_)
{
   S0 = S0_; r = r_; C = C_; srand(time(NULL));
   int d = S0.size();
   sigma.resize(d);
   for (int j=0; j<d; j++) sigma[j] = sqrt(C[j] ^ C[j]);
}
```

Monte Carlo methods- Pricing Error

BSModel02.cpp (cont.)

```
void BSModel::GenerateSamplePath(double T, int m, SamplePath& S)
{
   Vector St = S0;
   int d = S0.size();
   for(int k=0; k<m; k++)
   {
      S[k]= St*exp((T/m)*(r+(-0.5)*sigma*sigma)
                  +sqrt(T/m)*(C*Gauss(d)));
      St=S[k];
   }
}
```

Monte Carlo methods- Pricing Error

PathDepOption05.h

```
#ifndef PathDepOption05_h
#define PathDepOption05_h

#include "BSModel02.h"

class PathDepOption
{
    public:
        double T;
        int m;
        double PriceByMC(BSModel Model, long N);
        virtual double Payoff(SamplePath& S)=0;
};

class ArthmAsianCall: public PathDepOption
{
    public:
        double K;
        ArthmAsianCall(double T_, double K_, int m_)
              {T=T_; K=K_; m=m_;}
        double Payoff(SamplePath& S);
};

#endif
```

Monte Carlo methods- Pricing Error

PathDepOption05.cpp

```
#include "PathDepOption05.h"
#include <cmath>

double PathDepOption::PriceByMC(BSModel Model, long N)
{
    double H=0.0;
    SamplePath S(m);
    for(long i=0; i<N; i++)
    {
        Model.GenerateSamplePath(T,m,S);
        H = (i*H + Payoff(S))/(i+1.0);
    }
    return exp(-Model.r*T)*H;
}

double ArthmAsianCall::Payoff(SamplePath& S)
{
    double Ave=0.0;
    int d=S[0].size();
    Vector one(d);
    for (int i=0; i<d; i++) one[i]=1.0;
    for (int k=0; k<m; k++)
    {
        Ave=(k*Ave+(one^S[k]))/(k+1.0);
    }
    if (Ave<K) return 0.0;
    return Ave-K;
}
```

Monte Carlo methods- Pricing Error

Main24.cpp

```
#include <iostream>
#include "PathDepOption05.h"
using namespace std;

int main()
{
   int d=3;
   Vector S0(d);
      S0[0]=40.0;
      S0[1]=60.0;
      S0[2]=100.0;
   double r=0.03;
   Matrix C(d);
   for (int i=0; i<d; i++) C[i].resize(d);
      C[0][0] =  0.1;  C[0][1] = -0.1;  C[0][2] = 0.0;
      C[1][0] = -0.1;  C[1][1] =  0.2;  C[1][2] = 0.0;
      C[2][0] =  0.0;  C[2][1] =  0.0;  C[2][2] = 0.3;
   BSModel Model(S0,r,C);

   double T=1.0/12.0, K=200.0;
   int m=30;
   ArthmAsianCall Option(T,K,m);

   long N=30000;
   cout << "Arithmetic Basket Call Price = "
        << Option.PriceByMC(Model,N) << endl;

   return 0;
```

Monte Carlo methods- Pricing Error - Homework

Exercise 5: Write a class for pricing a European basket call option with payoff

$$H(T) = \left(\sum_{j=1}^{d} S_j(T) - K \right)^+ \quad . \tag{134}$$

Use as control variate to reduce the MC error the following basket of european calls

$$G(T) = \sum_{j=1}^{d} (S_j(T) - K_j)^+ \quad , \quad K_j = K \frac{S_j(0)}{\sum_{j=1}^{d} S_j(0)} \quad . \tag{135}$$

The above is similar to pricing options on sums of Libors, a very well known type of trade on the fixed income derivatives

$$H(T) = \left(\sum_{j=1}^{d} \tau_j L_j(T_j) - K \right)^+ \quad , \tag{136}$$

where τ_j are accrual periods for each Libor L_j.

Monte Carlo methods- Pricing Error - Homework

Notice also that Swaps are given as follows:

$$Swap(T) = B(T, T_0) - B(T, T_n) - K \sum_{j=1}^{n} \tau_j B_J(T, T_j) = \sum_{j=0}^{n} c_j B_j(T, T_j) \quad , \quad (1$$

where you can extract the value of c_j.
Swaptions are simply options on the $Swap(T)$, with payoff at time T equal to

$$Swaption(T) = (Swap(T))^+ = \left(\sum_{j=0}^{n} c_j B_j(T, T_j) \right)^+ \qquad (138)$$