Introduction

QF607 Numerical Methods

Zhenke Guan
zhenkeguan@smu.edu.sg

About Me

- Career
  - Quant Analyst ,
    Garda Capital Partners, 2023 Jan - present
  - Quant Analyst, Rates Option and Structured Notes Trading,
    Barclays Investment Bank, 2019 – 2023
  - Quant Analyst, IR/FX/XVA,
    Lehman Brothers, Mizuho, ANZ, Standard Chartered Bank, 2006- 2019
- Education
  - Mphil, PhD Mathematical Finance,
    Manchester University/MBS , UK

Course Objectives

- Introduction of numerical methods used in quantitative finance
- Understand the convention and standard models used in financial market: a little bit more than Black and Scholes
- Be able to use and implement numerical methods to solve problems
- Be able to implement pricers for different financial products: a little bit more than Call and Put
- Be able to prototype analytics tools to address problems in derivative market

Focus: **WHAT** are used in practice, **WHY** and **HOW**?

Main Topics

- Number representation and numerical errors
- Binomial and trinomial tree models
- Root search, interpolation, volatility smiles
- Monte-Carlo simulation (MC)
- Partial differential equations (PDE)
- Numerical Optimization

We will discuss concepts and techniques from practitioners' point of view.

Assessment

- Assignments 20%: 2 assignments, 10% each
- Project 30%: group project (1 to 3 persons)
- Final exam (open book, 3 hour) 50%
- Assignments are individual. Project can be done in group of one to three members. Number of members will not be taken into consideration at project assessment.

Required Knowledge

There is no strict requirements or assumptions on your knowledge base, but in general it would be helpful if you are familiar with

- linear algebra,
- probability theory,
- calculus,
- stochastic calculus

Most concepts that are used will be covered, briefly introduced, or referred to recommended reading materials.

We will be using **python** as the programming language for implementation.

Course Materials

- No textbook
- Slides and assignment / project handouts
- Recommended further readings from different books, papers and websites for different topics

Numerical Analysis in Financial Industry

- Financial industry is the industry that deals with numbers
  - asset prices
  - market movements (asset return and volatility)
  - interest rates
  - ...
- Mathematical models are used to
  - describe the relationship between the numbers
  - abstract / extract key information from the numbers

Numerical Analysis in Financial Industry

- There are few cases that application of standard formulas can solve the problem
- There are increasing demand on practitioners to apply the methodologies and adapt them appropriately to financial analyses, pricing, risk modeling, and risk management
- Numerical analysis is concerned with all aspects of the numerical solution of a problem, from the theoretical development and understanding of numerical methods to their practical implementation as reliable and efficient computer programs

How does it work?

- If a problem cannot be solved analytically, replace it with a “nearby problem” which can be solved more easily
- It uses the language and results of linear algebra, real analysis, and functional analysis.
- There is a fundamental concern with error, its size, and its analytic form.
- Numerical analysts are interested in measuring the efficiency of algorithms.
- Numerical methods are implemented with finite precision computer arithmetic.

Two primary concerns while using numerical methods:

- computational cost
- accuracy of the results

Computational Cost

Same mathematical formula can have different implementations
Example − $d_{\pm}$ in Black and Scholes formula:

$$d_{\pm} = \frac{\log \frac{Se^{\mu t}}{K} \pm \frac{1}{2}\sigma^2 t}{\sigma\sqrt{t}} \tag{1}$$

How to implement it so that the cost of calculation is minimum?

Cost of Operations

The cost of operation depends on

- Type of operands, CPU and computer architecture, programming language
- Example code measuring the cost of operations:

```python
import timeit

def opTiming(op, opName, repeat):
    elapsed_time = timeit.timeit(op, setup='import math', number=repeat)
    print(opName, "\t", elapsed_time / repeat)

repeat = int(1e8)
opTiming("x = 5.0 + 7.0", "add", repeat)
opTiming("x = 5.0 * 7.0", "mul", repeat)
opTiming("x = 5.0 / 7.0", "div", repeat)
opTiming("x = math.log(7.0)", "log", repeat)
opTiming("x = math.exp(7.0)", "exp", repeat)
opTiming("x = math.sqrt(7.0)", "sqrt", repeat)
```

Which implementation is faster to calculate d_+? And why?

- method 1

$$d1 = (log(S_0 e^{\mu t}/K) + vol * vol * t/2)/vol/sqrt(t) \tag{2}$$

- method 2

$$stdev = vol * sqrt(t) \tag{3}$$
$$d1 = (log(S/K) + \mu t)/stdev + stdev/2 \tag{4}$$

```
m1 = """
S = 100;K = 105;vol = 0.1;t=2;mu=0.01
d1 = (math.log(S * math.exp(mu*t) / K) + vol * vol * t / 2) / vol / math.sqrt(t)
"""
m2 = """
S = 100;K = 105;vol = 0.1;t=2;mu=0.01
stdev = vol * math.sqrt(t)
d1 = (math.log(S / K) + mu*t) / stdev + stdev / 2
"""
repeat = int(1e7)
opTiming(m1, 'm1', repeat)
opTiming(m2, 'm2', repeat)
```

Number Representation

- Translator between our language to computer's language
- Computer uses binary number system - everything is represented as combination of 0 and 1
- The binary form of a positive integer is given by $a_n a_{n-1} \dots a_1 a_0$, which is equivalent to

$$a_n 2^n + a_{n-1} 2^{n-1} + \dots + a_1 2^1 + a_0 2^0 \tag{5}$$

- The binary form of a positive number less than 1 is given by $a_{-1} a_{-2} \dots a_{-m-1} a_{-m}$, equivalent to

$$a_{-1} 2^{-1} + a_{-2} 2^{-2} + a_{-3} 2^{-3} \dots \tag{6}$$

Integer Representation

For example, a binary number

$$11011_2 \tag{7}$$

represents the value:

$$\begin{aligned} & 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 && (8) \\ & = 16 + 8 + 0 + 2 + 1 && (9) \\ & = 27_{10} && (10) \end{aligned}$$

If we use 32 binary bits to represent a "'integer"' number, what is the range of numbers we can represent?

- unsigned: 0 to $2^{32} - 1$
- signed: -2^{31} to $2^{31} - 1$

Fixed Point Representation

To represent a real number in computers, we can define a fixed point number type simply by implicitly fixing the binary point to be at the same position of a numeral.

For example, to represent 13.5 with binary point position at the left of the last bit: 1101.1_2

$$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} \tag{11}$$
$$= 8 + 4 + 0 + 1 + 0.5 \tag{12}$$
$$= 13.5_{10} \tag{13}$$

To define a fixed point type, we need two parameters:

- width of the number representation
- binary point position

- For ease of discussion, we use `fixed<w, b>` to denote a fixed point type with `w` width and binary point position at `b` counting from 0 (the least significant bit).
- `int` type is a special case of fixed point representation: `fixed<32, 0>`
- For example, `fixed<8, 3>` denotes a 8 bit fixed point number, of which 3 right most bits are fractional. Therefore, the number `00010110` represents:

$$00010.110_2 = 1 \times 2^1 + 1 \times 2^{-1} + 1 \times 2^{-2} = 2.75 \tag{14}$$

- The same bit patter `00010110` represents a different number if it is of a different type, e.g., `fixed<8, 5>`:

$$000.10110_2 = 1 \times 2^{-1} + 1 \times 2^{-3} + 1 \times 2^{-4} = 0.6875 \tag{15}$$

Negative Number Representation

Two common way of representing signed types:

- Sign and magnitude representation (SMR) - use a sign bit (first bit): $0 =$ positive and $1 =$ negative, the remaining bits in the number indicate the magnitude (absolute value)

- Example with `fixed<8, 0>`: $\mathbf{0}\,0001100 = 12_{10}$, $\mathbf{1}\,0001100 = -12_{10}$

- Two's-complement: the value x of a `fixed<w, b>` number $a_{w-1}a_{w-2}\dots a_0$ is given by:

$$x = -a_{w-1}2^{w-1-b} + \sum_{i=0}^{w-2} a_i 2^{i-b} \tag{16}$$

  - Example with `fixed<8, 0>`: $00001100 = 12_{10}$, $10001100 = -2^7 + 12 = -116$

Two's complement representation

The advantage of using two's complement representation is

- fundamental arithmetic operations of either positive or negative numbers are identical - including addition, subtraction, multiplication, and even shifting.

For example, if we look at two signed `fixed<4, 1>` numbers: `101.1` and `110.1`

$$
\begin{array}{rrlr}
 & 101.1 & -1\times 2^2 + 1\times 2^0 + 1\times 2^{-1} = -2.5_{10} & (17)\\
 & +\ 110.1 & -1\times 2^2 + 1\times 2^1 + 1\times 2^{-1} = -1.5_{10} & (18)\\
\hline
\underbrace{1}_{\text{overflow bit}} & 100.0 & \qquad\qquad\qquad\qquad\qquad \leftarrow -4_{10} & (19)
\end{array}
$$

Note that the left most overflow bit is discarded.

Converting a real number to signed `fixed<w, b>`

- input: a real number x, width w, binary position b
- output: array a of size w, a[i] is the i-th bit of the fixed point representation

$$x = -a_{w-1}2^{w-1-b} + \sum_{i=0}^{w-2} a_i 2^{i-b} \tag{20}$$

```python
def toFixedPoint(x : float, w : int, b : int) -> [int]:
    # set a[w-1] to 1 if x < 0, otherwise set a[w-1] to 0
    a = [0 for i in range(w)]
    if x < 0:
        a[0] = 1
        x += 2**(w-1-b)
    for i in range(1, w):
        y = x / (2**(w-1-i-b))
        a[i] = int(y)  # round y down to integer
        x -= a[i] * (2**(w-1-i-b))
    return a

print(toFixedPoint(-10, 8, 1))
print(toFixedPoint(-9.5, 8, 1))
print(toFixedPoint(9.25, 8, 2))
```

Range Matters

- It is not difficult to notice that the range represented by fixed point representation is limited.
- If we try to represent a number beyond the range of the type we do not get sensible result:

```
print(toFixedPoint(20, 8, 3))
print(toFixedPoint(20, 9, 3))
```

We need to deal with those cases:

```python
def toFixedPoint2(x : float, w : int, b : int) -> [int]:
    # set a[w-1] to 1 if x < 0, otherwise set a[w-1] to 0
    a = [0 for i in range(w)]
    if x < 0:
        a[0] = 1
        x += 2**(w-1-b)
    for i in range(1, w):
        y = x / (2**(w-1-i-b))
        if int(y) > 1:
            raise OverflowError('fixed<' + str(w) + "," + str(b) + "> is not sufficient to represent " + str(x))
        a[i] = int(y) # % 2  # round y down to integer
        x -= a[i] * (2**(w-1-i-b))
    return a

print(toFixedPoint2(20, 8, 3))
```

Exercise in assignment 1: implement a function converting fixed point to a real number.

- Problems caused by overflow can be serious.
- On 22 Apr 2018, the price of an ERC20 token BEC plunged by $\tilde{6}0\%$ due to an overflow bug in smart contract

- Code in smart contract:

```
function batchTransfer(address[] _receivers, uint256 _value) {
    uint cnt = _receivers.length;
    uint256 amount = uint256(cnt) * _value;
    require(cnt > 0 && cnt <= 20);
    require(_value > 0 && balances[msg.sender] >= amount);
    ... # transfer the token
 }
```

- Many other ERC20 tokens' smart contracts were found having the same bug

Microsoft Exchange Server Y2K22 Bug

- On 1 Jan 2022, some Microsoft Exchange admins had to come to work facing their first challenge of the new year: installing a patch to fix jammed messages that started at midnight on January 1st.
- The bug was due to "dates that are stored in the format yymmddHHMM converted to a signed 32-bit integer overflowed on 1 January 2022, as $2^{31} - 1 = 2147483647$" — 2201010001 cannot be represented as uint32.
- See Microsoft Exchange team blog on the details of the bug.

Floating Point Representation of Real Numbers

- Real number 1234.56 can be represented in scientific notation

$$1.23456 \times 10^3 \tag{21}$$

- Floating point representation uses scientific notation in binary format

$$\underbrace{1.01}_{\text{significand}} \times 2^{\overbrace{-1}^{\text{exponent}}} \tag{22}$$

- The IEEE 754 standard - the most widely used floating point standard:

$$\underbrace{0}_{\text{1 sign bit } s} \quad \overbrace{001\ldots10}^{x \text{ exponent bits } e_1e_2\ldots e_x} \quad \underbrace{1001\ldots001}_{y \text{ significand bits } m_1m_2\ldots m_y} \tag{23}$$

- The number it represents is

$$(-1)^s \times (1 + \text{significand}) \times 2^{\text{exponent - bias}} \tag{24}$$

$$\text{significand} = \sum_{j=1}^{y} m_j \times 2^{-j}, \quad \text{exponent} = \sum_{i=1}^{x} e_i \times 2^{x-i} \tag{25}$$

```
import numpy as np
for f in (np.float32, np.float64, float):
    finfo = np.finfo(f)
    print(finfo.dtype, "\t exponent bits = ", finfo.nexp, "\t significand bits = ", finfo.nmant)
```

- Floating point supports a much wider range of values
  - 32 bit "'float"': 10^{-38} to 10^{38}.
  - 64 bit "'float"': 10^{-308} to 10^{308}.
- More on floating point arithmetic: What Every Computer Scientist Should Know About Floating-Point Arithmetic

Numerical Errors

- A numerical error is the difference between the calculated approximation of a number and its exact mathematical value.
  - round off error (or rounding error)
    computers have size and precision limits on their ability to represent numbers
  - truncation error
    arises when we approximate a continuous model with a discrete one

Round off errors

- Finite-precision causes round off in individual calculations
- Effects of round off usually accumulate slowly

Round Off Errors

```python
x = 10776321
nsteps = 1235
s = x / nsteps
y = 0
for i in range(nsteps):
    y += s
print(x - y)
```

- Subtracting nearly equal numbers leads to severe loss of precision.
- A similar loss of precision occurs when two numbers of very different magnitude are added.
- Round off is inevitable: good algorithms minimize the effect of round off.

Machine Epsilon

- Machine epsilon is a number $\epsilon > 0$, such that the computer considers $1 + \epsilon = 1$

```python
x = 10.56
print(x == x + 5e-16)
```

- It measures the effects of round off errors made when adding, subtracting, multiplying, or dividing two numbers.
- Different machine may have different machine epsilon.
- Rule of thumb (but not precisely): ϵ is at the order of 10^{-16}

```python
x = 0.12345678912345678 90
y = 0.1234567891
scale = 1e16
z1 = (x-y) * scale
print("z1 = ", z1)

z2 = (x*scale - y*scale)
print("z2 = ", z2)
```

Truncation Error

- Truncation errors are from numerical methods: introduced by neglecting higher order terms.

- Example: Taylor expansion:

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2!}f^2(x) + \dots \tag{26}$$

Taylor Theorem

Assume that f is continuous and $f'(x), f^2(x), \ldots, f^{(n)}(x)$ exists over (a, b), let $x_0 \in [a, b]$, then for every $x \in (a, b)$, there exists a number c that lies between x_0 and x such that

$$f(x) = f(x_0) + \sum_{k=1}^{n} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k + \frac{f^{(n+1)}(x_0)}{(n+1)!}(x - c)^{n+1} \qquad (27)$$

Local truncation error

Local truncation error is the error that our increment function causes during a single iteration.
In the case of taylor expansion,

$$f(x_0 + h) = f(x_0) + hf'(x_0) + O(h^2) \tag{28}$$

If we approximate $f(x_0 + h)$ by $f(x_0) + hf'(x_0)$, the local truncation error is $O(h^2)$.

We say that the numerical method has order p if for any sufficiently smooth solution of the initial value problem, the local truncation error is $O(h^{p+1})$.

Global Truncation Error

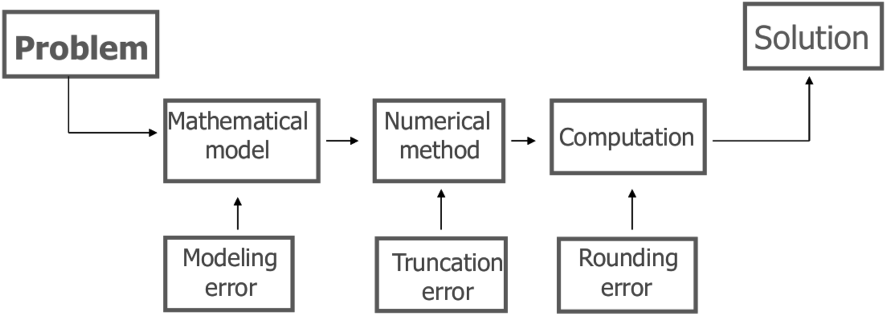
Global truncation error is the accumulation of the local truncation error over all of the iterations.

- If we know $f'(x)$ and $f(x_0)$, to approximate $f(x_0 + h)$ for a not so small h, we can divide h into n steps $\Delta h = \frac{h}{n}$, then

$$
\begin{aligned}
&y_0 = f(x_0) \\
&x_1 = x_0 + \Delta h, \quad y_1 = y_0 + f'(x_0)\Delta h \\
&\ldots \\
&x_i = x_{i-1} + \Delta h, \quad y_i = y_{i-1} + f'(x_{i-1})\Delta h \\
&\ldots \\
&x_n = x_{n-1} + \Delta h, \quad f(x_0 + h) \approx y_n = y_{n-1} + f'(x_{n-1})\Delta h
\end{aligned}
$$

  The global truncation error $e_n = f(x_0 + h) - y_n$.
- A numerical method is convergent if global truncation error goes to zero as the step size goes to zero.

Errors of Numerical Solution



Measuring Errors

- Errors can be measured with either absolute error or relative error, or both.
  - Abseolute error: $|x_c - x|$
  - Relative error: $\frac{|x_c - x|}{x}$ where x_c is the computed value and x is the exact value

- In some derivative pricing cases we take the notional of the trade to calculate relative error.

Convergence

- Convergent sequence: suppose that $\{x_n\}^\infty$ is an infinite sequence, the sequence is said to have the limit L

$$\lim_{n \to \infty} x_n = L \tag{29}$$

if for any given $\epsilon > 0$, there exists an N such that for any $n > N$, we have $|x_n - L| < \epsilon$.

Order and Rate of Convergence

- Let x_n be a sequence that converges to a number L, if there exists a positive constant λ, such that

$$\lim_{n\to\infty} \frac{|e_{n+1}|}{|e_n|^p} = \lim_{n\to\infty} \frac{|x_{n+1} - L|}{|x_n - L|^p} = \lambda \tag{30}$$

- p is the order of convergence

Order and Rate of Convergence

- More intuitively $|e_{n+1}| = \lambda |e_n|^p$ when $n \to \infty$.
- The larger p and the smaller λ, the more quickly the sequence converges
- Specifically:
  - if $p = 1$ and $0 \leq \lambda \leq 1$, $|e_{n+1}| = \lambda |e_n| < |e_n|$, then
    - if $\lambda = 1$, the convergence is **sublinear** - convergence is slower than linear
    - if $0 < \lambda < 1$, the convergence is **linear** with rate of convergence λ
    - if $\lambda = 0$, the convergence is **superlinear** - faster than linear
  - if $p = 2$, $|e_{n+1}| = \lambda |e_n|^2$, $\lambda > 0$, the convergence is quadratic
  - if $p = 3$, $|e_{n+1}| = \lambda |e_n|^3$, $\lambda > 0$, the convergence is cubic

Tolerance

- A variety of criteria can be used for deciding when your approximation solution is close enough to true solution.
- Possible tests
  - the absolute change is sufficiently small $|x_{k+1} - x_k| \leq \text{tol}$
  - the relative change is sufficiently small $|\frac{x_{k+1} - x_k}{x_{k+1}}| \leq \text{tol}$
- Which test to use depends on the application itself