NYU | TANDON SCHOOL OF ENGINEERING

Topic 8
Implementation of Non-linear Solvers in C++

Overview

- Implied volatility
- Well-known numerical methods for solving non-linear equations:
  - Bisection Method
  - Newton-Raphson Method
- Three implementations of these methods:
  - Function pointers
  - Virtual functions
  - Template functions (Class Template)
- Computing implied volatility

Implied Volatility

- Within the Black-Scholes model, the price of a European call option with expiry time ***T*** and strike price ***K*** is given by the ***Black-Scholes Formula:***

$$C(S(0),K,T,\sigma,r) = S(0)N(d_+) - Ke^{-rT}N(d_-)$$

$$d_+ = \frac{\ln(S(0)/K) + (r+\sigma^2/2)T}{\sigma\sqrt{T}}, d_- = d_+ - \sigma\sqrt{T},$$

Where S(0) is the price of the underlying stock (the spot price), σ is the stock volatility, r is the continuously compounded interested and where

$$N(x) = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} e^{-y^2/2} dy$$

is the distribution function of the standard normal distribution N(0,1), with mean 0 and variance 1.

- Among these variables, ***T*** and ***K*** are known quantities written into the option contract. Quotes for the current stock price S(0) and the interest rate r are also readily available.
- One popular way of estimating σ is to look up the market price C_{quote} quoted from some European call option and then to solve the non-linear equation:
  - **$C(S(0), K, T, \sigma, r) = C_{quote}$** , σ is called ***implied volatility***.

Bisection method

- We want to compute a solution ***x*** to an equation: f(x) = c
  - where f is given function from an interval [a, b] to 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 ϵ [a, b], such that f(x) = c.
- The bisection method works by constructing sequence l_n, r_n of left and right approximation 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
    - $l_{n+1} = l_n$, $r_{n+1} = (l_n + r_n)/2$ if ***$f(l_n)$-c and $f((l_n+r_n)/2)$-c*** have opposite signs.
    - $l_{n+1} = (l_n + r_n)/2$, $r_{n+1} = r_n$, otherwise
  - Then $l_n \nearrow x$ and $r_n \searrow x$ as n -> ∞, where x ϵ [a, b] is a solution to f(x) = c.

Newton-Raphson Method

- In the case f is assumed to be differentiable on [a, b]. We construct a sequence x_n as followings:
  - Take $x_0 \in (a, b)$
  - For n = 0, 1, … 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 as $f'(x) \neq 0$ and x_0 is chosen close enough to x, then x_n will converge to x.

Function01.h

```
#pragma once
namespace fre {
	double F1(double x);
	double DF1(double x);
}
```

Function01.cpp

```
#include "Function01.h"
namespace fre {
	double F1(double x) { return x * x - 2; }
	double DF1(double x) { return 2 * x; }
}
```

NonlinearSolver01.h - using Function Pointers

```
#pragma once
#include "Function01.h"
namespace fre {
   class NonlinearSolver
   {
   private:
      double Tgt;
      double LEnd;
      double REnd;
      double Acc;
      double Guess;
   public:
      NonlinearSolver() : Tgt(0), LEnd(0), REnd(0), Acc(0), Guess(0) {}
      NonlinearSolver(double Tgt_, double LEnd_, double REnd_, double Acc_, double Guess_)
      : Tgt(Tgt_), LEnd(LEnd_), REnd(REnd_), Acc(Acc_), Guess(Guess_) {}
      ~NonlinearSolver() {}
```

NonlinearSolver01.h (continue)

```
    void UpdateSolver(double Tgt_, double LEnd_, double REnd_, double Acc_, double Guess_)
    {
      Tgt = Tgt_;
      LEnd = LEnd_;
      REnd = REnd_;
      Acc = Acc_;
      Guess = Guess_;
    }
    double SolveByBisect(double (*Fct)(double x));
    double SolveByNR(double (*Fct)(double x), double (*DFct)(double x));
  };
}
```

NonlinearSolver01.cpp - implementing Function Pointers

```
#pragma once
#include "Function01.h"
#include "NonlinearSolver01.h"
namespace fre {
   double NonlinearSolver::SolveByBisect(double (*Fct)(double x))
   {   double left = LEnd, right = REnd, mid = (left + right) / 2;
      double y_left = Fct(left) - Tgt, y_mid = Fct(mid) - Tgt;
      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;
   }
```

NonlinearSolver01.cpp (continue)

```
double NonlinearSolver::SolveByNR(double (*Fct)(double x), double (*DFct)(double x))
{
    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;
}
}
```

- Notes:
  - The function ***SolveByBisect()***is defined to implemented the bisection method.
    - Care is taken in SolveByBisect() that only one evaluation of ***f*** is made for each loop iteration in order to speed up computation.
  - ***SolveByNR()*** implements the Newton-Raphson solver.
    - It takes two functions ***Fct*** and ***DFct*** so that both a function ***f*** and its derivative ***f′*** can be passed.
    - It takes arguments ***Tgt*** for target value ***c, Guess*** for the initial term x_0 of the approximating sequence, and ***Acc*** for the desired accuracy to be reached before the algorithm is terminated.
    - The iteration x_n converges to an ***x*** such that f(x) = c if f′(x) ≠ 0 and x_0 is close enough to x.

main01.cpp

```
#include <iostream>
#include <iomanip>
#include "Function01.h"
#include "NonlinearSolver01.h"
using namespace std;
using namespace fre;
int main()
{
   double Acc1 = 0.0001;
   double LEnd1 = 0.0, REnd1 = 2.0;
   double Tgt1 = 0.0, Guess1 = 1.0;
   NonlinearSolver solver(Tgt1, LEnd1, REnd1, Acc1, Guess1);
   cout << "Root of F1 by Bisect: " << fixed << setprecision(4) << solver.SolveByBisect(F1) << endl;
   cout << "Root of F1 by Newton-Raphson: " << fixed << setprecision(4)
            << solver.SolveByNR(F1, DF1) << endl << endl;
   return 0;
}
```

Notes:

- The above program shows how to solve the equation
  - f(x) = 0 for $f(x) = x^2 - 2$ via SolveByBisect() and SolvebyNR().
- Function pointers are simple enough but may limit the possibilities for expansion. For example, it would not be easy to handle a function with a parameter, for example, $f(x, a) = x^2 - a$, $a \varepsilon R$.

Function02.h

```
#pragma once
namespace fre {
   class Function
   {
   public:
        virtual double Value(double x) = 0;
        virtual double Deriv(double x) = 0;
        virtual ~Function() {}
   };
```

```
   class F1 : public Function
   {
   public:
      double Value(double x);
      double Deriv(double x);
   };

   class F2 : public Function
   {
   private:
      double a; //any real number
   public:
      F2(double a_) { a = a_; }
      double Value(double x);
      double Deriv(double x);
   }
}
```

Function02.cpp

```
#include "Function02.h"
namespace fre {
	double F1::Value(double x) { return x * x - 2; }
	double F1::Deriv(double x) { return 2 * x; }

	double F2::Value(double x) { return x * x - a; }
	double F2::Deriv(double x) { return 2 * x; }
}
```

NonlinearSolver02.h – Using virtual functions

```
#pragma once
#include "Function02.h"
namespace fre {
   class NonlinearSolver
   {
   private:
      double Tgt;
      double LEnd;
      double REnd;
      double Acc;
      double Guess;
   public:
      NonlinearSolver() : Tgt(0), LEnd(0), REnd(0), Acc(0), Guess(0) {}
      NonlinearSolver(double Tgt_, double LEnd_, double REnd_, double Acc_, double Guess_)
      :  Tgt(Tgt_), LEnd(LEnd_), REnd(REnd_), Acc(Acc_), Guess(Guess_) {}
      ~NonlinearSolver() {}
```

NonlinearSolver02.h (continue)

```
    void UpdateSolver(double Tgt_, double LEnd_, double REnd_, double Acc_, double Guess_)
    {
        Tgt = Tgt_;
        LEnd = LEnd_;
        REnd = REnd_;
        Acc = Acc_;
        Guess = Guess_;
    }

     double SolveByBisect(Function* Fct);
     double SolveByNR(Function* Fct);
  };
}
```

NonlinearSolver02.cpp

```
#include "NonlinearSolver02.h"
namespace fre {
   double NonlinearSolver::SolveByBisect(Function* Fct)
   {
      double left = LEnd, right = REnd, mid = (left + right) / 2;
      double y_left = Fct->Value(left) - Tgt, y_mid = Fct->Value(mid) - Tgt;
      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;
   }
```

NonlinearSolver02.cpp (continue)

```
double NonlinearSolver::SolveByNR(Function* Fct)
{
    double x_prev = Guess;
    double x_next = x_prev - (Fct->Value(x_prev) - Tgt) / Fct->Deriv(x_prev);
    while (x_next - x_prev > Acc || x_prev - x_next > Acc)
    {
        x_prev = x_next;
        x_next = x_prev - (Fct->Value(x_prev) - Tgt) / Fct->Deriv(x_prev);
    }
    return x_next;
}
}
```

- Notes:
  - An abstract class called ***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***.
  - ***Fct*** is passed to ***SolveByBisect()*** and ***SolveByNR()*** as a pointer to the ***Function*** class, and both ***Value()*** and ***Deriv()*** can be accessed through the pointer ***Fct:***
    - ***Fct->Value(x_prev)*** returns the value of the function at x_prev.
    - ***Fct->Derive(x_prev)*** returns the derivative at x_prev.
  - A concrete function can now be implemented as a derived class of ***Function*** class.

main02.cpp

```
#include <iostream>
#include <iomanip>
#include "Function02.h"
#include "NonlinearSolver02.h"
using namespace std;
using namespace fre;
int main()
{    double Acc1 = 0.0001;
     double LEnd1 = 0.0, REnd1 = 2.0;
     double Tgt1 = 0.0, Guess1 = 1.0;
     NonlinearSolver solver(Tgt1, LEnd1, REnd1, Acc1, Guess1);
     F1 MyF1;
     cout << "Root of F1 by Bisect: " << fixed << setprecision(4)
          << solver.SolveByBisect(&MyF1) << endl;
     cout << "Root of F1 by Newton-Raphson: " << fixed << setprecision(4)
          << solver.SolveByNR(&MyF1) << endl;
```

```
    double Acc2 = 0.0001;
    double LEnd2 = 0.0, REnd2 = 4.0;
    double Tgt2 = 0.0;
    double Guess2 = 3.0;
    F2 MyF2(10.0);
    solver.UpdateSolver(Tgt2, LEnd2, REnd2, Acc2, Guess2);
    cout << "Root of F2 by Bisect: " << fixed << setprecision(4)
         << solver.SolveByBisect(&MyF2) << endl;
    cout << "Root of F2 by Newton-Raphson: " << fixed << setprecision(4)
         << solver.SolveByNR(&MyF2) << endl;
    return 0;
}
/*
Root of F1 by Bisect: 1.4142
Root of F1 by Newton-Raphson: 1.4142
Root of F2 by Bisect: 3.1623
Root of F2 by Newton-Raphson: 3.1623
*/
```

- Notes:
  - Class F1, a derived class of the ***Function***.
  - Class F2, a derived class of the ***Function***, define a new function, f(x, a) = x^2-a, which was hard to do with function pointers.
  - Class F2 contains a constructor function:
    - F2(double a_){a=a_;}
  - The addresses, &MF1 and &MF2 are passed to SolveByBisect() and SolveByNR(). Both functions have a base class pointer as their parameters, while the arguments to the function are the addresses of derived class objects. When the virtual functions Value() and Derive() are called via the pointer, they do their checking at run time and select the correct version to be executed- **Polymorphism**

Template Class

- Virtual functions performs type checking at run time. When it occurs inside a loop, it will possibly create noticeable computing time overheads.
- Templates offer a technique that largely retains the advantage of virtual functions over the function pointers, but shifts type checking from runtime to compile time. In some cases of heavy computations, it could mean measurable runtime savings for the end user.

Function03.h

```
#pragma once
namespace fre {
   class F1
   {
   public:
      double Value(double x);
      double Deriv(double x);
   };
   class F2
   {
   private:
      double a; //any real number
   public:
      F2(double a_) { a = a_; }
      double Value(double x);
      double Deriv(double x);
   };
}
```

Function03.cpp

```
#include "Function03.h"
namespace fre {
        double F1::Value(double x) { return x * x - 2; }
        double F1::Deriv(double x) { return 2 * x; }

        double F2::Value(double x) { return x * x - a; }
        double F2::Deriv(double x) { return 2 * x; }
}
```

NonlinearSolver03.h

```
#pragma once
#include "Function03.h"
namespace fre {
   template<typename Function>
   class NonlinearSolver
   {
   private:
      double Tgt;
      double LEnd;
      double REnd;
      double Acc;
      double Guess;
   public:
      NonlinearSolver() : Tgt(0), LEnd(0), REnd(0), Acc(0), Guess(0) {}
      NonlinearSolver(double Tgt_, double LEnd_, double REnd_, double Acc_, double Guess_)
      : Tgt(Tgt_), LEnd(LEnd_), REnd(REnd_), Acc(Acc_), Guess(Guess_)  {}
      ~NonlinearSolver() {}
```

```
void UpdateSolver(double Tgt_, double LEnd_, double REnd_, double Acc_, double Guess_)
{
   Tgt = Tgt_;
   LEnd = LEnd_;
   REnd = REnd_;
   Acc = Acc_;
   Guess = Guess_;
}
```

```
// template member function
double SolveByBisect(Function* Fct)
 {
    double left = LEnd, right = REnd, mid = (left + right) / 2;
    double y_left = Fct->Value(left) - Tgt, y_mid = Fct->Value(mid) - Tgt;
    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;
 }
```

```
    // template member function
    double SolveByNR(Function* Fct)
     {
        double x_prev = Guess;
        double x_next = x_prev - (Fct->Value(x_prev) - Tgt) / Fct->Deriv(x_prev);
        while (x_next - x_prev > Acc || x_prev - x_next > Acc)
        {
           x_prev = x_next;
           x_next = x_prev - (Fct->Value(x_prev) - Tgt) / Fct->Deriv(x_prev);
        }
        return x_next;
     }
  };
}
```

main03.cpp

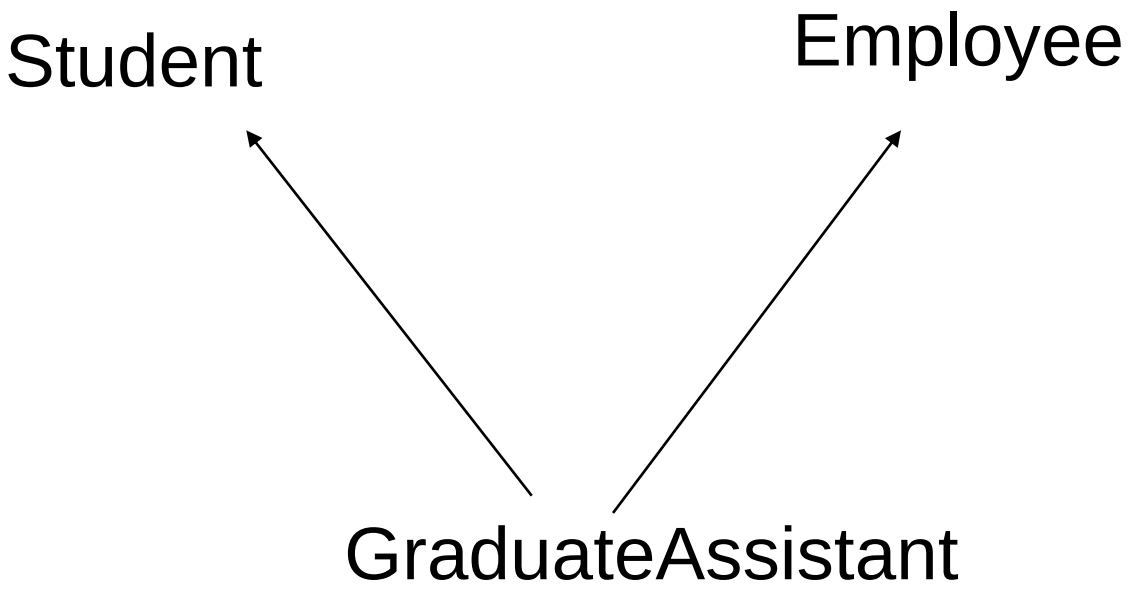
```
#include <iostream>
#include <iomanip>
#include "Function03.h"
#include "NonlinearSolver03.h"
using namespace std;
using namespace fre;
int main()
{   double Acc1 = 0.0001;
    double LEnd1 = 0.0, REnd1 = 2.0;
    double Tgt1 = 0.0, Guess1 = 1.0;
    F1 MyF1;
    NonlinearSolver<F1> solver1(Tgt1, LEnd1, REnd1, Acc1, Guess1);
    cout << "Root of F1 by Bisect: " << fixed << setprecision(4)
                << solver1.SolveByBisect(&MyF1) << endl;
    cout << "Root of F1 by Newton-Raphson: " << fixed << setprecision(4)
                << solver1.SolveByNR(&MyF1) << endl;
```

```
    double Acc2 = 0.0001;
    double LEnd2 = 0.0, REnd2 = 4.0;
    double Tgt2 = 0.0;
    double Guess2 = 3.0;
    F2 MyF2(3.0);
    NonlinearSolver<F2> solver2(Tgt2, LEnd2, REnd2, Acc2, Guess2);
    cout << "Root of F2 by Bisect: " << fixed << setprecision(4)
         << solver2.SolveByBisect(&MyF2) << endl;
    cout << "Root of F2 by Newton-Raphson: " << fixed << setprecision(4)
         << solver2.SolveByNR(&MyF2) << endl;
    return 0;
}
/*
Root of F1 by Bisect: 1.4142
Root of F1 by Newton-Raphson: 1.4142
Root of F2 by Bisect: 3.1623
Root of F2 by Newton-Raphson: 3.1623
*/
```

Notes:

- Function has become a template parameter. The Function class is gone. ***SolveByBisect()*** and ***SolveByNR()*** have become ***template functions.***
- ***F1*** and ***F2*** are no longer derived classes. NonlinearSolver objects are created for F1 and F2 correspondingly. When compiling the code, there will be two different versions of SolveByBisect(), one for F1 and the other for F2, as well as two version of SolveByNR().
- If more functions were involved, it can result in long compile time and large .exe file.

Multiple Inheritance



Example

```
#include <iostream>
class Student {
public:
  int GetAge() const;
  int GetId() const;
  double GetGPA() const;
  void SetAge( int age_ );
  void SetId( int id_ );
  void SetGPA( double gpa_ );
private:
  int age;
  int id;
  double gpa;
};
```

Example

```
class Employee {
public:
  int GetAge() const;
  int GetId() const;
  double GetSalary() const;
  void SetAge( int age_ );
  void SetId( int id_ );
  void SetSalary( double salary_ );
private:
  int age;
  int id;
  double salary;
};
```

Example

```
class GradAssistant :public Student, public Employee {
public:
  void Display() const;
};

void GradAssistant::Display() const
{
      cout  << GetGPA() << ','
            << GetSalary() << endl;
}
```

Example

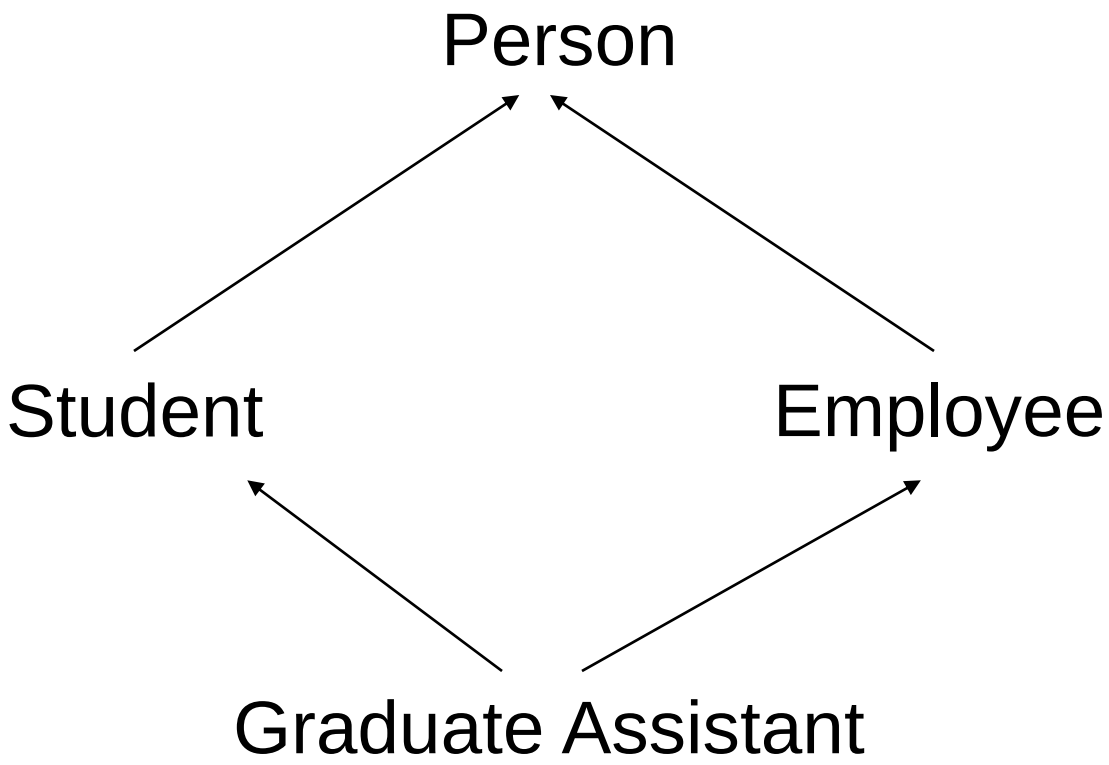
```
int main()
{    GradAssistant GA;
     //GA.SetId(12345);                  // error: ambiguous
      GA. Employee::SetId(12345);      // ok – but potential problem
     // GA.SetAge(22);                   // error: ambiguous
     GA.Student::SetAge(22);           // ok – but potential problem
     GA.SetGPA(3.5);
     GA.SetSalary(10000.00);
     GA.Display();
     return 0;
}
```

Example

Question:

- If we want to set a GradAssistant ‘s age by calling SetAge(), which SetAge() should we use ?
  1. Direct solution: Student::SetAge() or Employee::SetAge(), but may cause data consistency issue. Not an ideal solution
  2. Similar issue with id.
  3. Best solution: Virtual Inheritance

Virtual Inheritance



Example

```
#include <iostream>
class Person {
public:
  int GetAge() const;
  int GetId() const;
  void SetAge( int age_ );
  void SetId( int id_ );
private:
  int age;
  int id;
};
```

Example

```
class Student : public virtual Person {
public:
  double GetGPA() const;
  void SetGPA( double gpa_ );
private:
  double gpa;
};
```

Example

```
class Employee :public virtual Person {
public:
  double GetSalary() const;
  void SetSalary( double salary_ );
private:
  double salary;
};
```

```
class GradAssistant :public Student, public Employee {
public:
    void Display() const;
};
void GradAssistant::Display() const
{    cout << GetId() << ',' << GetAge() << ',' <<
          << GetGPA() << ',' << GetSalary() << endl;
          // no ambiguous
}
int main()
{    GradAssistant GA;
     GA.Display();
     return 0;
}
```

Virtual Base Class (virtual inheritance)

- Two function calls GetId() and GetAge() in GradAssistant::Display() are ambiguous unless Person is inherited as a virtual base class.
- Adding “virtual” lets the compiler decide which function and which variable should be accessed.

Computing Implied Volatility

- The normal distribution can be implemented using the following rational function approximation:
- For x ≥ 0

$$N(x) \approx 1 - \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} (a_1 k + a_2 k^2 + a_3 k^3 + a_4 k^4 + a_5 k^5),$$

- For x < 0

$$N(x) \approx 1 - N(-x)$$

- Where $k = (1+\gamma x)^{-1}$, and where γ, a_1, a_2, a_3, a_4, a_5 are suitably chosen constant.

- To compute the implied volatility using Newton-Raphson solver, we also need an expression for the derivative of the European call option price with respect to volatility σ. This derivative, called **vega**, is given by:

$$v = \frac{1}{\sqrt{2\pi}} S(0) \exp(\frac{-d_+^2}{2})$$

EurCall.h

```
#pragma once
#include "NonlinearSolver02.h"
namespace fre {
   class EurCall
   {
   private:
      double T, K;
   public:
      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);
   };
```

EurCall.h (Continue)

```
class Intermediary : public EurCall, public Function
{
private:
    double S0, r;
public:
    Intermediary(double S0_, double r_, double T_, double K_)
            : EurCall(T_, K_) , S0(S0_), r(r_)  { }
    double Value(double sigma);
    double Deriv(double sigma);
};
}
```

EurCall.cpp

```
#include "EurCall.h"
#include <cmath>
namespace fre {
   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);
   }
```

EurCall.cpp (Continue)

```
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);
}
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);
}
```

EurCall.cpp (Continue)

```
double Intermediary::Value(double sigma)
{
    return PriceByBSFormula(S0, sigma, r);
}
double Intermediary::Deriv(double sigma)
{
    return VegaByBSFormula(S0, sigma, r);
}
}
```

Notes:

- The ***EurCall*** class contains the variables T, K, a constructor function, member functions for calculating option price, vega, as well as auxillary functions to compute d_+ and d_-.
- The distribution function ***N(x)*** is computed using rational function approximation. N(x) is an independent function.
- The function ***PriceByBSFormula()*** computes the option price from the Black-Scholes formula.
- ***VegaByBSFormula()*** computes the vega parameter.

Notes (continue):

- The class ***Intermediary*** inherits from both the ***EurCall*** and ***Function*** classes, such as

  class Intermediary : public EurCall, public Function

  Therefore, the derived class ***Intermediary*** inherits the member functions of both parent classes, except the constructors and destructor – ***Multiple Inheritance***.

- The derived class ***Intermediary*** translates ***PriceByBSFormula()*** and ***VegabyBSFormula()*** into ***Value()*** and ***Derive()*** functions.
  - The constructor function of the ***Intermediary*** class calls the constructor of the EurCall class to initialize T, K, and then initialize its own private data S0 and r.
  - Value() is connected to ***PriceByBSFormula()*** .
  - Derive() is connected to ***VegabyBSFormula()***.

main4.cpp

```
#include <iostream>
#include <iomanip>
#include "Function02.h"
#include "NonlinearSolver02.h"
#include "EurCall.h"
using namespace std;
using namespace fre;
int main()
{
   double S0 = 100.0;
   double r = 0.1;
   double T = 1.0;
   double K = 100.0;

   Intermediary Call(S0, r, T, K);
```

main4.cpp (continue)

```
    double Acc = 0.0001;
    double LEnd = 0.01, REnd = 1.0;
    double Tgt = 12.56;
    double Guess = 0.23;
    NonlinearSolver solver(Tgt, LEnd, REnd, Acc, Guess);
    cout << "Implied Volatility by Bisect: " << fixed << setprecision(4)
         << solver.SolveByBisect(&Call) << endl;
    cout << "Implied Volatility by Newton-Raphson: " << fixed << setprecision(4)
         << solver.SolveByNR(&Call) << endl;
    return 0;
}
/*
Implied Volatility by Bisect: 0.1784
Implied Volatility by Newton-Raphson: 0.1784
*/
```

Notes

- In the main function:
  - ***Tgt*** is initialized with the observed call price $C_{quote.}$
  - Implied volatility is calculated by bisection solver and the New-Raphson solver.

Homework Assignment

- The ***yield*** y of a coupon with face value F, maturity T, and fixed coupon C_1, ….C_N payable at times $0 < T_1 < \ldots < T_n = T$ satisfies:

$$P = \sum_{n=1}^{N} C_n e^{-yT_n} + Fe^{-yT},$$

Where P is the bond price at time 0.

- Using NonlinearSolver02, write a program to compute the yield of a coupon bond by solving the above non-linear equation.

Homework Assignment

- Please see the assignment section for the details of an additional homework assignment on Black & Scholes Option Model.
- The homework questions are based on the Daniel Duffy's book, Introduction to C++ for Financial Engineers, for calculating European Option options with BS formula. We will update his European Option class and use the updated class to solve 2 questions from John Hull's Options, Futures, and Other Derivatives textbook.

References

- Numerical Methods in Finance with C++ (Mastering Mathematical Finance), by Maciej J. Capinski and Tomasz Zastawniak, Cambridge University Press, 2012, ISBN-10: 0521177162
- *Financial Instrument Pricing Using C++, Daniel J. Duffy, ISBN 0470855096, Wiley, 2004.*