Computational Finance with C++

Lecture 4: Non-linear Solvers

Panos Parpas
Imperial College London
p.parpas@imperial.ac.uk

Outline

- The role of non-linear solvers in finance
- Bisection methods & Newton-Raphson method
- C++ implementations:
    - ✦Function pointers
    - ✦Virtual Functions
    - ✦Function templates
- Discuss advantages/disadvantages of the different implementations
- Reading:
    - ✦**Chapter 4** Capinski+Zastawniak,Numerical Methods in finance with C++

Reminder: Implied Volatility

Black Scholes Model for European Call Option, expiry at T, strike K

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

Where

$$d_+ = \frac{\ln(S(0)/K) + (r + \sigma^2/2)/T}{\sigma\sqrt{T}} \quad N(d) = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} e^{-y^2/2} dy$$

$$d_- = d_+ - \sigma\sqrt{T}$$

All parameters are specified in contract or observed, apart from volatility

Can estimate volatility (called implied volatility) from observed option prices

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

Bisection Method

- General method (1-d) to solve for x $\text{in } [a, b] \in \mathbb{R}$

$$f(x) = c$$

$f(a) - c,\ f(b) - c,\ \text{have opposite signs}$

$\implies \text{there exists an } x \in [a, b] \text{ such that } f(x) = c$

Bisection Method

Works by constructing left and right approximations of the solution

0 Let $l_0 = a$, $r_0 = b$,$n = 0$

1-a If $f(l_n) - c$ and $f(\frac{1}{2}(l_n + r_n)) - c$ have opposite signs

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

1-b Otherwise

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

c Set $n \leftarrow n + 1$ go to 1-a

Can be shown $l_n \uparrow x$, and $l_n \downarrow x$ as $n \to \infty$ and $f(x) = c$

Newton-Raphson Method

Function is assumed to be differentiable on [a,b]

0 Let $x_0 \in (a, b)$

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

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

Can be shown if x_0 close enough to a solution ($f(x) - c = 0$) then $x_n \to x$

Implementation with Function Pointers:`Solver01.h`, `Main15.cpp`

- `SolveByBisect()` takes function pointer as argument and other problem parameters as arguments (a,b,c)
- Note efficient function evaluations
- Termination criterion is the size of the interval

- `SolveByNR()` implements Newton-Raphson solver
- Two function pointers required, for function and its derivative
- Also needs a starting point and the target value
- Termination criterion is change in current solution

Implementation with Function Pointers:`Solver01.h, Main15.cpp`

Advantages:

- Simple and direct implementation
- Generally quite fast

Disadvantages:

- Difficult to expand/modify e.g. add more parameters

Implementation with Function Pointers:`Solver01.h, Main15.cpp`

Advantages:

- Simple and direct implementation
- Generally quite fast

Disadvantages:

- Difficult to expand/modify e.g. add more parameters

Implementation withVirtual Functions:`Solver02.h, Main16.cpp`

- Abstract class `Function` to represent function we want to solve
- Virtual functions `Value()` and `Deriv()` return function and derivative value
- `Fct` is now a pointer to a *class* not a pointer to a *function.*
- Note use of –> to deal with pointers to classes
- Concrete functions `F1` and `F2` are subclasses if Function
- Note definition of parameter in `F2`
- Note that correct pointers to subclasses are passed on to, checking will be done in real time

Implementation with Function Templates:`Solver03.h, Main17.cpp`

- Some checking will need to be done at run time with the virtual function implementation (can add significant overhead if inside a loop)
- Function template solution means that checking can be done at compile time
- The function class is replaced by template parameter
- Functions are replaced by template functions
- Easy for the compiler to decide if `F1` or `F2` needs to be used.
- Compiled executable can be larger and take longer to compile

Exercise 4.1: Alter the code in `Solver03.h` and `Main17.cpp` to have an object of type `Function` passed by reference to `SolverByBisect()` and `SolveByNR()` rather than having a pointer to an object of type `Function` passed to them.

Exercise 4.2: The **yield** y of a coupon bond with face value F, maturity T, and fixed coupons $C_1, \ldots C_N$ payable at times $0 < T_1 < \ldots < T_N = T$,satisfies

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

where $P$is the bond price at time 0.
Using Solver03.h, write a program to compute the yield y of a coupon bond by solving the nonlinear equation above.

Exercise 4.3: Rewrite the code for numerical integration in Exercises 2.1 and 2.3 replacing function pointers/virtual functions by templates.

Exercise 4.4: Rewrite the code for option pricing in `Options09.h, Options09.cpp` and `Main14.cpp`, replacing virtual functions by templates.

Computing Implied Volatility: `EurCall.h`, `EurCall.cpp`, `Main18.cpp`

- Normal Distribution is approximated with:

$$N(x) = \begin{cases} 1 - \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}(a_1 k + a_2 k^2 + a_3 k^3 + a_4 k^4 + a_5 k^5) & \text{if } x \geq 0 \\ 1 - N(-x) & \text{otherwise} \end{cases}$$

where $k = (1 + \gamma x)^{-1}$, the other constants are specified in the code

Newton-Rahpson also needs to use the derivative of the Black and Scholes formula with respect to the volatility (a Greek known as vega)

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

Remarks on Templates

- Programming with templates can be hard.
- Debugging does not work very well
- Compilers give error messages that don’t make sense
- They can speed up coding a lot, if used correctly.
- One idea is to write a non-template version of the code, and make sure it works
- Write good documentation