Roxana Dumitrescu

C++ in Financial Mathematics

What have we learnt?

- We have learnt that any class that you intend to use as a base class should contain a **virtual destructor**.
- Pure virtual functions
- Abstract classes and interface classes
- We have written a **Binomial pricer** which contains:
  - A class representing the model (the evolution of the asset price): **class BinomialModel**
  - An hierarchy of classes representing the financial contract (of *European and American type*)
- C++ Design Techniques: we have learnt about multiple inheritance and virtual inheritance.

Plan

- Binomial Pricer and Templates
- Pricing in the Black-Sholes model: Design in C++ and simulation by Monte Carlo.

Binomial pricer

Templates

Binomial pricer

Class templates

- **Aim** We would like to compute and to store the price of an **American** option not only at time 0, but also for each time step n and node i in the binomial tree. In addition, we want to compute the **early exercise policy** for an **American option**. The time steps n and noded i at which the option should be exercised are characterised by **the condition**

$$H(n,i) = h(S(n,i)) > 0.$$

Binomial pricer

Class templates

- The idea will be to encode this information as data of type **bool**, taking just two possible values, 0 is the option should not be exercised at a given node, or 1 otherwise, depending on whether the above condition is violated or not.
- The natural structure for the price data is that of a lattice indexed by time steps $n = 0, 1, ..., N$ and nodes $i = 0, 1, ..., n$.

Binomial pricer

Class templates

- The convenient way to store the option prices will be a vector indexed by time variable *n* consisting vectors of type **double** indexed by the nodes *i* at each time *n*. We'll thus have a vector of vectors whose declaration is the following

```
vector<vector<double>> Lattice;
```

We also provide related functionality such as setting the number of time steps *N* or setting and retrieving a value at time *n* and node *i*.

Binomial pricer

Class templates

```
class BinLattice
{
private:
    int N;
    vector<vector<double>> Lattice;
         public:
    void SetN(int N)
{
this->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;}
```

Binomial pricer

Class templates

```
   double GetNode(int n, int i) const
{return Lattice[n][i];}
   void Display() const
{
for (int n=0;n<=N;n++)
{
for (int i=0;i<=n;i++)
cout<<GetNode(n,i)<<" ";
}
cout<<endl;
}
}
```

Binomial pricer

Class templates

The **BinLattice** class is defined here.

The class contains two variables:

- N to store the number of time steps in the binomial tree
- **Lattice**, a vector of vectors to hold data of type double

The **BinLattice** class also contains the following definitions:

- The **SetN()** function takes a parameter of type **int**, assigns it to N, sets the size of the **Lattice** vector to $N + 1$, the number of time instants n from to 0 to N, and then for each n sets the size of the inner vector **Lattice[n]** to $n + 1$, the number of nodes at time n.

Binomial pricer

Class templates

- **SetNode()** to set the value stored at step n, node i.
- **GetNode()** to return the value stored at step n node i.
- **Display()** to print the values stored at step n, node i.

Binomial pricer

Class templates

How to record the stopping policy?

We remark that in order to record the stopping strategy we need exactly the same class BinLattice, but with data of type **bool** instead of **double**. However, we do not want several duplicate code!

Solution?

Class templates offer a much neater solution. The type is not hardwired inside the class, but passed to it as a parameter. We achieve this by modifying the **BinLattice** class as follows.

Binomial pricer

Class templates

How to record the stopping policy?

```
template<typename Type> class BinLattice
{ private:
    int N;
    vector<vector<Type>> Lattice;
         public:
    void SetN(int N)
{       this->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;}
```

Binomial pricer

Class templates

```
    Type GetNode(int n, int i) const {return
    Lattice[n][i];}
    void Display() const
{     for (int n=0;n<=N;n++)
{
for (int i=0;i<=n;i++)
cout<<GetNode(n,i)<<" ";
}
cout<<endl;
}
}
```

Binomial pricer

Which are the changes?

- **class BinLattice** is replaced by **template<typename Type> class BinLattice** which specifies that **BinLattice** is no longer a class, but a **class template** with **type parameter** Type. **Every occurence of double is replaced by Type**
- **vector<vector<double>> Lattice;** is replaced by **vector<vector<Type>> Lattice;**
- **void SetNode(int n, int i, double x)** is replaced by **void SetNode(int n, int i, Type x);**
- **double GetNode(int n, int i)** is replaced by **Type GetNode(int n, int i);**

Binomial pricer

Which are the changes?

Remark: The previous code (declaration and definition of the template class BinLattice) has to be written in a header (.h) file. There is no .cpp file corresponding to BinLattice. A class template can only be compiled after an object has been declared using the template with a specific data type, for example *double*, substituted for the type parameter, and we have not done so yet. **Separate compilation will not work** for them.

Binomial pricer

Function templates Templates help in defining generic classes and functions and hence allow generic programming. Generic programming is an approach where generic data types are used as parameters and the same piece of code works for various data types.

Function templates are used to create family of functions with different argument types. The format of a function template is shown below:

```
template<typename T> return_type
   function_name(arguments of type T)
{...}
```

Binomial pricer

How should we modify the PriceBySnell function?

```
double PriceBySnell(const BinModel& Model,
   BinLattice<double>& PriceTree,
   BinLattice<bool>& StoppingTree);
```

Binomial pricer

How should we modify the PriceBySnell function?

```
double AmOption::PriceBySnell (const 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);
}
```

Binomial pricer

How should we modify the PriceBySnell function?

```
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);
```

Binomial pricer

How should we modify the PriceBySnell function?

```
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);
}
```

Pricing in the Black-Sholes Model

Pricing in the Black-Sholes Model

Mathematical background

- The model for stock price evolution is

$$dS_t = \mu S_t dt + \sigma S_t dW_t, \tag{1}$$

and a riskless bond B, grows at a continuously compounded rate r.

- The Black-Sholes pricing theory then tells us that the price of a vanilla option, with expiry T and payoff f, is equal to

$$e^{-rT}\mathbb{E}^{\mathbb{Q}}[f(S_T)], \tag{2}$$

where the expectation is taken under the risk-neutral probability measure $\mathbb{Q}$.

Pricing in the Black-Sholes Model

Mathematical background

Recall:

- The model for stock price evolution under the probability measure $\mathbb{P}$ is

$$dS_t = \mu S_t dt + \sigma S_t dW_t, \tag{3}$$

- The evolution of the stock price under the probability measure $\mathbb{Q}$ is

$$dS_t = rS_t dt + \sigma S_t dW_t. \tag{4}$$

Pricing in the Black-Sholes Model

Mathematical background

Notation:

- S: Price of the underlying (stock price)
- K: Strike price
- r: Risk free interest rate.
- σ: Volatility.
- t: Current date.
- T: Maturity.

Pricing in the Black-Sholes Model

Mathematical background

Analytical option prices (we give the formulas only for call options, similar formulas for put options).

- The **payoff** of the call option is

$$C_T = max(S_T - K, 0). \tag{5}$$

- The **analytical option price** has the following functional form:

$$c = S\mathcal{N}(d_1) - Ke^{-r(T-t)}\mathcal{N}(d_2), \tag{6}$$

where

$$d_1 = \frac{log(\frac{S}{K}) + (r + \frac{1}{2}\sigma^2)(T-t)}{} = \tag{7}$$

Pricing in the Black-Sholes Model

Mathematical background

$$= \frac{log(\frac{S}{K}) + r(T-t)}{\sigma\sqrt{T-t}} + \frac{1}{2}\sigma\sqrt{T-t}.$$

and

$$d_2 = d_1 - \sigma\sqrt{T-t}.$$

$\mathcal{N}(\cdot)$ represents the cumulative normal distribution.

Pricing in the Black-Sholes Model

Mathematical background

$$= \frac{log(\frac{S}{K}) + r(T-t)}{\sigma\sqrt{T-t}} + \frac{1}{2}\sigma\sqrt{T-t}.$$

and

$$d_2 = d_1 - \sigma\sqrt{T-t}.$$

$\mathcal{N}(\cdot)$ represents the cumulative normal distribution.

Pricing in the Black-Sholes Model

Mathematical background

Computation of Greeks

In trading of options, a number of partial derivatives (called sensitivities or Greeks) of the option price is important.

- **Delta**

  The first derivative of the option price with respect to the underlying is called the *delta* of the option price. It is the derivative most people will run into, since it is important in *hedging* options.

$$\frac{\partial c}{\partial S} = \mathcal{N}(d_1);$$

Pricing in the Black-Sholes Model

Mathematical background

- **Gamma**

  The second derivative of the option wrt the underlying stock. These are equal for puts and calls

$$\Gamma_c = \frac{\partial^2 c}{\partial S^2} = \frac{\mathcal{N}(d_1)}{S\sigma\sqrt{T-t}}$$

- **Theta**

  The partial derivative with respect to time-to-maturity

$$\Theta_c = \frac{\partial c}{\partial (T-t)} = -\frac{\mathcal{N}(d_1)S\sigma}{2\sqrt{T-t}} - rKe^{-r(T-t)}\mathcal{N}(d_2).$$

Pricing in the Black-Sholes Model

Mathematical background

- **Vega**

  The partial with respect to volatility

$$\text{Vega}_c = \frac{\partial c}{\partial \sigma} = S\sqrt{T-t}\mathcal{N}(d_1)$$

Pricing in the Black-Sholes Model

Mathematical background

- **Rho**

  The partial with respect to the interest rate

$$\text{Rho}_c = \frac{\partial c}{\partial r} = K(T-t)e^{-r(T-t)}\mathcal{N}(d_2)$$

Pricing in the Black-Sholes Model

Implementation in C++

Pricing in the Black-Sholes Model

Implementation

We define a class called **BlackSholesModel**, which corresponds to the model of the asset price.

```
class BlackSholesModel{
public:
    double stockPrice;
    double volatility;
    double drift;
    double riskFreeRate;
    double date;

}
```

In order to simplify the presentation, we have declared the member variables as public. In practice, you have to declare them as private and to use the set and get functions.

Pricing in the Black-Sholes Model

Implementation

The choice of member variables in this class is carefully considered. This class **only** contains the variables associated with the model and not variables associated with the financial contract. We will specify different option contracts in different classes shortly.

Pricing in the Black-Sholes Model

Implementation

We define a separate class for the path independent option class. We'll create a base class **PathIndependentOption** which contains a maturity and a strike, but does not contain any details about the current market data.
The class contains as **public functions**:

- *Common functions*, with the *same behavior* for any PathIndependent Option in the BlackSoles model.
- *Common functions*, with *different bahaviors* depending on the type of option. Example: **payoff, price**. These methods are declared using the keyword **virtual** which means that these functions may be overriden.

Pricing in the Black-Sholes Model

Implementation

```
class PathIndependentOption{
private:
double maturity;
double strike;
public:
void SetMaturity(double);
double GetMaturity() const;
void SetStrike(double);
double GetStrike() const;
virtual double payoff(double StockAtMaturity)
   const=0;
virtual double price(const BlackSholesModel&
   bsm) const=0;
virtual ~PathIndependentOption(){};};
```

Pricing in the Black-Sholes Model

Implementation

We now define a new class, named **CallOption**, which is derived from the base class **PathIndependentOption**.

```
class CallOption: public PathIndependentOption
{
public:
double payoff (double x) const;
double price (const BlackSholesModel& bsm)
    const;
~CallOption(){};

}
```

Pricing in the Black-Sholes Model

Implementation

The function **CallOption::price** is implementing the formula given at the beginning of the lecture.

```
double CallOption: price (const
   BlackSholesModel& bsm) const
{
double S=bsm.stockPrice;
double K=strike;
double sigma=bsm.volatility;
double r=bsm.riskFreeRate;
double T=maturity-bsm.date;
double
   numerator=log(S/K)+(r+sigma*sigma*0.5)*T;
```

Pricing in the Black-Sholes Model

Implementation

```
double denominator=sigma*sqrt(T);
double d1=numerator/denominator;
double d2=d1-denominator;
return S*normcdf(d1)-exp(-r*T)*K*normcdf(d2);
}
```

normcdf($\cdot$) has been inplemented in the first practical, representing the cumulative standard normal distribution function.

Pricing in the Black-Sholes Model

Implementation

We now define a new class, named **PutOption**, which is derived from the base class **PathIndependentOption**.

```
class PutOption: public PathIndependentOption
{
public:
double payoff (double x) const;
double price (const BlackSholesModel& bsm)
    const;
~PutOption(){};

}
```

Pricing in the Black-Sholes Model

Implementation

Remarks:

- In order to compute the price of the Call (resp. Put) option, we have used the analytical formulas.
- What does it happen if we don't have explicit formulas? Solution: **Monte-Carlo**! This is the resaon for which the function **price** has been declared using the keyword **virtual** (if we have explicit formulas, we use them in order to compute the price, otherwise we price by Monte-Carlo!)

Pricing in the Black-Sholes Model

Implementation

We write a new class **MonteCarloPricer** that uses a **BlackScholes** model to simulate stock prices and then uses risk-neutral pricing to price a Path Independent Option by Monte Carlo.

Of course, as we said, pricing a European call or put option by Monte-Carlo is unnecessary since one already knows the BlackScholes formulas. However, implementing Monte-Carlo method for a call (or put) option is a valuable exercise which will help you to extend the ideas slightly to price a path-dependent option for which we have no analytical formula.

Pricing in the Black-Sholes Model

Monte-Carlo

Algorithm

To compute the Black–Scholes price of an option whose payoff is given in terms of the prices at times $t_0, t_1, t_2, \ldots, t_n$:

- Simulate stock price paths *in the risk-neutral measure*. i.e. use the algorithm with $\mu = r$.
- Compute the payoff for each price path.
- Compute the discounted mean value.
- This gives an unbiased estimate of the true risk-neutral price.

Pricing in the Black-Sholes Model

Monte-Carlo

Algorithm

More precisely, the main idea of Monte-Carlo is: Given a payoff function f, the price of the option $\gamma := \mathbb{E}^{\mathbb{Q}}[e^{-rT} f(S_T)]$ can be approximated by $\overline{X}_N := e^{-rT} \frac{1}{N} \sum_{i=0}^{i=N} f(S_T^i)$, with N large, where

- N is the number of simulations (this approximation follows by the Law Large Number).
- S^i - the i^{th} simulation of the price of the asset S under the risk neutral probability measure $\mathbb{Q}$.

Pricing in the Black-Sholes Model

Monte-Carlo

Algorithm

The estimator $\bar{X}_N$ gives an approximation of the price. A second important problem is the **estimation of the error**. In order to do this, we use the Central Limit Theorem which gives:

$$\sqrt{N}(\bar{X}_N - \gamma) \to \mathcal{N}(0, \sigma^2). \tag{8}$$

We'll learn how to implement the estimation of the error during the next practical.

A third important problem: Variance reduction techniques! (see tomorrow the practical).

Pricing in the Black-Sholes Model

Monte-Carlo

Simulation of the stock price paths

- Recall the dynamis of the stock price in continuous time (under probability $\mathbb{Q}$)

$$dS_t = S_t(rdt + \sigma dW_t). \tag{9}$$

- Write the **discrete** geometric Brownian motion under the probability measure $\mathbb{Q}$, which will be used for simulation.

$$S_{t_{i+1}} = S_{t_i} exp((r - \frac{\sigma^2}{2})(t_{t+1} - t_i) + \sigma(W_{t_{i+1}} - W_{t_i})). \tag{10}$$

Pricing in the Black-Sholes Model

Algorithm for Black–Scholes price paths

- Define
$$\delta t_i = t_i - t_{i-1}$$
- Simulate independent, normally distributed ϵ_i, with mean 0 and standard deviation 1
- Define $s_{t_0} = log(S_0)$ and then for $i \geq 1$
$$s_{t_i} = s_{t_{i-1}} + \left(r - \frac{1}{2}\sigma^2\right)\delta t_i + \sigma\sqrt{\delta t_i}\epsilon_i$$
$\sqrt{\delta t_i}\epsilon_i \stackrel{\mathcal{L}}{=} W_{t_i} - W_{t_{i-1}}$.
- Define $S_{t_i} = \exp(s_{t_i})$.
- S_{t_i} simulate the stock price (geiven by (10)) at the desired times.

Pricing in the Black-Sholes Model

Generating random numbers with Monte Carlo

As we have seen in the previous slide, we have to simulate independent, normally distributed ϵ_i, with mean 0 and standard deviation 1. How to do this?

Pricing in the Black-Sholes Model

Generating random numbers with Monte Carlo

- Conventional computers cannot generate true random numbers, they can only generate *pseudo random numbers*.
- Old method to generate uniformly distributed random numbers - using the **rand** function (it isn't a good choice because the sequence of pseudo random numbers it generates start to repeat themselves rather quickly).
- A much better algorithm: **Mersenne Twister**.
- The C++ class **mt19337** allows you to use the Mersenne Twister algorithm. In order to do this, one has to $\#include < random >$.

Pricing in the Black-Sholes Model

Generating random numbers with Monte Carlo

- How to use? First create a global variable of type mt19337 called **mersenneTwister**

```
static mt19337 mersenneTwister;
```

mersenneTwister is our random generator.

Remark related to the keyword static in this context: By marking a variable or a functions as **static**, we are saying it can only be used in the current source file. This means that we can reuse the name in other source files if desired. Note that although simply not including things in the header files is the main way of achieving information-hiding, you can go further using the keyword static.

Pricing in the Black-Sholes Model

Generating random numbers with Monte Carlo

- Function which generates *n* **independent uniformly distributed random numbers**.

```
vector<double> randuniform (int n){
vector<double> ret(n, 0.0);
for (int i=0;i<n; i++){
ret[i]=(mersenneTwister()+0.5)/
   (mersenneTwister.max()+1.0);
}
return ret;}
```

Pricing in the Black-Sholes Model

Generating random numbers with Monte Carlo

- To generate a random integer we write **mersenneTwister()**. This returns a random integer in the range **mersenneTwister.min()=0** to **mersenneTwister.max()**.

Pricing in the Black-Sholes Model

Generating random numbers with Monte Carlo
Simulation of gaussian random variables

- **Method 1.** Generate a uniformly distributed random variable u. Let F^{-1} be the inverse of the standard normal cumulative distribution. Then $F^{-1}(u) \sim \mathcal{N}(0,1)$.
- **Method 2 (Box-Muller algorithm)** Generate two uniformy distributed random variables u_1 and u_2 in the interval $(0,1)$. Define

$$n_1 = \sqrt{-2.0 \log(u_1)} \cos(2\pi u_2) \tag{11}$$

$$n_1 = \sqrt{-2.0 \log(u_1)} \sin(2\pi u_2). \tag{12}$$

n_1 and n_2 will be independent normally distributed random variables with mean 0 and standard deviation 1.

Pricing in the Black-Sholes Model

Generating random numbers with Monte Carlo
Simulation of gaussian random variables

- Function which generates *n* **independent normally distributed random numbers using Method 1**.

```
vector<double> randn(int n)
{
vector<double> ret=randuniform(n);
for (int i=0;i<n;i++)
ret[i]=norminv(ret[i]);
return ret;
}
```

The function **norminv** representing the inverse of the standard normal cumulative distribution function has been implemented in the first practical.

Pricing in the Black-Sholes Model

Generate price paths in C++

We now wish to add a function **generatePricePath** to class BlackSholesModel which takes a final date toDate and a number of steps nSteps and generates a random Black–Scholes Price path with the given number of steps(+1).

```
class BlackScholesModel {
public:
    ... other members of BlackScholesModel ...

    std::vector<double> generatePricePath(
                                double toDate,
                                int nSteps) const;
};
```

Pricing in the Black-Sholes Model

Generate price paths in C++

Note that the class declaration effectively contains the specification. If you choose good function and variable names, you won't need too many comments.

Pricing in the Black-Sholes Model

Generate price paths in C++

We also want a function generateRiskNeutralPricePath which behaves the same, except it uses the $\mathbb{Q}$-measure to compute the path.

```
class BlackScholesModel {
public:
    ... other members of BlackScholesModel ...

    std::vector<double>
   generateRiskNeutralPricePath(
                                double toDate,
                                int nSteps) const;
};
```

Pricing in the Black-Sholes Model

Generate price paths in C++

Private helper function

To implement these functions, we introduce a private function that allows you to choose the drift in the simulation of the price path.

```
class BlackScholesModel {
    ... other members of BlackScholesModel ...
private:
    std::vector<double>
   generateRiskNeutralPricePath(
                                double toDate,
                                int nSteps,
                                double drift)
   const;};
```

Pricing in the Black-Sholes Model

Generate price paths in C++

Private helper function

This function is private because we've only created it to make the implementation easier. Users of the class don't need (or even want) to know about it.

Pricing in the Black-Sholes Model

Generate price paths in C++

Implement the helper function

```
vector<double>
   BlackScholesModel::generatePricePath(
        double toDate,
        int nSteps,
        double drift ) const {
    vector<double> path(nSteps+1,0.0);
    path[0]=stock;
    vector<double> epsilon = randn( nSteps );
    double dt = (toDate-date)/nSteps;
    double a = (drift -
   volatility*volatility*0.5)*dt;
    double b = volatility*sqrt(dt);
```

Pricing in the Black-Sholes Model

Generate price paths in C++

Implement the helper function

```
    double currentLogS = log( stockPrice );
    for (int i=0; i<nSteps; i++) {
        double dLogS = a + b*epsilon[i];
        double logS = currentLogS + dLogS;
        path[i+1] = exp( logS );
        currentLogS = logS;
    }
    return path;
}
```

Pricing in the Black-Sholes Model

Generate price paths in C++

Implement the public functions

```
vector<double>
    BlackScholesModel::generatePricePath(
          double toDate,
          int nSteps ) const {
     return generatePricePath( toDate, nSteps,
    drift );
}
```

Pricing in the Black-Sholes Model

Generate price paths in C++

Implement the public functions

```
vector<double> BlackScholesModel::
    generateRiskNeutralPricePath(
        double toDate,
        int nSteps ) const {
    return generatePricePath(
        toDate, nSteps, riskFreeRate );
}
```

Notice that with this design we've avoided writing the same complex code twice.

Pricing in the Black-Sholes Model

Monte-Carlo specification

We want to write a class called **MonteCarloPricer** that:

- Is configured with nScenarios , the number of scenarios to generate and nSteps , the number of time steps.
- Has a function **price** which takes a **CallOption** and a **BlackScholesModel**, and computes (by **Monte Carlo**) the price of the **CallOption**.

We'll see that the declaration for **MonteCarloPricer** is pretty much the same thing as this specification.

Pricing in the Black-Sholes Model

Monte-Carlo declaration (in the header file)

```
class MonteCarloPricer {
public:
    /*  Constructor */
    MonteCarloPricer();
    /*  Number of scenarios */
    int nScenarios;
    /* number of steps */
    int nSteps;
    /*  Price a call option */
    double price( const CallOption& option,
                  const BlackScholesModel&
   model );
};
```

Pricing in the Black-Sholes Model

MonteCarlo.cpp

```
#include "MonteCarloPricer.h"

using namespace std;

MonteCarloPricer::MonteCarloPricer() :
    nScenarios(10000), nSteps(100) {
}
```

Pricing in the Black-Sholes Model

MonteCarlo.cpp

The implementation of price

```
double MonteCarloPricer::price(
        const CallOption& callOption,
        const BlackScholesModel& model ) {
    double total = 0.0;
    for (int i=0; i<nScenarios; i++) {
        vector<double> path= model.
                generateRiskNeutralPricePath(
                    callOption.maturity,
                    nSteps );
        double stockPrice = path.back();
        double payoff = callOption.payoff(
   stockPrice );
        total+= payoff;
    }
```

Pricing in the Black-Sholes Model

MonteCarlo.cpp

The implementation of price

```
    double mean = total/nScenarios;
    double r = model.riskFreeRate;
    double T = callOption.maturity -
   model.date;
    return exp(-r*T)*mean;
}
```

Pricing in the Black-Sholes model

Implementation of the method CallOption::price using Monte-Carlo

```
double CallOption::price(
          const BlackScholesModel& model ) const
    {
     MonteCarloPricer pricer;
     return pricer.price( *this, model );
}
```

Pricing in the Black-Sholes Model

MonteCarlo.cpp

Our Monte-Carlo pricer defined above can be used for the pricing of a Call Option. What should we change in order to be able to price any Path Independent Option?
We have simply to replace:

```
double price( const CallOption& option,
                    const BlackScholesModel&
    model );
```

by

```
double price( const PathIndependentOption&
    option,
                    const BlackScholesModel&
    model );
```

Pricing in the Black-Sholes Model

MonteCarlo.cpp

The code developed in the function **price** will work with any Path Independent Option, because the **payoff** has been declared virtual.

Conclusion: Making the above changes, we are now able to price any Path Independent Option (try to add also other class, DigitalCall, Digital Put...).

A general class hierarchy to price path-independent and path-dependent options

Pricing in the Black-Sholes model

Aim: Add the ability to price a path-dependent option.

Example: An up-and-out knock-out call option with strike K, barrier B, and maturity T is an option which pays off:

$$\begin{cases} \max\{S_T - K, 0\} & \text{if } S_t < B \text{ for all } t \in [0, T] \\ 0 & \text{otherwise.} \end{cases} \tag{13}$$

What are the changes that we have to make to our previous hierarchy of classes?

Pricing in the Black-Sholes model

1. We add a super class **ContinuousTimeOption**.

```
class ContinuousTimeOption {
private:
    double maturity;
    double strike;
public:
    virtual double price( const
   BlackScholesModel& bsm ) const=0;
    virtual double payoff(std::vector<double>&
   stock)=0;
    virtual ~ContinuousTimeOption() {};
    virtual bool isPathDependent() const=0;
```

Pricing in the Black-Sholes model

```
    double GetMaturity() const;
    void SetMaturity( double maturity );
    double GetStrike() const;
    void SetStrike( double strike );
};
```

- Base class provides basic implementations of methods common to most options.
- The base class has a virtual destructor. Recall that any class used as a base class must have a virtual destructor!

Pricing in the Black-Sholes model

- The payoff function has as parameter a vector (containing the entire path of the stock price), instead of only one value representing the terminal value of the stock.

```
virtual double payoff(std::vector<double>&
    stock)=0;
```

- We have add a virtual method **isPathDependent**.
- How should we adapt the design of our PathIndependentOption class?

Pricing in the Black-Sholes model

2. We write the class **PathIndependentOption** as a derived class from **ContinuousTimeOption**.

```
class PathIndependentOption : public
    ContinuousTimeOption {
public:
     /*  Calculate the payoff of the option
    given a history of prices */
     double payoff(const std::vector<double>&
    stockPrices) const;
     virtual double payoff(double stock)
    const=0;
     bool isPathDependent() const;
     virtual ~PathIndependentOption(){};
};
```

Pricing in the Black-Sholes model

```
double PathIndependentOption ::payoff(const
   std::vector<double>& stockPrices) const
{return payoff(stockPrices.back());};

 bool PathIndependentOption :isPathDependent()
   const {
        return false;};
```

Pricing in the Black-Sholes model

- Note that: We have written the implementation of the function **payoff(vector<double> &)** using the function **payoff(double stock)**, which is a virtual function as before.
- No modification is needed to the classes derived from **PathIndependentOption**.

Pricing in the Black-Sholes model

3. We now add a class **PathDependentOption** derived from **ContinuousTimeOption**.

```
class PathdependentOption :
        public ContinuousTimeOption {
public:

    virtual ~PathDependentOption() {}

    bool isPathDependent() const {return
   true;};
```

Pricing in the Black-Sholes model

4. We add a class **UpAndOut knock-out** derived from **PathDependentOption** (we'll see the implementation the next practical).
We can easily extend our hierarchy of classes, by adding **Arithmetic Asian Calls**, **Arithmetic Asian Puts** etc.

Pricing in the Black-Sholes model

MonteCarlo.cpp

Modification in the MonteCarlo price function

```
double MonteCarloPricer::price(
        const ContinuousTimeOption& Option,
        const BlackScholesModel& model ) {
    double total = 0.0;
    for (int i=0; i<nScenarios; i++) {
        vector<double> path= model.
                generateRiskNeutralPricePath(
                    Option.maturity,
                    nSteps );
       double payoff = Option.payoff(path);
        total+= payoff;
    }
```

Pricing in the Black-Sholes Model

MonteCarlo.cpp

Modification in the MonteCarlo price function

```
    double mean = total/nScenarios;
    double r = model.riskFreeRate;
    double T = Option.maturity - model.date;
    return exp(-r*T)*mean;
}
```

Remark: In order to simplify the code, I have simply put *Option.maturity*. *maturity* is a private member of the class **ContinuousTimeOption**, so we can't access it from outside the class. In your implementation, you have to replace *Option.maturity* by *Option.GetMaturity*().

Summing up

- We have written a generic **Monte-Carlo pricer** which contains:
  - A class representing the model (the evolution of the asset price): **class BlackSholesModel**
  - A complex hierarchy of classes representing the financial contract (**path-independent** and **path-dependent**).
  - A class representing the Monte-Carlo pricer.