Root Searcher and Implied Volatility Surface

Numerical Methods in Quantitative Finance

Li Hao
matv113@nus.edu.sg

Black-Scholes Model

- We have been dealing with the Black-Scholes model

$$\frac{dS}{S} = (r - q)dt + \sigma dW \tag{1}$$

- Constant drift r and volatility σ
- If the model indeed describes how the market behaves, any derivative product based on the underlying asset S is *almost* risk-free
- Unfortunately the market does not behave so
- Does it make Black-Scholes model useless? No.

- Vanilla options are liquid products in the market
- The market agrees on the prices of them for certain maturities and certain strikes
- Financial institutes, or market makers, receive requests for options of any strike and any maturity
- Maintaining directly their prices is not quite convenient since it depends highly on the underlying asset's spot price which is moving every second

- Black-Scholes model comes handy in this situation
- It provides with formula for option prices

$$V_{call}(K, S, \sigma, T, r) = SN(d_+) - e^{-rT} KN(d_-) \tag{2}$$

where

$$d_\pm = \frac{\ln \frac{S}{K} + (r - q)T \pm \frac{1}{2}\sigma^2 T}{\sigma\sqrt{T}}$$

- In the formula, everything are known from the market and trade except the "volatility".
- If we know the price of the call option from the market, we are able to back out the corresponding volatility, and this is called Black-Scholes **implied volatility**

- Given all the other variables, we can look at the Black-Scholes formula as a function from volatility σ to the call option price V:

$$f_{\{S,r,q,K,T\}}(\sigma) = V$$

- So the implied volatility is

$$\sigma = f^{-1}(V)$$

- The call option price is a monotonic function to the volatility — time value due to volatility increases when σ increases, when $T > 0$
- However, f^{-1} has no closed form, (2) is an implicit function of σ
- We need to resort to a **root finding algorithm** to back out the volatility from call option prices

Root Finding

Given a function f, find the x such that

$$f(x) = 0 \tag{3}$$

- The problem can be one dimensional or N dimensional.
- One dimensional problem is easier and there are robust algorithms
- N dimensional problem is much more difficult and require a lot of insight of the function itself
- We mainly deal with one dimensional problem

Bracketing The Root

- Most root finding algorithms require a bracket that contains the root
- An interval (a, b) brackets the root if $f(a)$ and $f(b)$ have opposite signs, provided that the function $f(x)$ is **continuous**
- We can bracket the root by starting with an initial interval, expanding it outward or inward until $f(a)f(b) <= 0$

Root Bracketing Algorithm

Algorithm 1 (a, b) = RootBracketing (f, a, b, maxIter, $factor$)

1: **for** $k = 1$ to maxIter **do**
2: **if** $f(a)f(b) < 0$ **then**
3: return (a, b)
4: **end if**
5: **if** $abs(f(a)) < abs(f(b))$ **then**
6: $a+ = factor * (a - b)$ // if $f(a)$ is closer to 0, change a
7: **else**
8: $b+ = factor * (b - a)$ // if $f(b)$ is closer to 0, change b
9: **end if**
10: **end for**

- if $factor > 0$ the algorithm searches outward
- if $-1 < factor < 0$ the algorithm searches inward

Implementation — Through Function Pointer

- The input of the algorithm is a function, how do we implement this in C++?
- One way is to pass a function as an argument through `function pointer`
- `function pointer` is a pointer to functions

```
returnType (*fp)(argType1, argType2, ...); // function pointer declaration
```

- It means the function pointer `fp` points to functions taking `argType1, argType2, ...`, and returns `returnType`
- Very similar to function declaration, just add `*` in front of function name

Function Pointer — Example

- The function name stores its address:

```
1 int foo(int a, int b) {return a + b;}
2
3 int main()
4 {
5   // function name stores the address of the function
6     std::cout << "foo = " << foo << std::endl;
7     std::cout << "foo = " << (void *) foo << std::endl;
8 }
```

- So we can pass the address to a function pointer, then call it through dereferencing the pointer:

```
1 int main()
2   // assigning the address of the function foo to fp
3     int (*fp)(int, int) = foo;
4   // call function foo(5, 9) through fp.
5     std::cout << (*fp)(5, 9) << std::endl;
6     return 0;
7 }
```

Root Bracketing Implementation

```
bool RootBracketing(double (*fp)(double), double &a, double &b)
{
  const int NTRY=50;
  const double FACTOR=1.6;
  if (a >= b) throw("wrong input a and b in RootBracketing");
  double f1 = (*fp)(a);
  double f2 = (*fp)(b);
  for (int j=0;j<NTRY;j++) {
    if (f1*f2 < 0.0) return true;
    if (std::abs(f1) < std::abs(f2))
      f1=(*fp)(a += FACTOR*(a-b));
    else
      f2=(*fp)(b += FACTOR*(b-a));
  }
  return false;
}
```

Root Bracketing Example

```
1 double foo(double x) {return std::exp(x) - 5; }
2
3 int main()
4 {
5   double a = 3.4;
6   double b = 5.78;
7   RootBracketing(foo, a, b);
8   std::cout << a << " " << b << std::endl;
9 }
```

Output:

```
-0.408 5.78
```

Function Objects in C++

- Function pointer is the way of treating functions like variables in C
- The C++ way is to deal with function objects
- Example first:

```
template<typename T>
bool RootBracketingT(T f, double &a, double &b)
{
  const int NTRY=50;
  const double FACTOR=1.6;
  if (a >= b) throw("wrong input a and b in RootBracketing");
  double f1 = f(a);
  double f2 = f(b);
  for (int j=0;j<NTRY;j++) {
    if (f1*f2 < 0.0) return true;
    if (std::abs(f1) < std::abs(f2))
      f1=f(a += FACTOR*(a-b));
    else
      f2=f(b += FACTOR*(b-a));
  }
  return false;
}
```

- The use of `template` is also called **generic programming**

Generic Programming With Templates

In the simplest definition, generic programming is a style of computer programming in which algorithms are written in terms of types **to-be-specified-later** *that are then instantiated when needed for specific types provided as parameters*

— Wikipedia

- Templates are the foundation of generic programming
- Allows writing code in a way that is independent of any particular type
- It's not the first time we use a template:

```
std::vector<double> states;
std::vector<int>    intVector;
```

  All `std::vector`'s member functions, i.e., `push, pop, [], resize` etc, do not care about the actual type of the elements
- Standard template library (STL) is the "most" important library of C++, `vector` is part of it, read [1] for a serious study

- How do we call our root bracketing function that takes template?

```
class MyFunction   // function object
{
public:
  double operator()(double x)
  {
    return exp(x) - 5;
  }
};

int main()
{
  double a = 3.4;
  double b = 5.78;
  MyFunction f;
  RootBracketingT<MyFunction>(f, a, b);
  std::cout << a << " " << b << std::endl;
}
```

Template — Another Example

Example from TutorialsPoint

```cpp
#include <iostream>
#include <string>

template <typename T>
inline T const& Max (T const& a, T const& b)
{
    return a < b ? b:a;
}
int main ()
{
    int i = 39, j = 20;
    cout << "Max(i, j): " << Max(i, j) << endl;

    double f1 = 13.5, f2 = 20.7;
    cout << "Max(f1, f2): " << Max(f1, f2) << endl;

    std::string s1 = "Hello", s2 = "World";
    cout << "Max(s1, s2): " << Max(s1, s2) << endl;

    return 0;
}
```

Templates VS Virtual Function

- Generic programming, like function overloading, is also called static polymorphism
- Templates — static binding: polymorphism achieved at compilation time
  - Faster, but might result in lager size of executable or library file
- Virtual function — dynamic binding: polymorphism achieved at compilation time
  - Runtime overhead for virtual table lookup

Advantage of Function Objects Over Function Pointers

- Our function takes multiple inputs
- We want to pre-apply many of the inputs: spot price, interest rate, maturity, and leave with only volatility
- Not possible with function pointers
- With function objects we can achieve it

Implied Volatility Bracketing With Function Objects

```
class VolToBSPrice
{
public:
  VolToBSPrice(double _spot, double _rate, double _T)
    : spot(_spot), rate(_rate), T(_T) {}
  // vol to price function
  double operator()(double vol)
  {
    return blackShoclesPrice(spot, rate, strike, Call, T);
  }
private:
  double spot;
  double rate;
  double strike;
  double T;
};

int main()
{
  VolToPrice f(100, 0.02, 90, 1.0);
  RootBracketingT<VolToPrice>(f, a, b);
  std::cout << a << " " << b << std::endl;
}
```

Functional Programming

- Functional programming — a programming paradigm that treats computation as the evaluation of mathematical functions
- The output value of a function depends purely on the input, not order of execution, no side effect
- Functional programming roots with *lambda calculus*

```
\(x, y) -> x + y
```

- Very convenient to translate mathematical function in to code, popular functional programming language — Haskell
- C++11 starts to support functional programming

Functional Programming in C++11

- Header file

```
#include <functional>
```

- Main object (type): `std::function`, its declaration in `functional.h`:

```
template< class R, class... Args >
class function<R(Args...)>
```

- Example

```
1  double foo(double x) {return std::exp(x) - 5; }
2  std::function<double(double)> fun = foo;
3  std::cout << fun(5) << std::endl;
```

std::function — The Convenience

- An `std::function` can refer to a member function of a class

```
std::function<double(VolToBSPrice&, double)> fun2;
fun2 = &VolToBSPrice::operator();
VolToBSPrice f(100, 0.02, 90, 1.0);
std::cout << fun2(f, 0.04) << std::endl;
```

Not that impressive, we still need to create a class `VolToBSPrice`, and create an instance `f` of it

- With `std::function` you can partially apply your parameters:

```
1 // function partial application
2 std::function<double(double)> fun3 = std::bind(bsPricer, Call, 90, 1.0,
     100, std::placeholders::_1, 0.02);
3 std::cout << fun3(0.04) << std::endl;
```

Note the function `std::bind` and the keyword `std::placeholders::_1`: they mean when using `fun3(x)`, put `x` at the place of `_1`.

- If there are two arguments you would use `std::placeholders::_2`

std::function — More Examples

- It supports lambda expression

```
int main() {
    std::function<double(double)> double_n = [](double x) {return x * 2;};
    std::cout << double_n(50) << std::endl; // prints out 100
}
```

The [] (capture clause) in front of the lambda expression allows to pass local variables to the lambda

```
int main() {
    int sum = 0;
    std::function<void(double)> running_sum = [&sum](double x) {sum += x
     ;};
    running_sum(1);
    running_sum(2);
    running_sum(3);
    //sum is now 6 (1+2+3)
}
```

The auto Keyword

- `auto` is another very convenient feature in C++11
- When the compiler sees `auto` keyword, it will infer its type
- Instead of

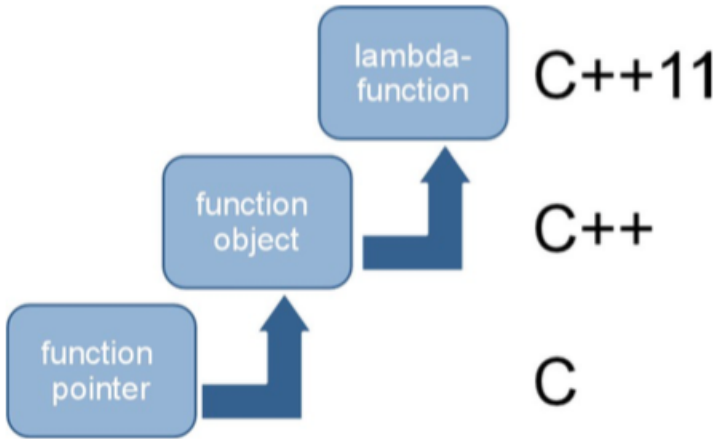
```
1 std::function<double(double)> running_sum = [&sum](double x) {sum += x;};
```

we can write

```
1 auto running_sum = [&sum](double x) {sum += x;};
```

The two statments are equivalent since the function's type can be inferred

- `std::function` is copy-constructible and copy-assignable, can be treated as objects
- Extremely useful in the field of quantitative finance — we are dealing with a lot of mathematical functions
- New features in C++11 almost made C++ a new language



- More details of std::function can be found in the standard and online tutorials

Root Bracketing with std::function

```
bool RootBracketing( std::function<double(double)> f, double &a, double &b
                     , double min=std::numeric_limits<double>::min()
                     , double max=std::numeric_limits<double>::max())
{
  const int NTRY=50;
  const double FACTOR=1.6;
  if (a >= b) throw("wrong input a and b in RootBracketing");
  double f1 = f(a);
  double f2 = f(b);
  for (int j=0;j<NTRY;j++) {
    if (f1*f2 < 0.0) return true;
    if (std::abs(f1) < std::abs(f2))
      f1=f(a = std::max(a + FACTOR*(a-b), min));
    else
      f2=f(b = std::min(b + FACTOR*(b-a), max));
  }
  return false;
}
```

```
  auto fun4 = [](double vol){return bsPricer(Call,90,1.0,100,vol,0.02) - 11;};
  double a = 0.1, b = 0.5;
  RootBracketing(fun4, a, b, 0.0001);
  std::cout << a << " " << b << std::endl;
```

Root Finding Algorithms

Now we are equipped with a powerful implementation tool. Back to our root finding problem

Root Finding

Given a function f, find the x such that

$$f(x) = 0$$

Commonly used root finding algorithms:

- Bisection method
- Secant method
- False position method (regula falsi)
- Brent's method
- Newton method

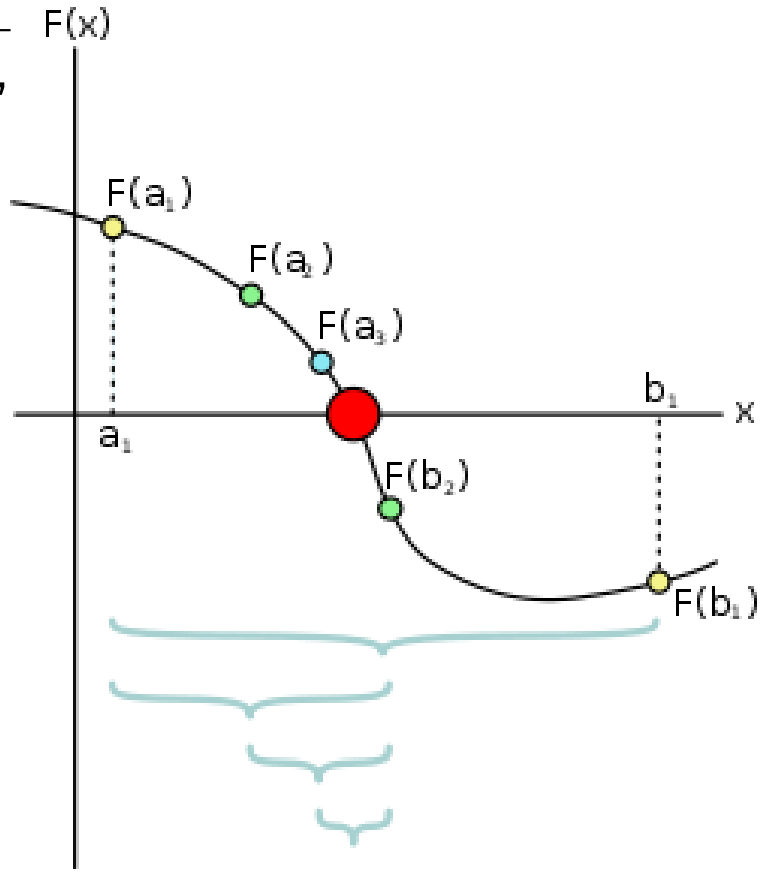
Bisection Method

Start with $[a, b]$ that brackets the root, cut it by half, then see which half contains the root.

Algorithm 2 root = rfbisect($f, a, b,$ *tol*)

Require: $f(a)f(b) < 0$, f is continuous, , tolerance *tol*

1: **while** $(b - a)/2 > tol$ **do**
2: $c \leftarrow \frac{a+b}{2}$
3: **if** $f(c) == 0$ **then**
4: return c;
5: **else if** $f(a)f(c) < 0$ **then**
6: $b \leftarrow c$
7: **else**
8: $a \leftarrow c$
9: **end if**
10: **end while**



Bisection Method — Implementation

Bisection method

```
1  double rfbisect(std::function<
       double(double)>& f, double a,
       double b, double tol)
2  {
3    assert(a < b && f(a) * f(b) < 0);
4    double c;
5    while( (b-a)/2 > tol ) {
6      c = (a+b) / 2.0;
7      if(std::abs(f(c)) < tol)
8        return c;
9      else {
10       if(f(a)*f(c) < 0)
11         b = c;
12       else
13         a = c;
14     }
15   }
16   return c;
17 }
```

Test

```
// bs price for 10% vol
double price = bsPricer(Call, 90,
     1.0, 100, 0.1, 0.02);
auto fun5 = [price](double vol){
     return bsPricer(Call, 90, 1.0,
     100, vol, 0.02) - price;};
std::cout << rfbisect(fun5, a, b, 1e
     -6) << std::endl;
```

Output:

```
(a, b) = (0.0001, 0.5)
(a, b) = (0.0001, 0.25005)
(a, b) = (0.0001, 0.125075)
(a, b) = (0.0625875, 0.125075)
(a, b) = (0.0938312, 0.125075)
(a, b) = (0.0938312, 0.109453)
(a, b) = (0.0938312, 0.101642)
(a, b) = (0.0977367, 0.101642)
(a, b) = (0.0996895, 0.101642)
(a, b) = (0.0996895, 0.100666)
(a, b) = (0.0996895, 0.100178)
(a, b) = (0.0999335, 0.100178)
..
```

Bisection Method — Pro's and Con's

- Easy to understand and implement
- It cannot fail in theory — very robust
  - If the interval happens to contain more than one root, bisection will find one of them.
  - If the interval contains no roots and merely straddles a singularity, it will converge on the singularity.
- Linear convergence:

$$\epsilon_{n+1} = \text{constant} \times \epsilon_n^m, \quad m = 1 \tag{4}$$

or the log of error changes *linearly*, when $m > 1$ the convergence is said to be *superlinear*

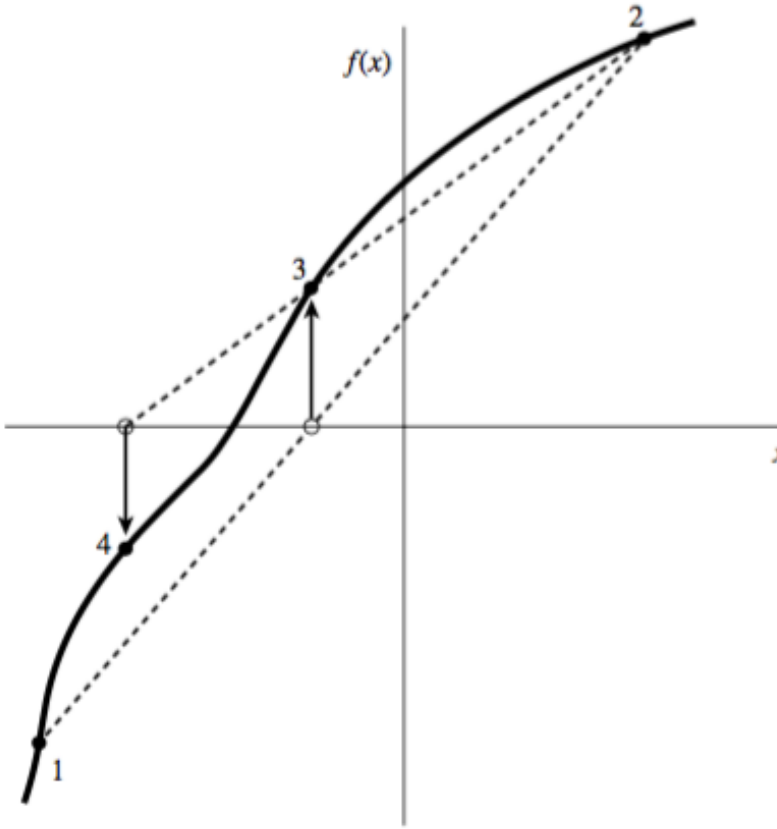
Secant Method

Start with x_1 and x_2, connect $(x_1, f(x_1))$ and $(x_2, f(x_2))$, find the intersection with x axis at x_3, then continue with x_2 and x_3

Algorithm 3 root = rfsecant(f, x_1, x_2, tol)

Require: f is continuous, tolerance tol

1: **while** $abs(x_2 - x_1) > tol$ **do**
2: $\quad x_3 \leftarrow \frac{x_1 f(x_2) - x_2 f(x_1)}{f(x_2) - f(x_1)}$
3: $\quad$ **if** $f(x_3) == 0$ **then**
4: $\quad\quad$ return x_3;
5: $\quad$ **else**
6: $\quad\quad x_1 \leftarrow x_2$
7: $\quad\quad x_2 \leftarrow x_3$
8: $\quad$ **end if**
9: **end while**



Secant Method — Implementation

Secant method

```
1  double rfsecant(std::function<
       double(double)> f, double a,
       double b, double tol)
2  {
3    double c;
4    const int maxIter = 100;
5    int nIter = 0;
6    while( std::abs(a - b) > tol &&
       nIter <= maxIter ) {
7      c = (a * f(b) - b * f(a)) / (f(
       b) - f(a));
8      if(std::abs(f(c)) < tol)
9        return c;
10     else {
11       a = b;
12       b = c;
13     }
14     nIter++;
15   }
16   return c;
17 }
```

Test

```
// bs price for 10% vol
double price = bsPricer(Call, 90,
   1.0, 100, 0.1, 0.02);
auto fun5 = [price](double vol){
   return bsPricer(Call, 90, 1.0,
   100, vol, 0.02) - price;};
std::cout << rfsecant(fun5, a, b, 1
   e-6) << std::endl;
```

Output:

```
(a, b) = (0.0001, 0.5)
(a, b) = (0.5, 0.0178693)
(a, b) = (0.0178693, 0.0350069)
(a, b) = (0.0350069, 58.2528)
(a, b) = (58.2528, 0.345852)
(a, b) = (0.345852, -4.97794)
(a, b) = (-4.97794, -0.0229199)
(a, b) = (-0.0229199, 0.674837)
(a, b) = (0.674837, 0.251415)
(a, b) = (0.251415, 0.130876)
(a, b) = (0.130876, 0.109296)
(a, b) = (0.109296, 0.101454)
(a, b) = (0.101454, 0.100093)
(a, b) = (0.100093, 0.100001)
```

Secant Method — Pro's and Con's

- Open bracket method — does not start with a bracket that contains the root
- Superlinear convergence — convergence rate at $m = 1.62$
- Works very well when the function is locally close to linear
- Convergence is not always guaranteed, need to add `maxIter` in the implementation

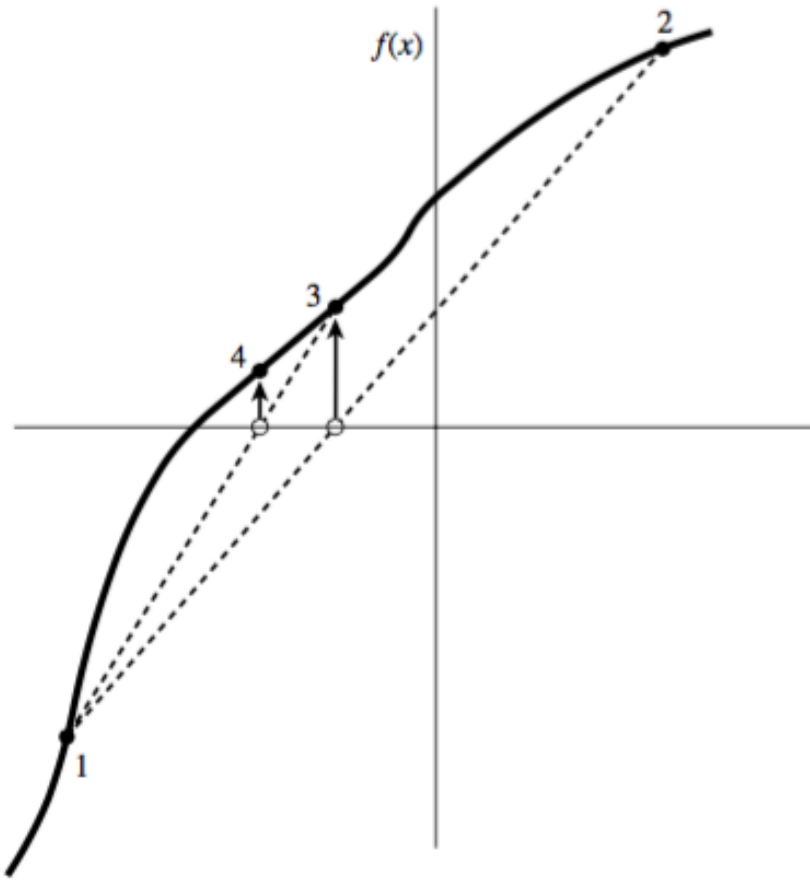
False Position Method (Regula Falsi)

Start with $[a, b]$ that brackets the root, connect $(a, f(a))$ and $(b, f(b))$, find the intersection with x axis at c, then continue with a or b, depending on who brackets the root with c

Algorithm 4 root = rffalsi(f, a, b, tol)

Require: $f(a)f(b) < 0$, f is continuous

while $b - a > tol$ **do**
  $c \leftarrow \frac{af(b)-bf(a)}{f(b)-f(a)}$
  if $f(c) == 0$ **then**
    return c;
  else if $f(a)f(c) < 0$ **then**
    $b \leftarrow c$
  else
    $a \leftarrow c$
  end if
end while



Flase Position Method — Implementation

False position method

```
1  double rffalsi(std::function<double
       (double)> f, double a, double
       b, double tol)
2  {
3    assert(a < b && f(a) * f(b) < 0);
4    double c;
5    while( std::abs(a - b) > tol) {
6      c = (a * f(b) - b * f(a)) / (f(
       b) - f(a));
7      if(std::abs(f(c)) < tol)
8        return c;
9      else {
10       if(f(a)*f(c) < 0)
11         b = c;
12       else
13         a = c;
14     }
15   }
16   return c;
17 }
```

Test

```
rffalsi(fun5, a, b, 1e-6)
```

Output (b does not change?):

```
(a, b) = (0.0001, 0.5)
(a, b) = (0.0178693, 0.5)
(a, b) = (0.0350069, 0.5)
(a, b) = (0.0515307, 0.5)
(a, b) = (0.0670825, 0.5)
(a, b) = (0.0800797, 0.5)
(a, b) = (0.0891336, 0.5)
(a, b) = (0.0944855, 0.5)
(a, b) = (0.0973167, 0.5)
(a, b) = (0.0987226, 0.5)
(a, b) = (0.0993985, 0.5)
(a, b) = (0.0997182, 0.5)
(a, b) = (0.0998683, 0.5)
(a, b) = (0.0999385, 0.5)
(a, b) = (0.0999713, 0.5)
(a, b) = (0.0999866, 0.5)
(a, b) = (0.0999938, 0.5)
(a, b) = (0.0999971, 0.5)
(a, b) = (0.0999986, 0.5)
(a, b) = (0.0999994, 0.5)
(a, b) = (0.0999997, 0.5)
(a, b) = (0.0999999, 0.5)
(a, b) = (0.0999999, 0.5)
```

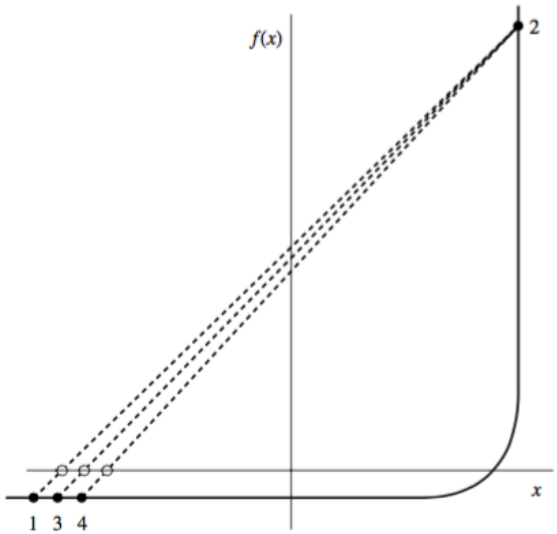
False Position Method — Pro's and Con's

- More robust than secant method — won't shoot outside and root is always bracketed
- If a solution exists it guarantees to find it
- Converges faster than bisection method (usually but not always)
- Often super-linear convergence, but exact order is hard to say

Root Finding Algorithms — Considerations

- Numerical methods are sensitive to
  - Inputs
  - Assumptions
  - The problem itself
  - Tolerance
  - Round-off errors, truncation errors, etc.
- Before applying an algorithm, we need to analyse our problem carefully

Example where both the secant and false-position methods will take many iterations to arrive at the true root. This function would be difficult for many other root-finding methods.



Brent's Method

- We have bisection method that is robust but converges linearly
- Secant, false position method that converge supear-linearly but might hit bad cases
- Natural improvement is to mix them, at each iteration, when Secant or its like work well, use them, otherwise fall back to bisection
- Brent is the most popular one following this logic
- Robust and efficient, normally **the** root finder in practice

Brent's Method

- Brent method is a mixture of
  - Bisection method
  - Secant method
  - Inverse quadratic interpolation: solve the quadratic function $q(y) = x$ that passes the last three iteration points, then pick the $x = q(0)$
- The logic is to use inverse quadratic interpolation when some conditions are satisfied, and bisection method otherwise
- It requires the roots to be bracketed by the input $[a, b]$
- Detailed algorithm can be found on Wikipedia
  https://en.wikipedia.org/wiki/Brent%27s_method

Brent's Method — Implementation

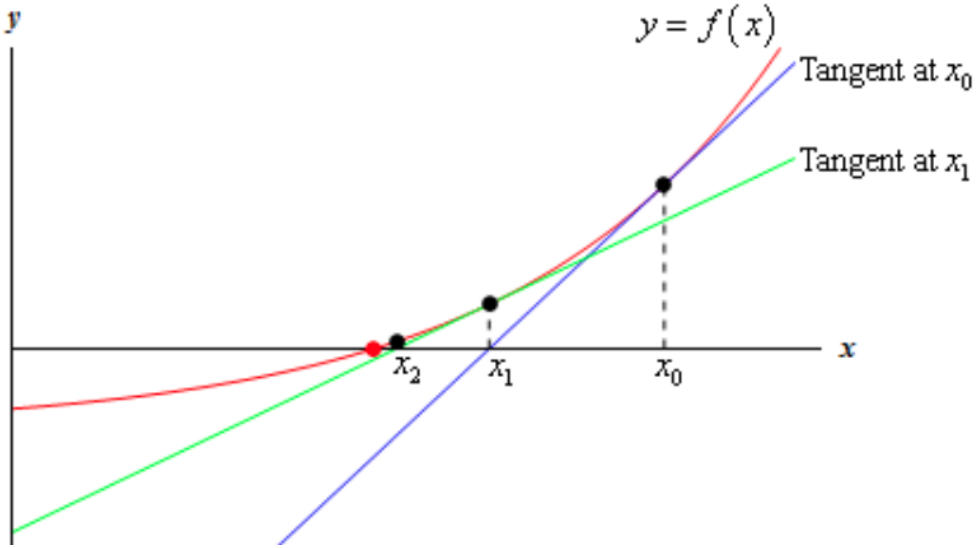
- We use the implementation from Numerical Recipes in C++ [2], just change it to `std::function`, in `RootFinding.cpp`
- Convergence is the best among the tests:

```
(a, b) = (0.0001, 0.0178693)
(a, b) = (0.0178693, 0.258935)
(a, b) = (0.0178693, 0.0410031)
(a, b) = (0.0410031, 0.149969)
(a, b) = (0.0410031, 0.0733168)
(a, b) = (0.0733168, 0.111643)
(a, b) = (0.111643, 0.0969836)
(a, b) = (0.0969836, 0.0997515)
(a, b) = (0.0997515, 0.100001)
(a, b) = (0.100001, 0.1)
(a, b) = (0.1, 0.1)
(a, b) = (0.1, 0.1)
(a, b) = (0.1, 0.100001)
```

Newton's Method

- Newton's method (or Newton-Raphson method) starts with one point instead of two
- At every iteration it updates it with the interception of the tangent line and the x axis:

$$x_{n+1} = x_n - \frac{f(x)}{f'(x)} \qquad (5)$$

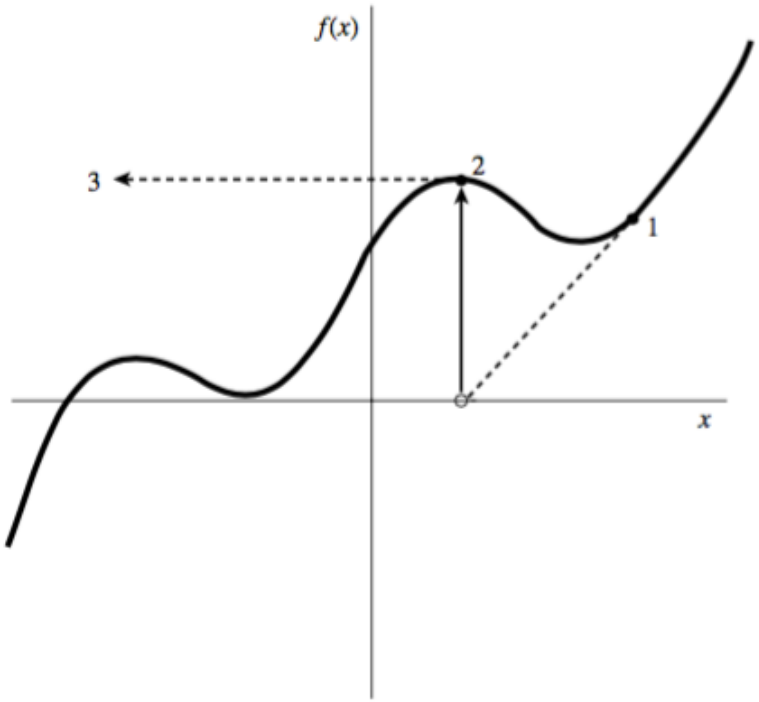


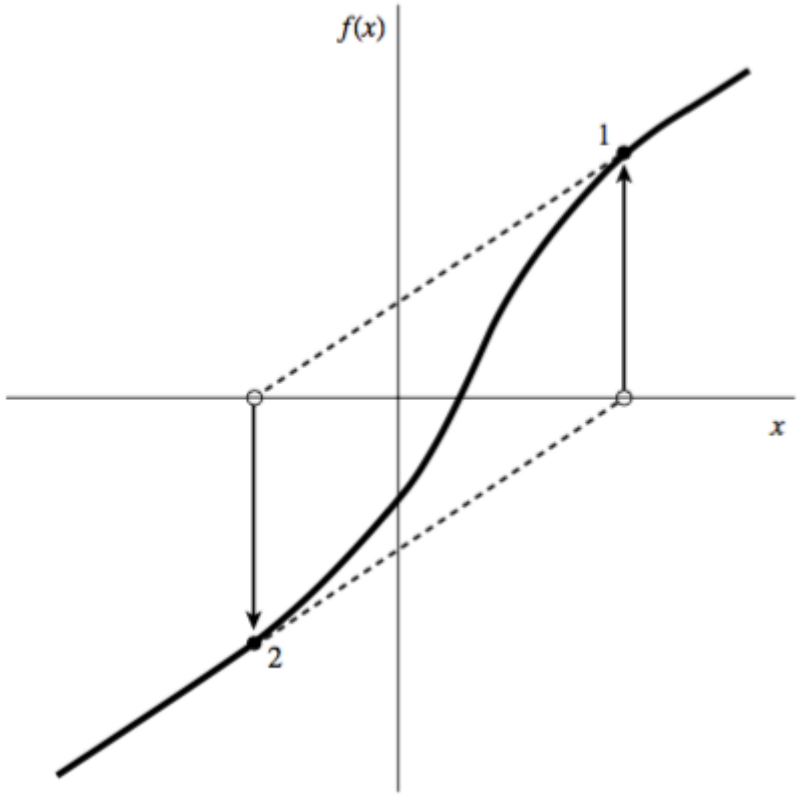
- Coming from the Taylor series expansion, eliminating higher order terms:

$$f(x+\delta) \approx f(x) + f'(x)\delta + \frac{f''(x)}{2}\delta^2 + \dots$$

Newton's Method

- Similar to secant method, where $f'(x)$ is replaced by finite difference $\frac{f(x_n)-f(x_{n-1})}{x_n-x_{n-1}}$
- Possible to go wrong:





- Solution? Mix with bisection method

Newton's Method in Multiple Dimension

- Newton's method can generate to multiple dimensions root search problem:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{J}^{-1}(x)\mathbf{f}(\mathbf{x}) \tag{6}$$

  where $\mathbf{J}(\mathbf{x})$ is the Jacobian matrix at $\mathbf{x}$, whose element J_{ij} is

$$J_{ij} = \frac{\partial f_i}{\partial x_j}$$

- We do not encounter many N dimensional root finding problems in our field
- Instead, we very often have problems on N dimension minimization / maximization — root finding on gradient vectors, but gradients are not arbitrary functions, they are highly restrictive (obey integrability condition) and well formed

Back To Market Making

- Now we are able to convert an option price to its implied volatility
- It's common practice to quote options using implied volatilities
  - Less dependent on change of spot price
  - The implied volatilities do not change as frequently as the spot price changes
- Now assume we have discrete set of liquid maturities, e.g., 1W, 2W, 1M, 2M, 6M, 1Y, etc.
- And for each maturity (or pillar), we have a sample set of strike to option price tuples, say 5 points, and we used our root finder to convert them to strike to implied volatility tuples: (k_i, σ_i)
- We need to construct an implied volatility surface so that we can quote / price vanilla option at any strike and maturity — we need to **interpolate** and **extrapolate** from our sample points

Volatility Smile Interpolation

- The function $k|-> \sigma$ for one maturity is often called volatility smile, due to it's convex shape
- One common way to interpolate volatility smile is to use cubic spline interpolation
- Reason to choose cubic spline:
  - It is C^2 — smooth in first derivative and continuous in second derivative, both within the interval and at the boundaries
  - It's coefficients are fast to compute

Cubic Spline Interpolation

Given sample points $(x_i, y_i), i \in [0, 1, \ldots, N-1]$

- Start with linear interpolation, for $x \in [x_i, x_{i+1}]$

$$y = Ay_j + By_{j+1}, \quad A = \frac{x_{j+1} - x}{x_{j+1} - x_j}, \quad B = 1 - A \tag{7}$$

- Impose second derivatives y''

$$y = Ay_j + By_{j+1} + Cy_j'' + Dy_{j+1}'' \tag{8}$$

$$C = \frac{1}{6}(A^3 - A)(x_{j+1} - x_j)^2, \quad D = \frac{1}{6}(B^3 - B)(x_{j+1} - x_j)^2 \tag{9}$$

- The choice of C and D make the second derivative of y be the linearly interpolated value of y_j'' and y_{j+1}''

$$\frac{d^2y}{dx^2} = Ay_j'' + By_{j+1}'' \tag{10}$$

So second derivative is continuous — our first purpose

Cubic Spline Interpolation

- Our second purpose: first derivative is smooth. It is already the case inside each segment, so we only need the left and right first derivative at each sample point to be equal.
- First derivative in segment $[x_j, x_{j+1}]$ is

$$\frac{dy}{dx} = \frac{y_{j+1} - y_j}{x_{j+1} - x_j} - \frac{3A^2 - 1}{6}(x_{j+1} - x_j)y_j'' + \frac{3B^2 - 1}{6}(x_{j+1} - x_j)y_{j+1}'' \tag{11}$$

- Equalizing $\frac{dy}{dx}$ calculated from left and right segments at the junction points, we get $N - 2$ equations
- For N unknowns y_j'', we are left with two degrees of freedom
  - Setting y_0'' and y_{N-1}'' to 0 — natural cubic spline
  - Calculate y_0'' and y_{N-1}'' such that y_0' and y_{N-1}' equal to specified value

Cubic Spline In Our Application

- We set y'_0 and y'_{N-1} to 0 as boundary conditions because we want our volatility to be flattened beyond the boundary
- We can choose to interpolate σ^2 (variance) or σ (volatility), no particular advantage of either one
- Note that cubic spline can generate negative values — normally does not happen in our ordinary market
- Also note that there is no guarantee that the interpolated implied volatilities may contain arbitrage — problem for pegged currencies

Time Interpolation

- Now for each maturity we have a volatility smile
- When we need to evaluate options expiring not exactly on a pillar date, we need to interpolate our implied volatility in time
- Time interpolation is easy, it can be linear on variance along the strike line (moneyness line to be precise)
  - For options with the same strike, the longer the maturity the higher the option value — calendar arbitrage otherwise
  - Interpolating on variance along the strike line makes sure that the variance will be increasing, as long as it is for the same K the next pillar has a higher variance than the previous pillar
- Mathematically,

$$v_t(K) = \frac{t_{i+1} - t}{t_{i+1} - t_i} v_{t_i}(K) + \frac{t - t_i}{t_{i+1} - t_i} v_{t_{i+1}}(K), \quad v_t(K) = \sigma_t^2(K)t \tag{12}$$

Implied Volatility Surface Implementation

- Implied volatility surface and smile builder: it takes as input the strike to implied volatility pairs for each pillar, and provide with function that returns implied volatility for a given strike and maturity:

ImpliedVol.h

```
// CubicSpline interpolated smile, extrapolate flat
class Smile
{
 public:
  Smile( const vector< pair<double, double> >& _marks );   // constructor
  double Vol(double strike);
};
class ImpliedVol
{
 public:
  ImpliedVol( const vector< pair<double, Smile> >& );
  // linear interpolation in variance, along the strike line
  double Vol(double t, double k);
};
```

Example Code For Smile and ImpliedVol

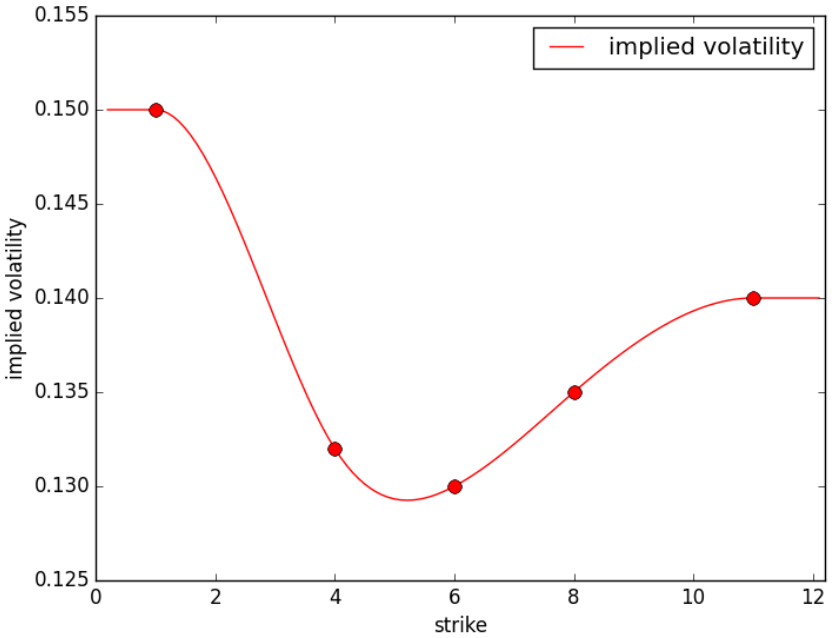
test.cpp

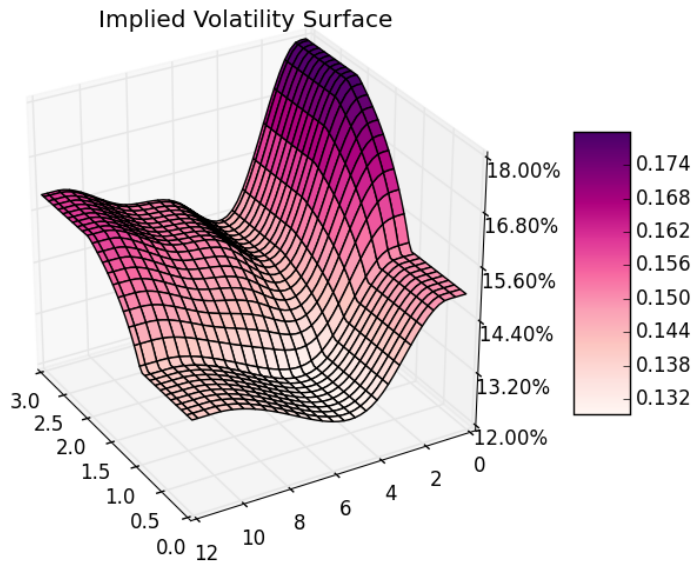
```
#include "ImpliedVol.h"
...
using namespace std;
int main()
{
  vector< pair<double, double> > marks;
  marks.push_back(pair<double, double>(1.0, 0.15));
  marks.push_back(pair<double, double>(4.0, 0.132));
  ... // set up the marks for the first pillar
  Smile sm(marks);
  ofstream fout("smile.txt"); ... // output the marks to smile.txt
  ... // construct marks for the second pillar
  Smile sm2(marks);

  vector< pair<double, Smile> > pillarSmiles;
  pillarSmiles.push_back( pair<double, Smile>(1.0, sm) );
  pillarSmiles.push_back( pair<double, Smile>(2.0, sm2) );
  ImpliedVol iv(pillarSmiles); // form a implied vol surface with two pillars

  ofstream fout3("impliedvol.txt");
  ... // output the surface to impliedvol.txt
}
```

Interpolated Implied Volatility Smile and Surface





- Interpolated smile (left) and volatility surface (right)
- Cubicspline for smile interpolation, linear on variance for time interpolation

We Are Not Done Yet

- If the market quotes us sample set of strike to option price pairs, we can now construct our implied volatility surface, with that we can in-house vanilla option with any maturity
- Markets of some asset classes do provide you with those, e.g., some commodities and equity, interest rates market (there would be other obstacles though)
- Some market does not quote you those. For example, FX market quotes
  - at-the-money volatility
  - volatility differentials — risk reversals and butterflies at 25 and 10 delta points (such that the strike dimension is more homogeneous)
- We need an extra step to convert our market marks to the inputs of our implied volatility constructor. Root finders come very handy here.

FX Market Conventions — Spot

- Spot: the exchange rate between two currencies
  - In practice the FX spot rate refers to the rate for exchanging two currencies in one or two days later, this is called the spot-lag
  - Spot-lag makes the spot rate effectively a forward rate, and makes things a little bit more complicated in implementation
  - We ignore the spot-lag and treat spot as cash exchange rate in this course

FX Market Conventions — ATMVOL

- ATMVOL: the implied volatility for at-the-money options
- Not necessarily mean $K_{ATM} = S$ (usually not the case in FX):
  - **ATM forward**, ATM strike is the forward:

$$K_{ATM} = F = Se^{\int_0^t (r_d(s) - r_f(s))ds} \tag{13}$$

    "*ATM call and put options have the same price.*" Here the ATM option refers to ATM forward.
  - **ATM delta-neutral**: $K_{ATM} = K_{DNS}$ the strike at which the straddle corresponds to a pure long vega position with no net delta:

$$\Delta(Call, K_{DNS}, T, \sigma_{ATM}) + \Delta(Put, K_{DNS}, T, \sigma_{ATM}) = 0 \tag{14}$$

$$N\left(\frac{ln\frac{F}{K_{DNS}} - \frac{1}{2}\sigma_{ATM}^2 T}{\sigma_{ATM}\sqrt{T}}\right) - N\left(\frac{-ln\frac{F}{K_{DNS}} + \frac{1}{2}\sigma_{ATM}^2 T}{\sigma_{ATM}\sqrt{T}}\right) = 0,$$

    so

$$K_{DNS} = Fe^{-\frac{1}{2}\sigma_{ATM}^2 T}. \tag{15}$$

FX Market Conventions — Delta

- Delta is the sensitivity of option price to the underlying
- Recall that Black-Scholes call formula for an FX option

$$Call_{BS} = Se^{-r_f T} N(d_+) - Ke^{-r_d T} N(d_-) \tag{16}$$

$$d_\pm = \frac{\ln(F/K) \pm \frac{1}{2}\sigma^2 T}{\sigma\sqrt{T}} \tag{17}$$

where r_f is the risk free rate of the LHS currency (foreign ccy), r_d is the risk-free interest rate of the RHS currency (domestic ccy)

- Call delta:

$$\Delta_{call}(\sigma, K, T, r_d, r_f) = \frac{\partial Call_{BS}}{\partial S} = e^{-r_f T} N(d_+) \tag{18}$$

FX Market Conventions — Delta

- Similarly the Black-Scholes put formula

$$Put_{BS} = Ke^{-r_d T} N(-d_-) - Se^{-r_f T} N(-d_+) \tag{19}$$

- And put delta

$$\Delta_{put}(\sigma, K, T, r_d, r_f) = \frac{\partial Put_{BS}}{\partial S} = -e^{-r_f T} N(-d_+) \tag{20}$$

FX Market Conventions — Spot Delta and Forward Delta

- Similar to ATM convention, delta has its conventions
- What we have seen are the first derivative to the spot — **spot delta**
- It can be also the first derivative with respect to the forward — **forward delta**
- Forward delta is the first derivative of the **undiscounted** price with respect to the forward

$$\Delta_{call} = \frac{\partial[FN(d_+) - KN(d_-)]}{\partial F} = N(d_+) \tag{21}$$

$$\Delta_{put} = \frac{\partial[KN(-d_-) - FN(-d_+)]}{\partial F} = -N(-d_+) \tag{22}$$

FX Market Convention — Our Use Case

We will use ATM forward convention and forward delta for illustration

- The forward is more the center of the strike, rather than spot, especially for long dated options

$$K_{ATM}(t) = F(t) = Se^{(r_d - r_f)t} \tag{23}$$

- Forward delta ranges from 0 to 1 — the cumulative normal function

$$\Delta_{call} = N(d_+), \quad \Delta_{put} = -N(-d_+) \tag{24}$$

- Simple in form, without the tedious discount factor
- Keep in mind that an FX market in practice handles all the hassles regarding the conventions
- Also note that we have omitted one convention — the premium convention. All our premium are quoted in the the RHS (domestic) currency. If the option premium is quoted in the LHS currency, the delta will change since the unit of delta is the LHS currency

Why Deltas?

- It is the first component of your risk to be hedged
- It is used as a homogeneous measure of moneyness across option maturities. A 1Y call and a 10Y call, both struck at 100 does not have the same delta risk — forward curve is not flat
- The market players often quote like `1Y 25d call, 20%`, meaning the call option with the strike K such that $\Delta_{call}(\sigma = 20\%) = 0.25$. The exact strike K needs to be found through a root finder.
- Most liquid FX option instruments have strikes at call/put, $25d/10d$

FX Market Instruments

- Ideally if the market quotes delta to implied volatility pairs (d_i, σ_i) we can just convert them to strike and implied volatility pairs (K_i, σ_i) and we will have the inputs for our implied volatility surface interpolater.

- Unfortunately except ATMVOL, most likely you will get quotes for volatility spreads, in the form of **market strangle** or **smile strangle**, and **risk reversal**

Market Strangle

Market strangle is a volatility spread $\sigma_{\Delta_{ms}}$ used to represent a strangle payoff: a call + a put option.

- The strike of the call option is calculated as the strike K_c such that

$$\Delta_{call}(K_c, \sigma_{ATM} + \sigma_{\Delta_{ms}}) = \Delta \tag{25}$$

- The strike of the put option is calculated as the strike K_p such that

$$\Delta_{put}(K_p, \sigma_{ATM} + \sigma_{\Delta_{ms}}) = -\Delta \tag{26}$$

- The price of the strangle instrument is the sum of the two options using Black-Scholes formula using K_c, K_p, and volatility $\sigma_{ATM} + \sigma_{\Delta_{ms}}$

- Given σ_{ATM} and $\sigma_{\Delta_{ms}}$, the strangle instrument and its price is determined. It forms a constraint of the smile we build.
- Market strangle describes the convexity of our volatility smile
- Typical strangle instruments: 10 or 25 delta, i.e., $\Delta = 0.25$ or 0.10

Smile Strangle and Risk Reversal

Smile Strangle

Smile strangle BF_Δ is the pure volatility spread:

$$BF_\Delta = \frac{1}{2}(\sigma_{\Delta_c} + \sigma_{\Delta_p}) - \sigma_{ATM} \tag{27}$$

For example 25 delta smile strangle is the average of 25 call delta's implied volatility and 25 put delta's implied volatility minus σ_{ATM}. It is commonly called the **butterfly**. It measures the convexity of the smile.

Risk Reversal

Risk reversal RR_Δ is the pure volatility spread:

$$RR_\Delta = \sigma_{\Delta_c} - \sigma_{\Delta_p} \tag{28}$$

It measures the skewness of the smile.

The FX Volatility Marking Problems

- Given σ_{ATM}, if we know the smile strangle and risk reversals for 25 and 10 delta points, we can easily calculate the implied volatilities for the four delta points:

$$\begin{cases} BF_{\Delta} &= \frac{1}{2}(\sigma_{\Delta_c} + \sigma_{\Delta_p}) - \sigma_{ATM} \\ RR_{\Delta} &= \sigma_{\Delta_c} - \sigma_{\Delta_p} \end{cases} \tag{29}$$

  — two equations and two unknowns for each delta point
- Converting (Δ_i, σ_i) pairs to (K_i, σ_i) pairs, and together with the ATM pairs, we can form the inputs to our implied volatility surface
- Risk reversal is available in the raw market
- Smile strangle is not available. Instead we have only market strange in the raw market.
- The practical problem is to build implied volatility surface given σ_{ATM}, market strangles, and risk reversals — exercise for those interested

How About Diffusion Models?

- We are able to price plain vanilla option and match the quotes in the market
- What else can we price? Maybe some digital option with static replication — note that the Black-Scholes digital option formula does not work now, why?
- We are model-less now — our only model so far Black-Scholes does not work. We do not have a diffusion model that can reproduce all the vanilla prices, so not possible to price any path dependent exotic products
- This is in fact the basic requirement of any advanced models: they need to be able to calibrate to the implied volatility surface — reproducing the vanilla prices
- The **local volatility model** on the way — the straight forward yet effective extension to Black-Scholes model

References


Nicolai M. Josuttis.
The C++ Standard Library: A Tutorial and Reference.
Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.


William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery.
Numerical Recipes in C++, chapter 9, Root Finding and Nonlinear Sets of Equations.
2002.