FE545 Design, Patterns and Derivatives Pricing

Solvers, Templates, and Implied Volatilities

Steve Yang

Stevens Institute of Technology

steve.yang@stevens.edu

04/02/2024

Overview

The Problem

Function Object

Bisecting with Template

Newton-Raphson and Function Template Arguments

Using Newton-Raphson to Calculate Implied Volatility

Breakout Exercise: Solver for Option Pricer

The Pros and Cons of Templatization

Key Points

- Whatever model one is using to compute vanilla options' prices, it is traditional to quote prices in terms of the Black-Scholes implied volatility.
- The implied volatility is by definition the number to plug into the Black-Scholes formula to get the price desired.
- Thus, we must solve the problem of finding the value σ such that

$$\mathrm{BS}(S, K, r, d, T, \sigma) = \text{quoted price.} \tag{1}$$

- In other words, we must invert the map

$$\sigma \mapsto \mathrm{BS}(S, K, r, d, T, \sigma). \tag{2}$$

- The Black-Scholes formula is sufficiently complicated that there is no analytic inverse and this inversion must be carried out numerically.
- We study two of the simplest: bisection and Newton-Ralphson. Our objective is to illustrate the programming techniques for defining the interfaces in a reusable fashion rather than to implement the most efficient algorithm available.
- Given a function, f of one variable we wish to solve the equation

$$f(x) = y \tag{3}$$

- In the above, f is the Black-Scholes formula, x is volatility and y is the price. IF the function f is continuous, and for some a and b we have

$$f(a) < y,$$
$$f(b) > y,$$

then there must exist some c in the interval (a, b) such that $f(c) = x$. Bisection is one technique to find c.

Bisection Method

- The idea is straightforward: we simply take the midpoint m, of the interval, then one of three things must occur:
  1). $f(m) = y$ and we are done.
  2). $f(m) < y$ in which case there must be a solution in (m, b);
  3). $f(m) > y$ in which case there must be a solution in (a, m).
- Thus by taking the midpoint, we either find the solution, or halve the size of the interval in which the solution exists. We terminate when we achieve

$$|f(m) - y| < \epsilon, \tag{4}$$

for some pre-determined tolerance accuracy ϵ.

Bisection Method

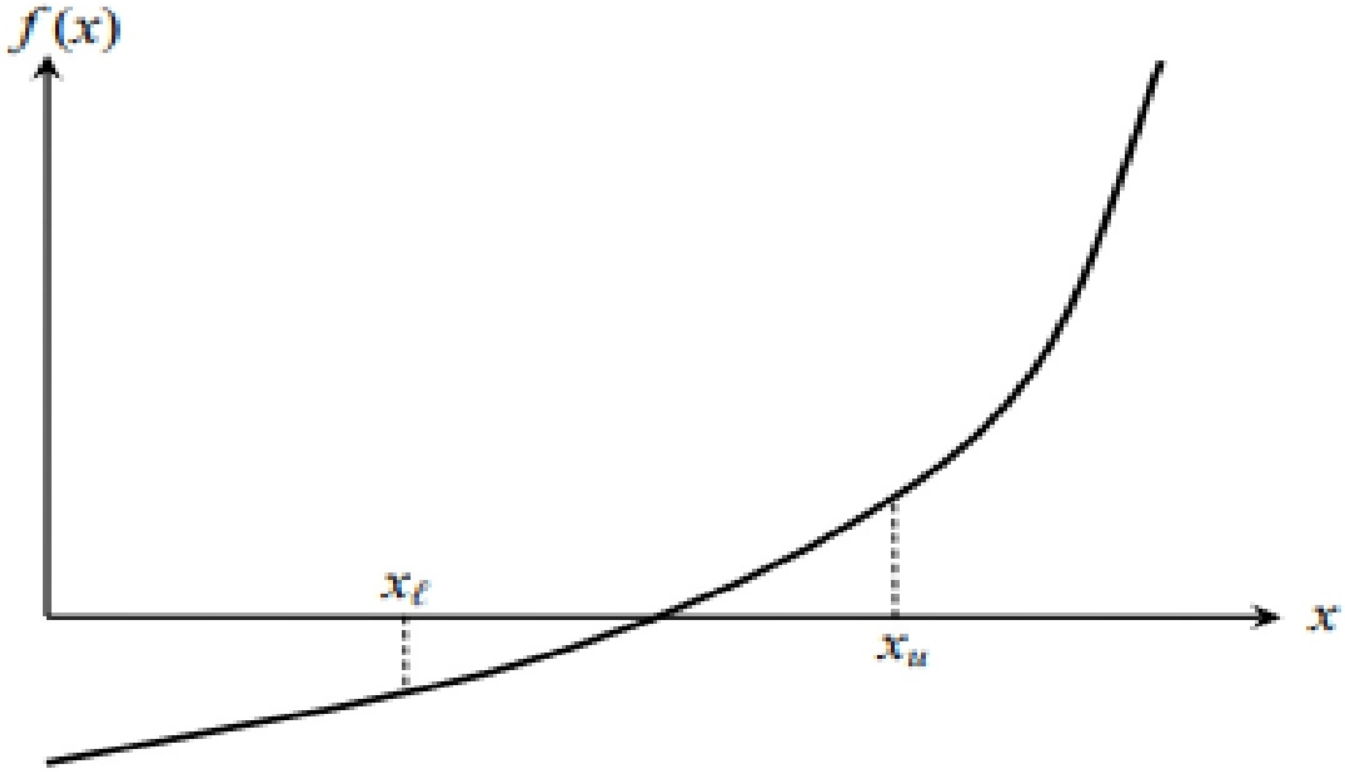

Figure: The Bisection for Solving $f(x) = 0$ between x_l and x_u.

Newton-Raphson Method

- When we have a well-behaved function with an analytic derivative then Newton-Ralphson can be much faster.
- The idea of Newton-Raphson is that we pretend the function is linear and look for the solution where the linear function predicts it to be. Thus we take a standing point x_0, and approximate f by

$$g_0(x) = f(x_0) + (x - x_0)f'(x_0). \tag{5}$$

- We have that $g_0(x)$ equal to zero if and only if

$$x = \frac{y - f(x_0)}{f'(x_0)} + x_0. \tag{6}$$

- We therefore take this value as our new guess x_1. We repeat until we find that $f(x_n)$ is within ϵ of y.

Newton-Raphson Method

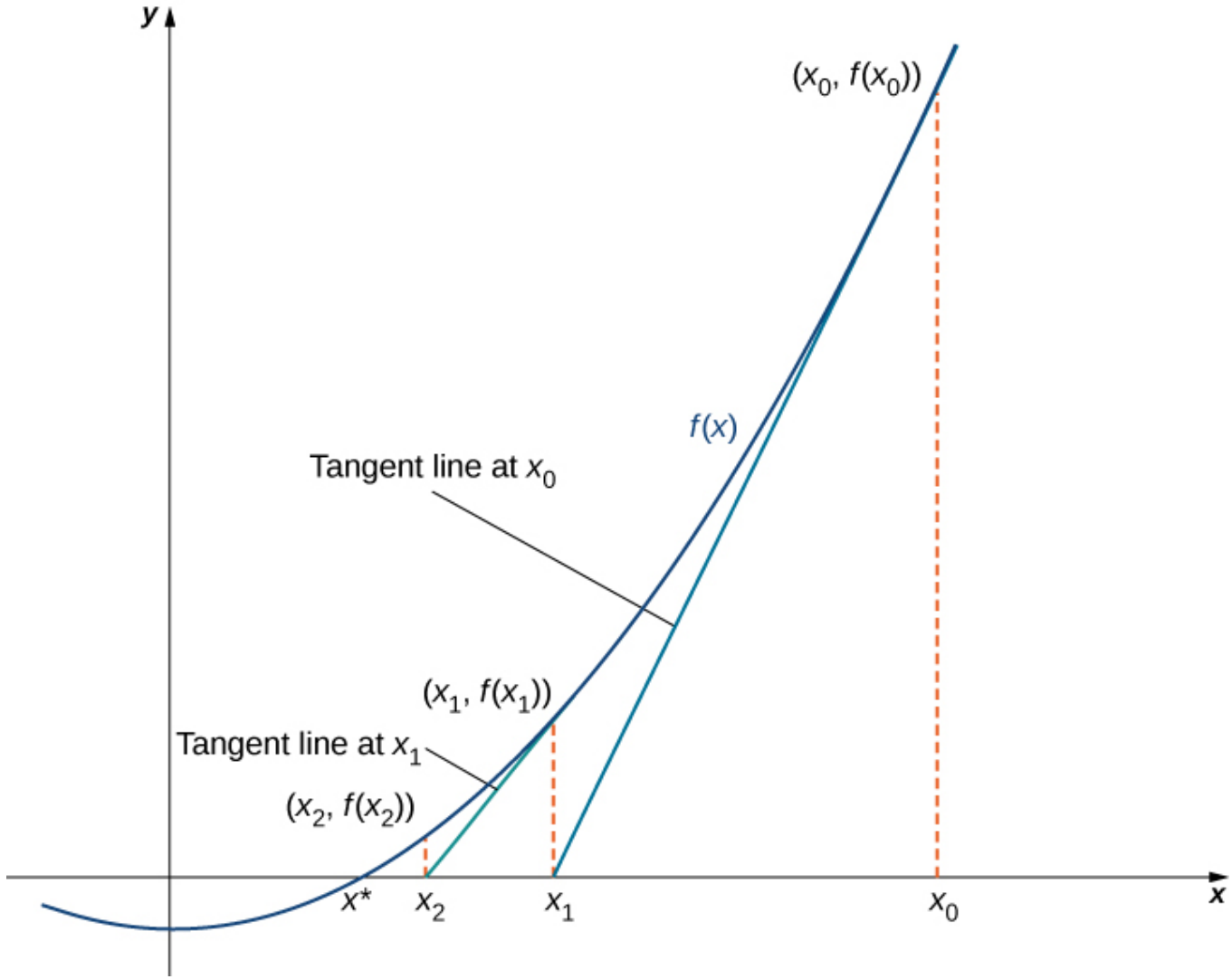

Figure: The Newton Raphson for Solving Equations of the Form $f(x) = 0$.

Function Object

- We want to implement the bisection algorithm in a re-usable way; this means that we will need a way to pass the function f into the bisection routine.
- There are many different ways to tackle this problem:
  - The engine template with which we define a base class for which the main method is to carry out the bisection.
  - A function pointer but his would buy us little over virtual functions.
  - Templatization in which the type of the function used in the optimization is decided at compile time rather than run time.
  - The function object is an object for which *operator*() is defined.
- So if we have an object f of a class T for which *const operator*() (*doublex*) *const* has been defined it is then legitimate to write $f(y)$ for a double y, and this is equivalent to $f.operator()(y)$.
- Thus our object f can be used with function-like syntax. However, as f is an object it can contain extra information.

Function Object Class

BSCallClass.h

```
#ifndef BS_CALL_H
#define BS_CALL_H

class BSCall
{
public:
    BSCall(double r_, double d_,
           double T_, double Spot_,
           double Strike_);
    double operator() (double Vol) const;
private:
    double r;
    double d;
    double T;
    double Spot;
    double Strike;

};
```

Function Object Class

BSCallClass.cpp

```
#include "BSCallClass.h"
#include "BlackScholesFormulas.h"


BSCall::BSCall(double r_, double d_,
               double T_, double Spot_,
               double Strike_)
:
r(r_),d(d_),
T(T_),Spot(Spot_),
Strike(Strike_)
{}

double BSCall::operator()(double Vol)const
{
    return BlackScholesCall(Spot,Strike,r,d,Vol,T);
}
```

Bisecting with Template

- In the previous section, we showed how we could define a class for which the syntax $f(x)$ makes sense when f was an object of the class, and x was a double.
- The basic idea of templatization is that you can write code that works for many classes simultaneously provided they are required to have certain operations defined with the same syntax.
- Our requirement is that the class should have:

  double operator() (*double*) *const*

  and thus the syntax $f(y)$ is well-defined for class objects as we discussed above.

Function Object Class

Bisection.h

```
#include <cmath>

template<class T> // class T is a function object, T(args) behave like a normal
double Bisection(double Target,
                 double Low,
                 double High,
                 double Tolerance,
                 T TheFunction)
{
    double x = 0.5 * (Low + High);
    double y = TheFunction(x); // compute function value at mid

    do
    {
        if (y < Target)
            Low = x;

        if (y > Target)
            High = x;
        x = 0.5*(Low + High);

        y = TheFunction(x);
    }
```

Newton-Raphson and Function Template Arguments

- We now want to adapt the pattern we have presented to work for Newton-Raphson as well as for bisection.
- One solution would be simply to pass in two function objects: one for the value and another for the derivative. This is unappealing, however, in that we would then need to initialize a set of parameters for each object and we would have to be careful to make sure they are the same.
- We assume a name for the derivative function. After all, that is essentially what we did for the value function; it was a special name with special syntax but ultimately it was just assuming a name. *double Derivative*(*double*) *const*

  defined and at appropriate points in our function we would then put *TheFunction.Derivative*(x)
- This would certainly work. However, it is a little ugly and if our class already had a derivative defined under a different name, it would be annoying.

Newton-Raphson and Function Template Arguments

- Fortunately, there is a way of specifying which class member function to call at compile time using templatization.
- The key to this is a **pointer to a member function**. A pointer to a member function is similar in syntax and idea to a function pointer, but it is restricted to methods of a single class.
- The difference in syntax is that the class name with a :: must be attached to the $*$ when it is declared.
- Thus to declare a function pointer called *Derivative* which must point to a method of the class T, we have
  double (T::*Derivative) (double) double
- The function *Derivative* is a const member function which takes in a double as argument and outputs a double as return.
- If we have an object of class T called *TheObject* and y is a double, then the function pointed to can be invoked by $TheObject.*Derivative(y)$.

Function Object Class

NewtonRaphson.h

```
#include <cmath>

//three template parameters
template<class T, double (T::*Value)(double) const,double (T::*Derivative)(doub
    double NewtonRaphson(double Target,
                         double Start,
                         double Tolerance,
                         const T& TheObject)
{
    double y = (TheObject.*Value)(Start); // pass initial guess to function
    double x=Start;

    while ( fabs(y - Target) > Tolerance )
    {
        double d = (TheObject.*Derivative)(x);
        x+= (Target-y)/d;
        y = (TheObject.*Value)(x); // check the target function value
    }

    return x;

}
```

Using Newton-Raphson to Calculate Implied Volatility

- Now that we have developed a Newton-Raphson routine, we want to use it to compute implied volatilities.
- Our class will therefore have to support pricing as a function of volatility and the vega as a function of volatility.
- As before, the other parameters will be class data members which are not imputed in the constructor rather than via these methods.
- We present a suitable class in *BSCallTwo.h* and *BSCallTwo.cpp*

Function Object Class

BSCallTwo.h

```
class BSCallTwo
{
public:
    BSCallTwo(double r_, double d_,
              double T, double Spot_,
              double Strike_);

    double Price(double Vol) const;
    double Vega(double Vol) const;

private:
    double r;
    double d;
    double T;
    double Spot;
    double Strike;
};
```

Function Object Class

BSCallTwo.cpp

```
#include "BSCallTwo.h"
#include "BlackScholesFormulas.h"

BSCallTwo::BSCallTwo(double r_, double d_,
                     double T_, double Spot_,
                     double Strike_)
:
r(r_),d(d_),
T(T_),Spot(Spot_),
Strike(Strike_)
{}


double BSCallTwo::Price(double Vol) const
{
    return BlackScholesCall(Spot,Strike,r,d,Vol,T);
}

double BSCallTwo::Vega(double Vol) const
{
    return BlackScholesCallVega(Spot,Strike,r,d,Vol,T);
}
```

Breakout Exercise: Create a Solver for Option Pricer

- Use the classes introduced in class and finish the main.cpp file to price European Option with the following parameters:
  - Expiry: 1
  - Spot: 50
  - Strike: 50
  - Risk free rate: 0.05
  - Dividend: 0.08
  - Price: 4.23
  - Vol low guess: 0.2
  - Vol high guess: 0.4
  - Vol initial guess: 0.23
  - Tolerance: 0.01
- Use the Bisection class from the sample starting point.

The Pros and Cons of Templatization

- There are advantages and disadvantages to each approach. The principal difference is that for templates argument types are decided at the time of compilation, whereas for virtual functions argument type is not determined until runtime.
  1. The first is speed. No time is spent on deciding which code to run when the code is actually running.
  2. The second is code size. As the code is compiled for each template argument used separately, we have multiple copies of very similar code.
  3. Templatization is harder for the user of the code to make choices. If the user was allowed to choose, we would have to write code to deal with a large number of cases.
  4. Templatization is harder to debug. Compilers will often not actually compile lines of template code that are not used.
- One way to solve the problem is first to write non-template code for a particular choice of the template parameter. This code can be thoroughly tested and debugged, and then afterwards the code can be rewritten by changing particular parameter into a template parameter.

Key Points

- Templates are an alternative to inheritance for coding without knowing an object's precise type.
- Template code can be faster as function calls are determined at compile time.
- Extensive use of template code can lead to very large executables.
- Pointers to member functions can be a useful way of obtaining generic behavior.
- Implied volatility can only be computed numerically.