Roxana Dumitrescu

C++ in Financial Mathematics

What have we learnt?

- Class tempates and function templates
- Simulation of random variables and Monte-Carlo algorithm in C++
- Design and implemetation of an option pricer by Monte-Carlo for path-independent options.

Plan

- Extension of the option pricer by Monte-Carlo to path dependent options.
- Computation of the implied volatility in the Black-Sholes model using different C++ techniques

Pricing in the Black-Sholes Model

Monte-Carlo

Algorithm

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:

$$\sqrt{N}(\bar{X}_N - \gamma) \to \mathcal{N}(0, \sigma^2). \tag{1}$$

Pricing in the Black-Sholes Model

Monte-Carlo

Algorithm

In order to obtain an asymptotic $1-\alpha$ confidence interval (for simplicity we take $r=0$):

- By symmetry of the normal distribution,

$$\mathbb{P}\left[-\frac{q_{1-\frac{\alpha}{2}}}{\sqrt{N}}\sigma \leq \bar{X}_N - \gamma \leq \frac{q_{1-\frac{\alpha}{2}}}{\sqrt{N}}\sigma\right] = 1-\alpha$$

- This implies that

$$\mathbb{P}\left[\gamma \in \left[\bar{X}_N - \frac{q_{1-\frac{\alpha}{2}}}{\sqrt{N}}\sigma, \bar{X}_N + \frac{q_{1-\frac{\alpha}{2}}}{\sqrt{N}}\sigma\right]\right] = 1-\alpha$$

Pricing in the Black-Sholes Model

Monte-Carlo

Algorithm

- An estimator of the standard deviation of the price $\frac{\sigma}{\sqrt{N}}$ is $\frac{\hat{\sigma}}{\sqrt{N}}$, with

$$\hat{\sigma} = \sqrt{\frac{1}{N}\sum_{i=1}^{N} Payoff(S^i(T))^2 - \left(\frac{1}{N}\sum_{i=1}^{N} Payoff(S^i(T))\right)^2}$$

- $q_{1-\frac{\alpha}{2}} = \text{norminv}(1-\frac{\alpha}{2})$.

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 for the initial time).

```
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:

$$\begin{cases} \max\{S_T - K, 0\} & \text{if } S_t < B \text{ for all } t \in [0, T] \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

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*().

Computation of the implied volatility in the Black-Sholes model

Computation implied volatility Black-Sholes

Recall that within the Black-Sholes model, the price of a European call option with expirt time T and strike price K is given by the Black-Sholes formula

$$C(S(0), K, T, \sigma, r) = S(0)\mathcal{N}(d_1) - ke^{-rT}\mathcal{N}(d_2), \tag{3}$$

with d_1, d_2 and $\mathcal{N}(\cdot)$ habe been indicated in a previous lecture.

Computation implied volatility Black-Sholes

If the values of $S(0)$, K, T, σ, r are well known, then formula can be used to compute the option price predicted by the Black-Sholes model. Among these variables, T and K are **known** quantities written into the option contract. Quotes for the stock price $S(0)$ and interest rate r are also readily available. The **volatility** σ is not available and needs to be **estimated from market data**.

Computation implied volatility Black-Sholes

How to estimate σ?
We have too look up the market price C_{quote} quoted for some European call option and then to solve the nonlinear equation

$$C(S(0), K, T, \sigma, r) = C_{\text{quote}}$$

for σ, given the values of $S(0), K, T, r$. The solution σ, called **implied volatility**, can then be used to price and hedge options of other kinds.
There are several numerical methods for solving this kind of non-linear equations. We will show how to implement two of them: the **Bisection method** and the **Newton-Raphson method**.

Non-linear solvers

Non-linear solvers

First method: Bisection method

We want to compute a solution x to an equation

$$f(x) = c, \tag{4}$$

where f is a given function from an interval $[a, b]$ to $\mathbb{R}$. It is assumed that f is continuous on $[a, b]$ and $f(a) - c, f(b) - c$ have opposite signs. Then there must be an $x \in [a, b]$ such that $f(x) = c$.

Non-linear solvers

First method: Bisection method

The bisection method works by constructing sequences l_n, r_n of left and risht approximations by induction:

- Let
$$l_0 = a, r_0 = b.$$
- If l_n and r_n have already been constructed for some $n = 0, 1, 2, ...,$ let
$$l_{n+1} = l_n, r_{n+1} = \frac{1}{2}(l_n + r_n)$$

Non-linear solvers

First method: Bisection method

if $f(l_n) - c$ and $f(1/2(l_n + r_n)) - c$ have opposite signs, and

$$l_{n+1} = 1/2(l_n + r_n), \quad r_{n+1} = r_n$$

otherwise.

Then $l_n \uparrow x$ and $r_n \downarrow x$ as $n \to \infty$, where $x \in [a, b]$ is a solution of $f(x) = c$.

Non-linear solvers

Second method: Newton-Raphson method

In this case f is assumed to be differentiable on $[a, b]$. We construct a sequence x_n as follows:

- Take

$$x_0 \in (a, b).$$

- For $n = 0, 1, 2, \ldots$ let

$$x_{n+1} = x_n - \frac{f(x_n) - c}{f'(x_n)}.$$

If the equation $f(x) = c$ has a solution $x \in (a, b)$ such that $f'(x) \neq 0$ and x_0 is chosen close enough to x, then x_n will converge to x.

Non-linear solvers

Second method: Newton-Raphson method

Implementation: 3 possible techniques

- Function pointers
- Virtual functions
- Templates

Non-linear solvers

Implementation: First technique - Function pointers

```
double SolveByBisect(double (*Fct)(double x),
    double Tgt, double LEnd, double REnd,
    double Acc)
{
double left=Lend, right=Rend,
    mid=(left+right)/2;
     double y_left=Fct(left)-Tgt,
    y_mid=Fct(mid)-Tgt;
```

Non-linear solverss

Implementation: First technique - Function pointers

```
    while(mid-left>Acc)
    {
        if
   ((y_left>0&&y_mid>0)||(y_left<0&&y_mid<0))
            {
                left=mid;
                y_left=y_mid;
            }
            else right=mid;
            mid=(left+right)/2;
            y_mid=Fct(mid)-Tgt;
    }
    return mid;
}
```

Non-linear solvers

Implementation: First technique - Function pointers

- The function **SolveByBisect()** is defined to implement the bisection method. It takes a function pointer Fct as an argument, so a function *f* can be passed to the solver.
- In addition, SolveByBisect() takes arguments `Tgt` for the target value `c` of the function, `LEnd` and `REnd` for the left and right endpoints of the interval $[a, b]$, and `Acc` for the desired accuracy of the numerical solution, which will determine when the algorithm should be stopped.

Non-linear solvers

Implementation: First technique - Function pointers

```
double SolveByNR(double (*Fct)(double x),
    double (*DFct)(double x), double Tgt,
    double Guess, double Acc)
{
double x_prev=Guess;
double
    x_next=x_prev-(Fct(x_prev)-Tgt)/DFct(x_prev);
while (x_next-x_prev>Acc || x_prev-x_next>Acc)
{
x_prev=x_next;
x_next=x_prev-(Fct(x_prev)-Tgt)/DFct(x_prev);
}
return x_next;
}
```

Non-linear solvers

Implementation: First technique - Function pointers

- The function **SolveByNR()** implements the Newton-Raphson solver.
- It takes two function pointers `Fct` and `DFct` so that both a function f and its derivative f' can be passed. Moreover, SolveByNR() also takes arguments `Tgt` for the target value `c` of the function, `Guess` for the initial term x_0 of the approximating sequence and `Acc` for the desired accuracy of the numerical solution.

Non-linear solvers

Implementation: First technique - Function pointers

How to use?

```
double F1(double x){return x*x-2;}
double DF1(double x){return 2*x;}
int main()
{
double Acc=0.001;
double LEnd=0.0;
double REnd=2.0;
double Tgt=0.0;
cout<<SolveByBisect(F1,Tgt,LEnd,REnd,Acc)<<endl;
double Guess=1.0;
cout<<SolveByNR(F1,DF1,Tgt,Guess,Acc)<<endl;
return 0;}
```

Non-linear solvers

Implementation: First technique - Function pointers

Remark

Function pointers are simple enough, but limit the possibilities for expansion. The code can readily be adapted for other functions $f(x)$ with a single argument x, but it would not be easy to handle a function with a parameter, for example $f(x, a) = x^2 - a$. The type of the function pointer is hardwired as `double (*Fct)(double x)`, which does not allow for extra parameters.

Solution? Virtual functions!

From now on, will present only the changes related to the implementation of the Bisection method. The same ideas apply for the Newton-Raphson method.

Non-linear solverss

Implementation: Second technique - Virtual functions

1. We first define an hierarchy of classes

```
class Function
{
public:
virtual double Value(double x)=0;
virtual double Deriv(double x)=0;
};

class F1: public Function
{
public:
double Value(double x){return x*x-2;}
double Deriv(double x){return 2*x;}
};
```

Non-linear solvers

Implementation: Second technique - Virtual functions

1. We first define an hierarchy of classes

```
class F2: public Function
{       private:
  double a; // parameter
public:
F2(double a){this->a=a;}
double Value(double x){return x*x-a;}
double Deriv(double x){return 2*x;}
};
```

Non-linear solvers

Implementation: Second technique - Virtual functions

2. We rewrite the functions

```
double SolveByBisect(Function* Fct, double
    Tgt, double LEnd, double REnd, double Acc)
{
double left=Lend, right=Rend,
    mid=(left+right)/2;
     double y_left=Fct->Value(left)-Tgt,
    y_mid=Fct->Value(mid)-Tgt;
```

Non-linear solvers

Implementation: Second technique - Virtual functions

2. We rewrite the functions

```
    while(mid-left>Acc)
    {
        if
   ((y_left>0&&y_mid>0)||(y_left<0&&y_mid<0))
              {
                   left=mid;
                   y_left=y_mid;
              }
              else right=mid;
              mid=(left+right)/2;
              y_mid=Fct->Value(mid)-Tgt;
    }
    return mid;}
```

Non-linear solvers

Implementation: Second technique - Virtual functions

Remarks

- An interface class `Function` is introduced to represent a general function f. The class has two pure virtual member functions `Value()` to return the value of f at x, and `Deriv()` to return the value of the derivative f' at x. Some concrete functions can now be introduced as subclasses of the `Function` class, and then the solvers called to do their business.

Implementation: Second technique - Virtual functions

Remarks

- `Fct` is passed to `SolveByBisect()` and `SolveByNR()` no longer as a function pointer to the `Function` class. We need to work with a pointer to this class to take the advantage of virtual functions. `DFct`, the function pointer passed to `SolveByNR()` in the previous version to compute the derivative is no longer needed. `Deriv()` is now a member of the `Function` class, and both `Value()` and `Deriv()` can be accessed through the pointer `Fct` to this class.

Non-linear solvers

Implementation: Second technique - Virtual functions

How to use?

```
int main()
{
double Acc=0.001;
double LEnd=0.0;
double REnd=2.0;
double Tgt=0.0;
F1 MyF1;
F2 MyF2(3.0);
cout<<SolveByBisect(&MyF1,Tgt,LEnd,REnd,Acc)<<endl;
double Guess=1.0;
cout<<SolveByBisect(&MyF2,DF1,Tgt,Guess,Acc)<<endl;
return 0;}
```

Non-linear solvers

Implementation: Third technique - Template functions

1. Write the non-linear solvers as template functions

```
template <typename Function> double
   SolveByBisect(Function* Fct, double Tgt,
   double LEnd, double REnd, double Acc)
{
double left=Lend, right=Rend,
   mid=(left+right)/2;
    double y_left=Fct->Value(left)-Tgt,
   y_mid=Fct->Value(mid)-Tgt;
```

Non-linear solvers

Implementation: Third technique - Template functions

1. Write the non-linear solvers as template functions

```
while(mid-left>Acc)
{
    if
((y_left>0&&y_mid>0)||(y_left<0&&y_mid<0))
          {
              left=mid;
              y_left=y_mid;
          }
          else right=mid;
          mid=(left+right)/2;
          y_mid=Fct->Value(mid)-Tgt;
}
return mid;}
```

Non-linear solvers

Implementation: Third technique - Templates

2. Declaration of the classes

```
class F1
{
public:
double Value(double x){return x*x-2;}
double Deriv(double x){return 2*x;}
};
```

Non-linear solvers

Implementation: Third technique - Templates

2. Declaration of the classes

```
class F2
{       private:
  double a; // parameter
public:
F2(double a){this->a=a;}
double Value(double x){return x*x-a;}
double Deriv(double x){return 2*x;}
};
```

Non-linear solvers

Implementation: Third technique - Templates

Remarks

- `Function` has become a template parameter. The `Function` class is gone. Gone with it are the pure virtual functions. The old function **SolveByBisect()** has have become **function templates**.
- F1 and F2 are no longer subclasses of anything, because the parent class is gone.
- The compiler can decide how to compile the template functions by looking at the first parameter passed to **SolveByBisect()**. If &*MyF*1 is passed, it substitutes class *F*1 for the parameter `Function` when compiling the code. When &*MyF*2 is passed, then it substitutes *F*2 and compiles another version of the code.

Implementation: Third technique - Templates

Remarks

- Because we no longer work with virtual functions, using pointers is not absolutely necessary. One can change the type of the argument `Fct` passed to **SolveByBisect()** from a pointer to `Function` to an object of type `Function`.

Non-linear solvers

Implementation: Third technique - Templates

How to use?

```
int main()
{
double Acc=0.001;
double LEnd=0.0;
double REnd=2.0;
double Tgt=0.0;
F1 MyF1;
F2 MyF2(3.0);
cout<<SolveByBisect(&MyF1,Tgt,LEnd,REnd,Acc)<<endl;
double Guess=1.0;
cout<<SolveByBisect(&MyF2,DF1,Tgt,Guess,Acc)<<endl;
return 0;}
```

Computation of the implied volatility

We present here only the method using templates. We let as exercise the technique using virtual functions.

Computation implied volatility Black-Sholes

1. We define a class EurCall.

```
class EurCall{
public: double T,K;
    EurCall (double T, double K);
    double PriceByBSFormula(double S0, double
    sigma, double r);
    double VegaByBSFFormula(double S0, double
    sigma, double r);
};
```

Remark: Of course, it would be a much better design to consider the BlackAndSholes class. Here we concentrate only on the use of nonlinear solvers defined as templates.

Computation implied volatility Black-Sholes

1. We define a class Intermediary, which contains the functions Deriv and Value that are used in the non-linear solver functions.

```
class Intermediary: public EurCall
{
private: double S0,r;
public:
Intermediary(double S0, double r, double T,
    double K): EurCall(T,K) {this->S0=S0;
    this->r=r;}
double Value (double sigma)
{return PriceByBSFormula(S0,sigma,r);}
double Deriv(double sigma)
{ return VegaByBSFormula(S0,sigma,r);}
};
```

Computation implied volatility Black-Sholes

How to use?

```
int main()
{
double S0=100.;
double r=0.1;
double T=1.0;
double K=100.;
Intermediary Call(S0,r,T,K);
double Acc=0.001;
double LEnd=0.01, REnd=1.0;
double Tgt=12.56;
cout<<SolveByBisect(&Call,Tgt,LEnd,REnd,Acc)<<endl;
return 0;}
```

Computation implied volatility Black-Sholes

Remark: `Tgt` has to be initiated with the observed call price $C_{\texttt{quote}}$.

Computation implied volatility Black-Sholes

Remarks on templates

- One difficulty with templates is that they are prone to programming bugs and be harder to debug than code without templates, producing mystfying compiler errors.
- **Use templates** for relatively small **generic tasks and structures** that are likely to be recyled in several different contexts. Larger or more specific tasks might be better off without templets.
- For example , templates are a very good choice for our BinLattice class which stores the prices/stopping strategy in the case of an American option (we have to use the same stucture data with **bool** and **double**).On the other hand, a full-scale option pricer would seem to be unlikely candidate for templatising, as it would be too large and too specific. This is the reason for which you should prefer virtual function for option pricing.

Summing up

We have learnt about:

- The design and the implementation of an option pricer for path-dependent options by Monte-Carlo.
- How can we compute the implied volatility in the BlackAndSholes model using the nonlinear solvers?
- Three different techniques to implement the nonlinear solvers (with their advantages and disadvanatges): function pointers, virtual functions, templates.