FN6815_Lect_01_Intro

February 27, 2024

FN6815 Numerical Methods for Financial Instrument Pricing

Lecture 1: Foundations of Numerical Computations

- Dr. Yang Ye
- Email: yy@runchee.com

- 2023/2024 Mini Term 5

1. Introduction

1.1 Objective

As you advance into the 5th mini term of the MFE program, you've accumulated a wealth of knowledge in finance, mathematics, and programming. Our objective is to consistently apply and broaden this knowledge in real-world scenarios.

The transition from theoretical understanding to practical application is seamlessly achieved through proficient implementations. Within the financial sector, the pricing of financial instruments is a critical component of these systems, requiring both precision and efficiency.

Computation is at the core of finance For financial institutions, pricing is at the core to calculate risk, profit and loss, settlement.

The ability to compute fast and accurately is the competitive edge between different banks / institutions. Those can quickly calculate the price of a financial instrument and manage the risk will have a competitive edge over slower competitors.

The Quants or "Strats" (abbr. for "quantitative Strategist") combines the quantitative, programming and system building experience are to drive the development and delivery of efficient pricing and risk management systems.

For instance, a Quant might use advanced mathematical models and high-performance computing to develop a system that can quickly price a wide range of financial instruments, thereby helping the institution respond rapidly to market changes.

Numerical methods Numerical methods are computational strategies employed to solve mathematical problems that are challenging or impossible to solve analytically (using straightforward mathematical operations). They provide an approximate solution but with a precision that can be controlled by the user.

In the context of financial instrument pricing models, which frequently involve stochastic differential equations (SDEs) or partial differential equations (PDEs), numerical methods serve as the principal tools for solving these complex equations.

For example, the Black-Scholes model for pricing an option involves a partial differential equation. While an analytical solution is available for certain types of options, numerical methods are often used when the model is modified to more accurately reflect the complexities of financial markets, such as early exercise features or stochastic volatility.

Objectives Our goal is to merge two fundamental areas of knowledge:

- Mathematical modeling of derivative pricing, culminating in the implementation of the model. This process involves the selection of appropriate methods, understanding the complexity of the model, ensuring its accuracy, and managing potential errors.
- Programming skills, which range from structured programming, functional programming and Object-Oriented Programming (OOP) to data analysis. These skills are essential in creating efficient and maintainable code.

The expected outcomes of this integration are:

- The ability to solve complex problems with high accuracy and performance. This could mean accurately pricing a complex derivative within a short time frame, which can be crucial in fast-paced financial markets.
- The development of a system that is scalable and adaptable, capable of addressing a variety of similar problems. For instance, a well-designed derivative pricing system could be easily extended to handle new types of derivatives or updated to use new pricing models.

This provides an exceptional opportunity for you to hone your coding abilities and enhance your system design capabilities, thereby preparing you for the challenges of the dynamic world of finance.

1.1 Tools

We will be utilizing Python for this course due to several reasons:

- Pros
    1. Python boasts a clean, readable syntax, which makes it an excellent choice for beginners and experienced programmers alike.
    2. It is highly flexible, supporting various programming paradigms including structured programming, Object-Oriented Programming (OOP) and functional programming.
    3. While Python itself may not be the fastest language, it compensates with a plethora of fast and well-designed libraries such as `NumPy`, `SciPy`, `Matplotlib`, etc. These libraries provide powerful tools for numerical computation, scientific computing, data visualization respectively.
    4. Python offers a user-friendly development environment. Tools like Jupyter Notebook and Visual Studio Code make coding, testing, and debugging Python code a breeze.
- Cons

1. Python may not be the most performant language for certain applications, but it is excellent for prototyping and testing ideas quickly.
2. Python is dynamically typed, which means type errors can only be caught at runtime. However, this also provides flexibility in writing code.
3. To use Python effectively, especially its libraries, it's beneficial to understand the underlying implementation from the libraries that are open sourced. This isn't necessarily a disadvantage, as it encourages a deeper understanding of the tools you're using.

1.2 Teaching Approach

1. We will be using Jupyter notebooks, which combine documentation and runnable code in an interactive environment.
2. I will use Jupyter notebooks to present the material, and you will use them to follow along and experiment.
3. You will also be expected to complete your assignments in Jupyter notebooks.

Please feel free to run, copy and paste the code snippets to experiment with them and understand how they work!

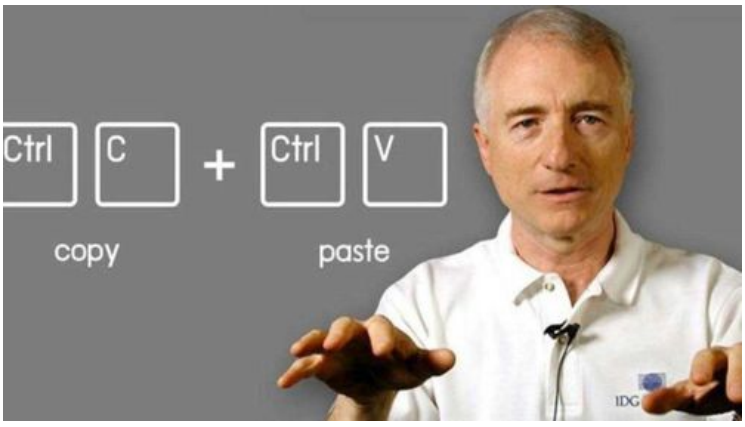


Image: Larry Lester, father of Copy&Paste. RIP Feb 16th, 2020.

1.3 Assessment

Components The course assessment will consist of the following components:

- 6 weekly assignments: These assignments will be based on the topics covered during the week. The list of questions for each assignment will be provided after the respective lecture.
- One mid-term exercise: This will be a short, 20-30 minute in-class exercise conducted during either on Week 4 or 5. The format of this exercise will be similar to that of the final exam, providing you with a useful preview of what to expect.
- Final exam: The final exam will be an open-book format, with online submission of answers. The test format and requirement will be similar to my FN6806 OOP II.

Specifically, the use of generative AI tools or search engines like Google to directly search for answers is not permitted. This is to ensure that the exam assesses your understanding and application of the course material, rather than your ability to find answers online.

Submission

Here are the guidelines for organizing and submitting your assignments:

- Organization:
  - Create a directory and name it as "FN6815-Your Name".
  - Organize your assignments into separate directories within this main directory, named as A1, A2, ..., A6.
  - Write your solution code and explanation in a Jupyter notebook for each assignment.
- Submission by sharing:
  - Share the main directory with *yy@runchee.com* on Google Drive.
  - You only need to set up the sharing once. Subsequent assignments can be added to the respective directories in the shared folder.
  - The deadline for each assignment is 6pm every Wed of the following week.

1.4 Topics

he course is divided into two main parts.

- Part 1:
  - Foundations of Numerical Computations
  - Python programming
  - NumPy programming
  - Solving Linear and Nonlinear Equations
  - Numerical Differentiation and Integration
- Part 2:
  - Solving ODE
  - Pricing with Numerical integration
  - Pricing with Monte Carlo simulation
  - Pricing with Finite Difference method

2. Foundations

What does numerical algorithm do?

- A numerical algorithm is designed to solve mathematical problems and provide numerical results.
- The primary goal of these algorithms is to deliver solutions both accurately and quickly.

Can everything be computed? The "computing revolution" over the past 30-40 years has shown that the scope of what can be computed continues to expand with the advent of more powerful computers and smarter algorithms.

- The evolution of computers: from human calculators, to mechanical and electronic devices, and now to advanced semiconductor-based systems. You can learn more about this evolution from the following resources:
  - Human Computers
  - Difference Engine
  - ENIAC
- The development of smarter theorems and algorithms.

The frontier of what can be computed has been pushed forward in several areas:

- **Precision problems**, such as calculating the values of Pi and e. The required precision depends on the application. For instance, finance might be satisfied with a precision of 1e-08, but space exploration would require much higher precision.
- **Result-sensitive problems, such as predicting stock prices or weather. These predictions are only valid for a short period and need to be constantly updated. However, the data used for these predictions is often incomplete.
- **Efficiency problems**, such as training large models like AlphaGo or GPT. These models have billions of parameters and require significant computational resources.
- **Compute-hard problems**, such as cracking passwords. These problems can theoretically be solved by a computer but would take an impractically long time. This property is actually useful in cryptography, where we rely on the difficulty of these problems to secure our data.

Numerical problems in Pricing

- Financial pricing presents a combination of precision, result-sensitive, and efficiency problems. The task involves translating mathematical models with a small number of parameters into practical computations.
- The challenging part is often dealing with stochastic inputs. More sophisticated models tend to have more stochastic inputs, which means they are more computationally intensive due to the increased number of possibilities.
- Additionally, these models require more effort to update or "calibrate" their parameters.

2.1 What do numerical methods require?

When we employ numerical methods with computers, there are certain assumptions and approximations at play, even if we may not be explicitly aware of them.

1. Assumptions:

- The general ones:
  - Equations are **solvable** within the supported number range. In the context of quantitative finance, this typically means having roots within the real number range.
  - There is **convergence** in our results. In other words, we expect the results to converge to an acceptable range with a narrow standard deviation.
- There are also problem-specific assumptions that need to be considered.

2. Approximations:

- The general ones:
  - **Rounding errors** occur due to the inherent limitation of computers in representing real numbers exactly. This is a result of the finite precision of the computer's numerical representation.
  - **Truncation errors** arise from the simplification of the mathematical model. Examples include truncating an infinite series, replacing a derivative with a finite difference, or discretizing a continuous variable.
- There are also problem-specific approximations that need to be considered.

Question: What are examples for approximation?

1. `sqrt(2)`
2. Harmonic series `1 + 1/2 + 1/3 + ... + 1/n`

2.2 Approximation: Understanding Computational Limits

Our code executes on digital semiconductor chips. Despite their widespread use and capabilities, these chips have inherent limitations.

A key limitation is the representation of floating-point numbers. Due to memory constraints, only a finite number of digits can be stored after the decimal point.

- `e` is the largest number that makes `x + e = x` for all `x` of floating point numbers.
- `mach_e` is the gap between the number 1 and the next largest floating point number. Any number lower than it will be round-off.

[1]:
```
import numpy as np

# for mach_e
(np.finfo(np.float64).eps, np.finfo(float).eps)
```

[1]:
```
(2.220446049250313e-16, 2.220446049250313e-16)
```

[2]:
```
e = 1
while 1 + e > 1:
    e = e / 2

# e is the largest number that make 1+e ~= 1
```

```
print("e:", e)

# mach_e is the smallest number that make 1+mach_e > 1.
# mach_e = e*2 because all number is binary number in compute.as_integer_ratio
mach_e = 2 * e
print("mach_e:", mach_e)

# Behavior with e
# Add single e won't change the value
print(("((100 + e) + e) + e == 100", ((100 + e) + e) + e == 100))
# Add double e will change the value
print(("(1 + (e + e)) > 1", (1 + (e + e)) > 1))
```

```
e: 1.1102230246251565e-16
mach_e: 2.220446049250313e-16
('((100 + e) + e) + e == 100', True)
('(1 + (e + e)) > 1', True)
```

What does this tiny number (e) affect us? The sum of the series $\frac{1}{n}$ from $n = 1$ to ∞ is divergent, meaning it tends towards infinity.

$$\sum_{n=1}^{\infty} \frac{1}{n} = \infty$$

This can be approximated by the integral of $\frac{1}{x}$ from 1 to n, which equals $\log n$.

$$\sum_{n=1}^{n} \frac{1}{n} \approx \int_{1}^{n} \frac{dx}{x} = \log n$$

We can code this formula, but the computation is slow and never reaches even 100. This serves as a counter-intuitive example to the notion that "more time yields better results".

In below code, `harmonic_series(n)` computes the sum of the series, while `log_approximation(n)` computes the logarithmic approximation. Despite increasing `n`, the harmonic series never reaches even 100.

[3]:
```
import math


def harmonic_series(n):
    # use list comprehension - we will cover later.
    # return sum(1.0/i for i in range(1, n+1))

    # use for loop
    ss = 0
    for i in range(1, n):
        ss += 1 / i
```

```
    return ss


def log_approximation(n):
    return math.log(n)


print(harmonic_series(int(1e7)), log_approximation(int(1e7)))
```

```
16.69531126585727 16.11809565095832
```

We can calculate the maximum value of the harmonic series that we can reach by computation using the machine epsilon.

[4]:
```
import numpy as np

mach_e = np.finfo(np.float64).eps
# When we reach the 1 / mach_e, 1 / (1 / mach_e) ~ mach_e has no impact to the␣
 ↪sum.
# this number is
limit = 1 / mach_e
print(f"limit: {limit}")

# the limit to the limit is
sum_limit = np.log(int(1 / mach_e))
print(f"sum: {sum_limit}")
```

```
limit: 4503599627370496.0
sum: 36.04365338911715
```

In summary, we shall be aware of the inherent limitations of the computer's numerical representation.

- We shall be aware of the any calculation that may involve very large or very small numbers, as they cause **rounding errors**.
- We shall be also aware of **truncation error** that we use finite for the infinite, or discretize the continuous variable.

2.3 Stability and Convergence

Effective numerical algorithms should meet the following criteria:

- Stability: They should handle diverse inputs consistently.
- Convergence: The solution shall reach a better result with more efforts (e.g. more iterations). And we should reach a solution with required precision in a reasonable number of iterations.

For instance,

- an unstable iterative algorithm might produce increasingly large numbers, eventually approaching infinity.
- If an algorithm assumes all inputs are positive integers, it may yield incorrect results for negative or decimal inputs.

- Conversely, a stable algorithm will converge to a finite solution with minimal variation upon further iterations. These three characteristics—convergence, finiteness, and minimal variation—are hallmarks of a robust algorithm.

2.3.1 Numerical Instability Despite the logic flow of an algorithm can cause instability, there is an inherent instability with how computer handles the numbers.

Numerical stability refers to the accuracy of an algorithm can damp out the small fluctuations (errors) in the input data, not magnifying it.

Following two situations are common

- Errors are amplified by the algorithm
- Small perturbations of data generate large changes in the solution → ill conditioned problem

Numerical stability is crucial in algorithm design. It refers to an algorithm's ability to suppress small fluctuations (errors) in the input data, rather than amplifying them.

Two common scenarios that can compromise numerical stability are:

- The algorithm amplifies errors.
- Small perturbations in data cause large changes in the solution, indicating an ill-conditioned problem.

For instance, consider the subtraction of two nearly equal numbers in Python:

This operation is unstable due to the loss of significant digits, leading to a larger error in the result.

[5]:
```
a = 1.0
b = 1.0 + 1e-10
b - a > 1e-10
```

[5]:
```
True
```

Example: Numerically instability `f()` and `g()` are equivalent mathematically, but might not numerically.

$$f = \frac{\sin x^2}{x^2}$$

$$g = \frac{1 - \cos x^2}{x^2}$$

[6]:
```
import math

print((math.cos(5e-5), math.sin(5e-5), math.cos(5e-5) ** 2 + math.sin(5e-5) **␣
↪2))
```

```
(0.99999999875, 4.999999997916667e-05, 1.0)
```

If we have accidentally round one-side

[7]:
```python
import math


def f(x: float) -> float:
    return math.sin(x) ** 2 / x**2


def g2(x: float) -> float:
    a, b = (1 - math.cos(x) ** 2), x**2
    print(f"{a=}, {b=}")
    return a / b


# almost the same
print((f(5e-3), g2(5e-3)))
# g2 > f2
print((f(5e-5), g2(5e-5)))
# g2 > 1 > f2
print((f(1e-5), g2(1e-5)))
```

```
a=2.499979166736832e-05, b=2.5e-05
(0.9999916666944443, 0.9999916666947328)
a=2.4999999848063226e-09, b=2.5e-09
(0.9999999991666668, 0.999999993922529)
a=1.000000082740371e-10, b=1.0000000000000002e-10
(0.9999999999666666, 1.0000000827403708)
```

This is called cancellation error. Cancellation occurs when we subtract two numbers that are almost equal. The result is a loss of significant digits, leading to a larger error in the result.

2.3.2 Ill Condition Problem For a small change in the inputs (the independent variables or the right-hand-side of an equation), there is a large change in the answer or dependent variable. This means that the correct solution/answer to the equation becomes hard to find.

$$Ax = b$$

[8]:
```python
import numpy as np

A = np.array([[0.780, 0.5630], [0.913, 0.659]])
b = np.array([0.217, 0.254])

solve_a_b = np.linalg.solve(A, b)
print(f"Solution of Ax = b: {solve_a_b}")

# Different result with small addition of E
err = np.array([[0.001, 0.001], [-0.002, -0.001]])
solve_Aerr_b = np.linalg.solve(A + err, b)
```

```
print(f"Solution of (A+err)x = b: {solve_Aerr_b}")
```

```
Solution of Ax = b: [ 1. -1.]
Solution of (A+err)x = b: [-5.          7.30851064]
```

Explanation The stability of a linear solver, which requires matrix inversion, can be compromised by precision loss during arithmetic operations. This is quantified by the "condition number" of a matrix:

$$\kappa(A) = \left\|A^{-1}\right\| \|A\|$$

The condition number $cond(A)$ helps identify ill-conditioned matrices:

- If $cond(A) \approx 1$, the matrix is well-conditioned.
- If $cond(A) > 1$, the matrix is ill-conditioned. Approximately log10(k) digits are lost when solving a linear system with a condition number k.

Approximately $\log_{10} k$ digits are lost when solving a linear system with a condition number k.

https://math.stackexchange.com/questions/2392992/matrix-condition-number-and-loss-of-accuracy

[9]:
```
# Machine accuracy limit
eps = np.finfo(float).eps
print("eps:", eps)

# When we multiple two float number, the accuracy for each number is up to␣
 ↪sqrt(eps) to reach result at accuracy of eps.
print("max condition:", 1 / np.sqrt(eps))
```

```
eps: 2.220446049250313e-16
max condition: 67108864.0
```

[10]:
```
cond_A = np.linalg.cond(A)
print(cond_A, int(np.log10(cond_A)))
```

```
2193218.99965077 6
```

A matrix is nearly singular (its determinant is close to zero) if its condition number is very large. Such a matrix is not invertible, and computations involving its inverse or the solution of a related system of linear equations are susceptible to significant numerical errors.

[11]:
```
np.linalg.det(A), np.linalg.det(np.linalg.inv(A))
```

[11]:
```
(1.000000000122681e-06, 999999.9998539208)
```

Another example of ill-conditioned problem

[12]:
```
import numpy as np

A = np.array([[1.0, 1.0], [1.0, 1.0001]])
```

```
b = np.array([2.0, 2.0001])

x = np.linalg.solve(A, b)
print(x)  # Small changes in 'b' can lead to large changes in 'x'

b = np.array([2.0, 2.001])

x = np.linalg.solve(A, b)
print(x)
```

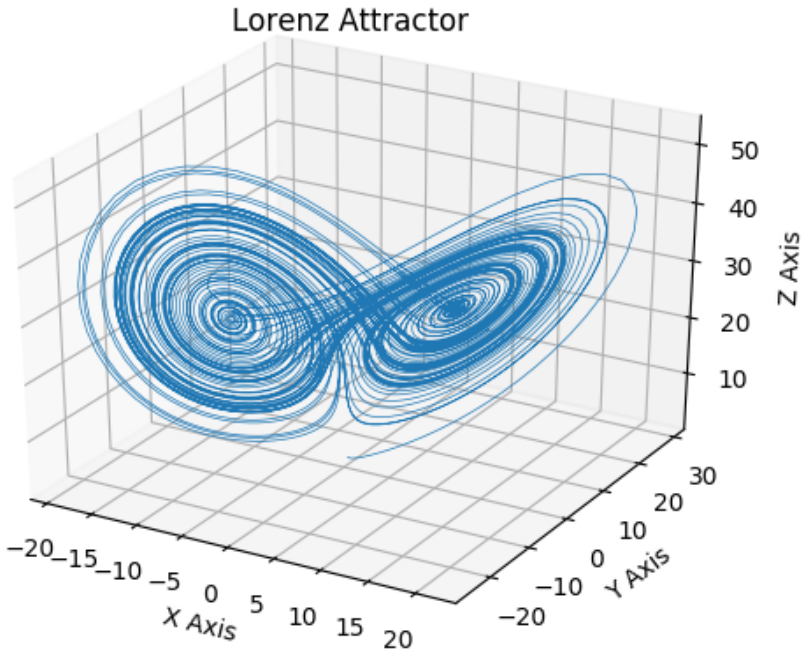
```
[1. 1.]
[-8. 10.]
```

2.3.3 (Optional) Chaos Chaos refers to systems that, while governed by deterministic laws, exhibit apparent randomness due to high sensitivity to initial conditions. This makes future states difficult to predict, resulting in *neither stability nor convergence*.

Examples include:

- N-body problem: Predicting the individual motions of a group of celestial objects interacting with each other gravitationally is notoriously difficult due to the system's chaotic nature.
- Lorenz attractor: This set of differential equations is known for its chaotic solutions, which exhibit sensitive dependence on initial conditions.

Ref to https://matplotlib.org/3.1.0/gallery/mplot3d/lorenz_attractor.html

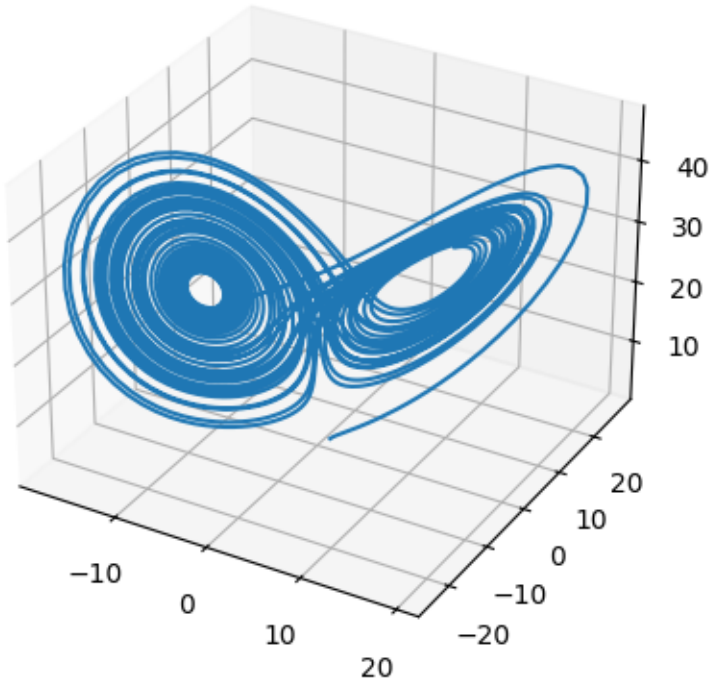


In this code, `lorenz(state, t)` defines the Lorenz system. The function `odeint` from `scipy.integrate` is used to solve this system over the time array `t`, starting from the initial state state0. The resulting states array holds the solution at each time point, demonstrating the system's chaotic behavior.

[13]:
```
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import odeint


def lorenz(state, t):
    x, y, z = state  # Unpack the state vector
    sigma, beta, rho = 10, 2.667, 28  # Lorenz system parameters
    return sigma * (y - x), x * (rho - z) - y, x * y - beta * z  # Derivatives


state0 = [1.0, 1.0, 1.0]
t = np.arange(0.0, 40.0, 0.01)

states = odeint(lorenz, state0, t)

# states now holds the solution to the Lorenz system for each time in 't'
fig = plt.figure()
```

```
ax = plt.figure().add_subplot(projection="3d")
ax.plot(states[:, 0], states[:, 1], states[:, 2])
plt.show()
```

```
<Figure size 640x480 with 0 Axes>
```



2.4 Summary: What makes a good numeric solution

1. Method:
   - It's grounded in mathematical or statistical principles.
   - It's stable, i.e., not significantly affected by small input deviations.
   - It's convergent, i.e., it reaches a solution within a reasonable time and resource usage.
2. Implementation
   - The program is designed to correctly feed inputs to the core algorithm.
   - The code is stable (e.g., loops are finite).
   - There are no coding errors in the core algorithm.
   - It runs on robust hardware with ample memory, potentially leveraging parallel processing.
3. Test and analysis
   - The solution's precision and limitations are thoroughly understood through rigorous testing and analysis.

3. Assignment

To check wether a number is a prime number? Do you think below algorithm is stable and convergent? Please explain why. Update the code to speed it up.

[14]:
```python
# Example prime number code in Python
def is_prime(I):
    if I % 2 == 0:
        return False
    for i in range(3, int(I**0.5) + 1, 2):
        if I % i == 0:
            return False
    return True
```

Appendix: timestamp

[15]:
```python
from datetime import datetime

print(f"Generated on {datetime.now()}")
```

```
Generated on 2024-02-27 21:11:10.449574
```