AP30019

Supplementary Notes

Lu Shan

December, 2024

Preface

This supplementary notes are modified from my personal notes, and i tried to make it as simple as possible. I hope the content is easy to understand and I also add some extension beyond the examination. Prof. Lam is a really good instructor, but its speaking speed is too slow. And i will always go to class, since i treat this behaviour as showing respect to the lecturer's preparation. Then in order to force myself to focus on during the class, i made some notes to reorganize the content in the class. And also searching some additional materials, writing some proofs myself to make this notes complete.
Since AP is a new programme, and lots of courses are open to us for the first time, i think there will be some classmates wondering how to find corresponding materials for them to prepare the test/exam, or to grasp the all content.
This notes is designed for all range of students. If you only care about passing the exam, you could just go through the basic concept, understanding the simple examples etc. Personally, i think that is enough for you to pass Lam's exam.
If you are highly interested into this course, you could spend some time to figure out the exercise that i left. Those exercise is not used for struggling you, but to help you to think deeper. Although some of it will not be covered in exam, if you really think thorough these question, i think it is enough for you to get a satisfactory grade, and good for your future study in this area.
I try to deliver some related theorem or example in the note, and also specify the course that you need to take in order to delve into this subject. Therefore, this is not for our AP students. To reiterate, this note is not difficult to read, and also not difficult to solve the exercise, if you really think it for considerable time.
Since the time limited, i only spent roughly 20 hours to finish this notes, then if there are any typo, or you feel like it is useful and want to add more details, feel free to email me. My personal email is 33at20@gmail.com
Lastly, hope you enjoy the journey to data science and mathematics.

Contents

1 Nonlinear Solution

1.1 Fixed Point Iterations

To solve question like: x_0, s.t. $x_0 = f(x_0)$, for certain function f.
The iteration equation: $x_{n+1} = f(x_n)$.

1.1.1 Derivation:

Let's assume the function f is defined on the interval [a, b], s.t. $|f'([a, b])| < 1$, which means $\forall\, x \in [a, b], f'(x) < 1$, provided that f' exists. And another assumption is that $f([a, b]) \subseteq [a, b]$. So we first show that there exists a fixed point, i.e. $\exists\, x_0 \in [a, b], s.t. x_0 = f(x_0)$, primarily based on second assumption:

Existence:

Firstly, case i: if $f(a) = a$ or $f(b) = b$, then it is obvious that there exists a fixed point.
Case ii: if not, then we must have $f(a) > a$ and $f(b) < b$, then we set a new function called $g(x) = f(x) - x$, where $x \in [a, b]$, so we know that $g(a) > 0$, $g(b) < 0$, which indicates that there is a root s.t. $g(x_0) = 0$.

Uniqueness:

Let's suppose that we have two distinct fixed points, denoted as $p, q \in [a, b]$, therefore, we have $|p-q| = |f(p)-f(q)| = |f'(\xi)| \cdot |p-q| < |p-q|$, which is a contradiction, where $\xi \in [a, b]$.

1.1.2 Error Analysis:

Fixed points method can't be analysed generally, only can be checked case by case.

1.1.3 Implementation:

```python
def FixedPoints(f, InitialGuess, tol, MaxIter):
    x_0 = InitialGuess
    for i in range(MaxIter):
        x_1 = f(x_0)
        if abs(x_1 - x_0) <= tol:
            return x_1
        x_0 = x_1
    return x_0
```

Example 1.1: Define functions

$$f(x) = rx(1-x) \quad \text{and} \quad g(x) = f(f(x)).$$

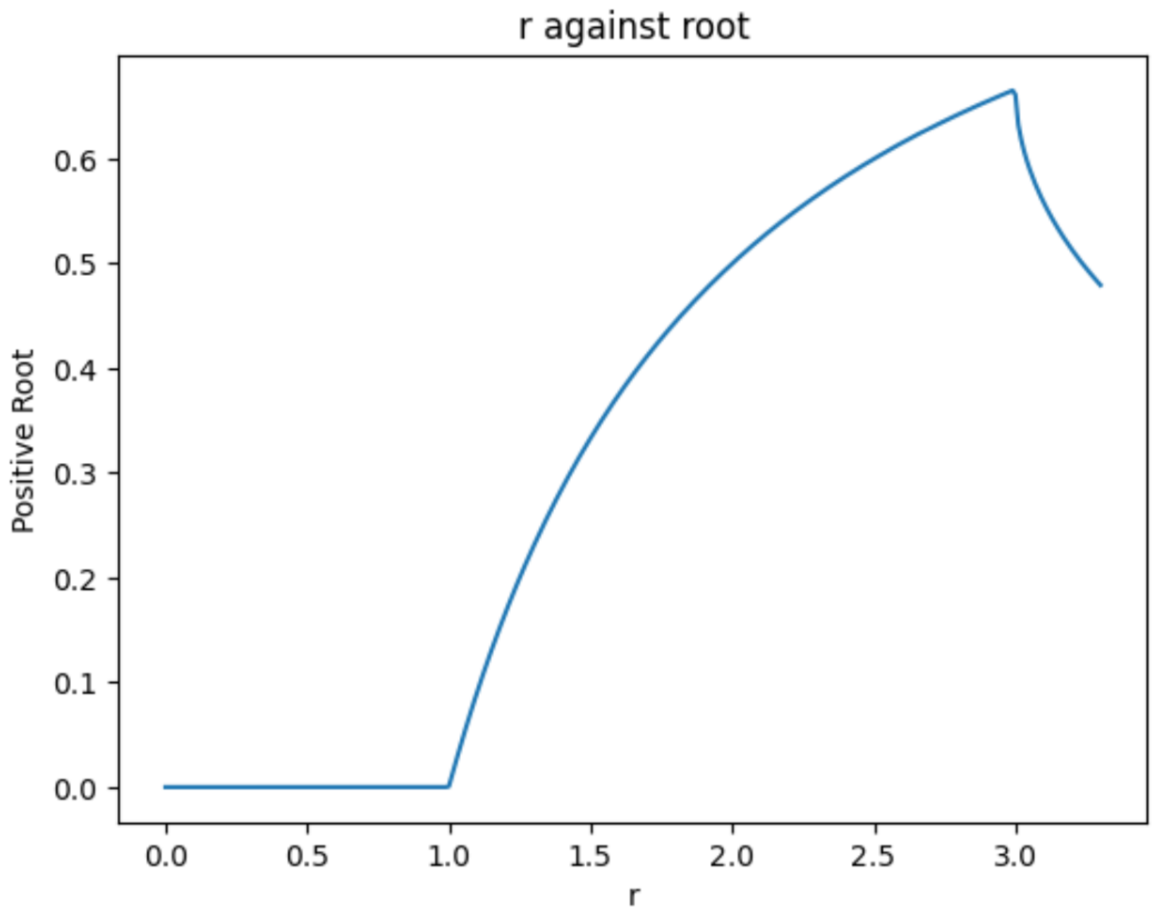
The equation

$$x = g(x)$$

can be solved using the fixed-point iteration method with an initial value $x_0 = 0.5$. Write a computer program which solves x multiple times for $r = 0.01, 0.02, \ldots, 3.30$ with a tolerance of 10^{-6}. Submit the program and a plot of x against r.

Solution:

```python
def FixedPoints(f, r, InitialGuess, tol, MaxIter):
    x_0 = InitialGuess
    for i in range(MaxIter):
        x_1 = f(x_0, r)
        if abs(x_1 - x_0) <= tol:
            return x_1
        x_0 = x_1
    return x_0


import matplotlib.pyplot as plt


def f(x, r):
    return r * (1 - x) * x


def g(x, r):
    return f(f(x, r), r)


MaxIter = 1000
InitialGuess = 0.5
tol = 10 ** (-6)

rList = [i/100 for i in range(0, 331, 1)]
RootList = []

for r in rList:
    root = FixedPoints(g, r, InitialGuess, tol, MaxIter)
    RootList.append(root)

plt.title('roots-r\'s')
plt.xlabel('r')
plt.ylabel('roots')

plt.plot(rList, RootList)
plt.show()
```



1.2 Bisection Method

Bisection method is a powerful method simply due to its simplicity and convenience. It can be used to solve any type of questions, provided that the root should be bounded by certain interval, and the performance will be better if it is monotonic in such interval.
The iteration criteria is that bisecting a new interval, whose length is half of the original, then choose one of these two, which contains the root that we want.

1.2.1 Derivation:

No detailed derivation and proof for existence in this part, since before we use it, we should make sure that there exists one. And theoretical guarantee is given by Bolzano's theorem.

1.2.2 Error Analysis:

Let's denote each iteration, we generate a new interval whose bounds are a_n, b_n, then our approximation would be $p_n = \frac{a_n+b_n}{2}$, and the true root is p^*.
Then $|p_n - p^*| \leq \frac{b_n-a_n}{2} = \frac{1}{2^2}(b_{n-1}-a_{n-1}) = \cdots = \frac{1}{2^n}(b-a)$, so we know the rate of convergence is at order $O(\frac{1}{2^n})$, but notice that it is a bound, which means the true error is unknown.

1.2.3 Implementation:

```python
def Bisection(f, a, b, tol):
    left, right = a, b
    f_1, f_2 = f(left), f(right)
    if f_1 == 0:
        return left
    elif f_2 == 0:
        return right
    while right - left > tol:
        mid = (right - left) / 2 + left
        f_3 = f(mid)
        if f_3 == 0:
            return mid
        if f(mid) * f_2 <= 0:
            left = mid
            f_1 = f_3
        else:
            right = mid
            f_2 = f_3
    return (left + right) / 2
```

Example 1.2: Using a computer program, find the positive root of

$$x^3 = \sin x \tag{10}$$

using the bisection method with a tolerance of 10^{-8}. Submit your source code and program output. State your numerical answer and the number of iterations used.

Solution:

```python
def Bisection(f, a, b, tol):
    left, right = a, b
    f_1, f_2 = f(left), f(right)
    cnt = 0
    if f_1 == 0:
        return left, cnt
    elif f_2 == 0:
        return right, cnt
    while right - left > tol:
        mid = (right - left) / 2 + left
        f_3 = f(mid)
        cnt += 1
        if f_3 == 0:
            return mid, cnt
        if f(mid) * f_2 <= 0:
            left = mid
            f_1 = f_3
        else:
            right = mid
```

```
                f_2 = f_3
    return (left + right) / 2, cnt


from math import sin


def f(x):
    return x ** 3 - sin(x)

# Case 1
a = 0.5
b = 1.5
tol = 10 ** (-8)

# # Case 2
# a = -1.5
# b = -0.5
# # -0.9286263100802898 27

# # Case 3
# a = -1
# b = 1
# # 0.0 1

# # Case 3
# a = -1.5
# b = 1
# # -0.9286263128742576 28

# # Case 4
# a = -1
# b = 1.5
# # 0.9286263128742576 28

root, count = Bisection(f, a, b, tol)
print(root, count)
```

Remark:
The Bisection Method has a number of significant drawbacks.
Firstly it is very slow to converge in that N may become quite large before $|p_n - p^*|$ becomes sufficiently small.
Also it is possible that a good intermediate approximation may be inadvertently discarded.

1.3 Newton-Raphson Method

To solve general question like: x_0, s.t. $f(x_0) = 0$, for certain function f.
The iteration equation: $p_{n+1} = p_n - \frac{f(p_n)}{f'(p_n)}$.

1.3.1 Derivation:

Let's assume our interest lies in [a, b], and f is $C^2[a,b]$, then we may pick our initial guess p_0, then denotes our true value as p^*. We want our guess as close as possible, then we could expand the function using Tayor expansion at $p_0: f(p) = f(p_0)+f'(p_0)(p-p_0)+\frac{f''(\zeta)}{2}(p-p_0)^2$, where $\zeta \in (a,b)$. Since we pick the p_0 as close as p^*, so the square error term will be much smaller tham the first order, and we try to solve the next best guess according to our current state, then set $f(p) = 0 = f(p_0) + f'(p_0)(p-p_0)$, which reduces to $p = p_0 - \frac{f(p_0)}{f'(p_0)}$.
From the derivation we know that the prerequisite of finding a nice root is based on our choice of initial guess, then let's look at the error analysis.

1.3.2 Error Analysis:

We denote the error of n th term as ε_n, then we know that $\varepsilon_{n+1} = p^* - p_{n+1} = p^* - (p_n - \frac{f(p_n)}{f'(p_n)}) = \varepsilon_n + \frac{f(p_n)}{f'(p_n)}$, then we expand f at p_n, in order to find the relationship between ε_{n+1} and $\varepsilon_n: 0 = f(p^*) = f(p_n)+f'(p_n)\varepsilon_n+\frac{\varepsilon_n^2}{2}f''(p_n)$, then put this back to the original equation, we have: $|\varepsilon_{n+1}| = |\varepsilon_n+\frac{-f'(p_n)\varepsilon_n-\frac{\varepsilon_n^2}{2}f''(p_n)}{f'(p_n)}| = \frac{\varepsilon_n^2}{2}\cdot|\frac{f''(p_n)}{f'(p_n)}|$, so we know that is the term, $|\frac{f''(p_n)}{f'(p_n)}|$, doesn't explode, then our error converges qudratically.

1.3.3 Implementation:

```python
def NewtonRaphson(f, df, x, tol):
    xLast = x
    fx = f(x)
    dfx = df(x)
    x -= fx / dfx
    while abs(xLast - x) > tol:
        xLast = x
        fx, dfx = f(x), df(x)
        x -= fx / dfx
    return x
```

Example 1.3: In a diffraction and interference experiment, light of wavelength λ goes through a single slit of width d. The diffracted light intensity I at angle θ is given by:

$$I = I_0(\frac{\sin(\alpha)}{\alpha})^2$$

, where $\alpha = (\pi d/\lambda)\sin(\theta)$ and I_0 is a constant.

(a) Plot a graph of I/I_0 against α which includes 3 maxima on each side of the y-axis.

(b) From your graph, roughly estimate the values of α_i ¿ 0 (i = 1, 2, 3) at which the first, second and third order maxima occur.

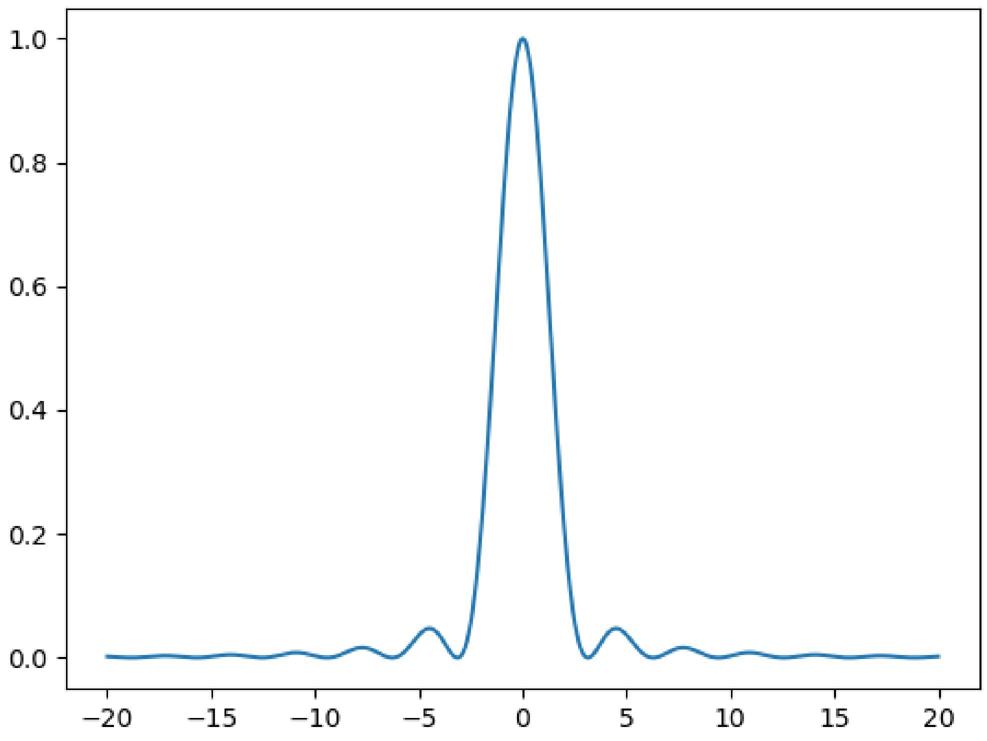
(c) To evaluate α_i accurately, the roots of a nonlinear equation F(α) = 0 should be calculated. Find F(α).

(d) Hence, write a computer program for finding a root of your nonlinear equation using the Newton-Raphson method up to a tolerance of 10-8. Run your program 3 times with appropriate initial guesses to locate the 3 maxima. In each case, write down your answer, initial guess, and the number of iterations used. Submit only the version of your program for finding the first order maximum.

(e) What are the values of sin(θ) for the 3 maxima?

Solution: (a)They are the first, second, and third order maxima.

```
import matplotlib.pyplot as plt
from math import *

xInterval = sorted([i / 100 for i in range(1, 2000)] + [- i / 100 for
    i in range(1, 2000)])
yInterval = [(sin(i) / i) ** 2 for i in xInterval]

plt.plot(xInterval, yInterval)
plt.show()
```



(b) We could guess from the graph that our three maxima are close to 4.3, 7.4, 11 respectively

```
firstGuess = 4.3
secondGuess = 7.4
thirdGuess = 11
```

(c)

$$\frac{I}{I_0} = (\frac{\sin(\alpha)}{\alpha})^2 \equiv g(\alpha)$$

$$\frac{dg(\alpha)}{d\alpha} = 2 \cdot \frac{\sin(\alpha)}{\alpha} \cdot \frac{\alpha\cos(\alpha) - \sin(\alpha)}{\alpha^2} \equiv 2h(\alpha)h'(\alpha), \frac{d^2g(\alpha)}{d\alpha^2} = 2[(h'(\alpha))^2 + h(\alpha)h''(\alpha)]$$

$$\text{If } h(\alpha) = 0, \frac{d^2g(\alpha)}{d\alpha^2} = 2(h'(\alpha))^2 > 0,$$

$$\text{If } h'(\alpha) = 0 \Rightarrow \tan(\alpha) = \alpha, \text{then } \frac{d^2g(\alpha)}{d\alpha^2} = 2h(\alpha)h''(\alpha) = \frac{-2\sin^2(\alpha)}{\alpha^2} < 0.$$

, which shows that only the root $h'(\alpha)$ gives us the maximum. Then F$(\alpha) = h'(\alpha) = \alpha\cos(\alpha) - \sin(\alpha)$.

```
def f(x):
    return x*cos(x) - sin(x)

def df(x):
    return - x*sin(x)
```

(d)

```
def NewtonRaphson(f, df, x, tol):
    xLast = x
    fx = f(x)
    dfx = df(x)
    x -= fx / dfx
    cont = 1
    while abs(xLast - x) > tol:
        xLast = x
        fx, dfx = f(x), df(x)
        x -= fx / dfx
        cont += 1
        print(cont,x)
    return x, cont

firstRoot = NewtonRaphson(f, df, firstGuess, 10e-8)
secondRoot = NewtonRaphson(f, df, secondGuess, 10e-8)
thirdRoot = NewtonRaphson(f, df, thirdGuess, 10e-8)

print(f"The first root is {firstRoot[0]}, and the number of iteration used are {firstRoot[1]}")
print(f"The second root is {secondRoot[0]}, and the number of iteration used are {secondRoot[1]}")
print(f"TThe third root is {thirdRoot[0]}, and the number of iteration used are {thirdRoot[1]}")
```

(e)

```
alpha_1 = firstRoot[0]
alpha_2 = secondRoot[0]
alpha_3 = thirdRoot[0]

print(f"The first maxima's sine is {alpha_1}.")
print(f"The second maxima's sine is {alpha_2}.")
print(f"The third maxima's sine is {alpha_3}.")
```

Then the first one is: $\sin(\theta_1) = 4.493409457909064\frac{\lambda}{\pi d}$,
and second one is: $\sin(\theta_2) = 7.725251836937707\frac{\lambda}{\pi d}$,
lastly, third one is: $\sin(\theta_2) = 10.904121659428899\frac{\lambda}{\pi d}$.

Remark:

After the whole process, we know that Newton Raphson Method converges really fast, but there are three problems.

First one is that when the derivative of potential function is difficult to find, then it will cause problems.

Secondly, for some of situation, when the initial guess is bad, we will encounter oscillation, which impedes us to give the correct approximation.

Lastly, it can't figure out the question with large slope around the suspicious interval.

1.4 Extension Method

In above discussion, we notice that in order to use these numerical method, we always need to gain the suspicious interval in advance, then intuitively, as human, we can pick a fixed interval, and check its end points' sign, if their product is negative, by Bolzano's theorem, we know it hides at least a potential root.

1.4.1 Incremental Search Method

```python
def RootSearch(f, a, b, dx):
    x_1, f_1 = a, f(a)
    x_2, f_2 = a + dx, f(a + dx)
    while f_1*f_2 > 0:
        if x_1 >= b:
            return None, None
        x_1, f_1 = x_2, f_2
        x_2 = x_2 + dx
        f_2 = f(x_2)
    else:
        return x_1, x_2
```

But, this method has some disadvantages:

(1) It is possible to miss two closely spaced roots if the dx is too large.

(2) A double root(two roots coincide) will not be detected.

(3) Certain singularities(poles) of f(x) can be mistaken for roots.

1.4.2 Modified Newton Raphson Method with Bisection

When we know all the tools that we need for getting a numerical solution, i also want to give some other more practical used method. The first one would be the modified version of Newton-Raphson method, which integrates into bisection, when it extrapolates out the given interval or the slope is really close to zero.

```python
def NewtonRaphsonModified(f, df, a, b, tol, MaxIter):
    fa = f(a)
    if fa == 0:
        return a
    fb = f(b)
    if fb == 0:
        return b
    x = 0.5 * (a + b)

    for i in range(MaxIter):
        fx = f(x)
        if fx == 0.0:
            return x
        # Tighten the brackets on the root
        if (fa * fx) < 0:
            b = x
        else:
            a = x
        # Try a Newton-Raphson step
        dfx = df(x)
        # If division by zero, push x out of bounds
        if abs(dfx) < tol:
            dx = -fx / dfx
        else:
            dx = b - a
        x = x + dx
        # If the result is outside the brackets, use bisection
        if (b - x) * (x - a) < 0.0:
            dx = 0.5 * (b - a)

        x = a + dx
        # Check for convergence
        if abs(dx) < tol * max(abs(b), 1.0):
            return x
    return None
```

1.4.3 Ridder's Method

And another method is also a really good one, called ridder's method, which can be showed converges quadratically, but we wouldn't do that here. It is worthy to mention, this method is motivated by secant and false position method related to interpolation, here we use extra exponential terms to decorate our original points such that the three points, where the third point is the mean of boundaries, are in one line.

```
def Ridder(f, a, b, tol, MaxIter):
    x_1, x_2 = a, b
    f_1 = f(x_1)
    if f_1 == 0:
        return a
    f_2 = f(x_2)
    if f_2 == 0:
        return b
    for i in range(MaxIter):
        x_3 = (x_2 - x_1) / 2 + x_1
        f_3 = f(x_3)
        s = (f_3**2 - f_1*f_2)**(0.5)
        if s == 0:
            return None
        dx = (x_3 - x_1)*f_3 / s
        if f_1 - f_2 < 0:
            x_4 = x_3 - dx
        else:
            x_4 = x_3 + dx
        f_4 = f(x_4)
        if i > 0:
            if abs(x_4 - Old4) < tol*max(abs(x_4), 1):
                return x_4
        Old4 = x_4
        if f_3*f_4 > 0:
            if f_4*f_1 < 0:
                x_2 = x_4
                f_2 = f_4
            else:
                x_1 = x_4
                f_1 = f_4
        else:
            x_1, f_1 = x_3, f_3
            x_2, f_2 = x_4, f_4
    return None
```

We could use one example to give you some taste of solving equations numerically.

Example 1.4: Find the root of this using ridder method, $f(x) = x^3 - 10x^2 + 5$.

First, we could guess any number to begin with, why it wouldn't be zero and one, and set a small step to search a suspicious interval.

```
x_0 = 0
x_1 = 1
dx = 0.2

def f(x):
  return x**3 - 10*x**2 + 5

print(RootSearch(f, x_0, x_1, dx))
```

Then it give us (0.6, 0.8). So we could use this suspicious interval to try each our method.

```
a = 0.6
b = 0.8

def df(x):
  return 3 * x ** 2 - 20 * x

x = (a + b) / 2
tol = 10e-8
MaxIter = 30

print("The result of Ridder's Method is: ", Ridder(f, a, b, tol,
    MaxIter))
```

1.5 Exercise

(1)Let f be a function defined on all of $\mathbb{R}$, and assume there is a constant $0 < c < 1$ and for all $x, y \in \mathbb{R}$:

$$|f(x) - f(y)| \leq c|x - y|.$$

(a) Show that f is continuous on $\mathbb{R}$.

(b) Pick some point $y_1 \in \mathbb{R}$ and construct the sequence

$$(y_n) = (y_1, f(y_1), f(f(y_1)), \ldots).$$

(c) In general, if y_n is defined as above, prove that the resulting sequence (y_n) is a Cauchy sequence. Hence we may let $y = \lim_{n\to\infty} y_n$.

(d) Finally, prove that if x is any arbitrary point in $\mathbb{R}$, then the sequence

$$(x, f(x), f(f(x)), \ldots)$$

converges to y defined in (b).

(2) Show that if $f'(x) < 1$ $\forall x \in [a, b]$, then fixed point methos will give you the solution in the corresponding region.

2 Approximation of data

2.1 Lagrange Interpolation

2.1.1 How to generate the idea

If we have, let's say, five data points, i.e. $\{(x_i, y_i)\}_5$, then we believe that they all come from the same curve, as a result, we want to get a curve that perfectly goes through these five points. And i think this question is really simple, we know this from junior school, because we have five data points, then we could solve a polynomial with five unknown, which is a fourth degree polynomial. To put it more clearly, we could write this in matrix form as below:

$$\underbrace{\begin{bmatrix} 1 & x_0 & x_0^2 & x_0^3 & x_0^4 \\ 1 & x_1 & x_1^2 & x_1^3 & x_1^4 \\ 1 & x_2 & x_2^2 & x_2^3 & x_2^4 \\ 1 & x_3 & x_3^2 & x_3^3 & x_3^4 \\ 1 & x_4 & x_4^2 & x_4^3 & x_4^4 \end{bmatrix}}_{\text{Vandermonde matrix}} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_4 \\ a_5 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_4 \\ y_5 \end{bmatrix}$$

Then we could easily solve this by simple elementary LU decomposition. We will see this later, since Lagrange gives us a better understanding intuitively of this formulation. (Actually, in the end, we will see that they are the same.)

Let's first check the simplest case, where there are only two data points, i.e. $(x_0, y_0), (x_1, y_1)$, we want to find a equation that passes through these two points, which is quite easy, because we could lend some idea from the way that we write binomial formulas: $x \sim Bin(p) \Rightarrow \boldsymbol{P}(x) = (p)^x \cdot (1-p)^{(1-x)}$.

Then we could write our simplest two sample data into: $y = y_0 \cdot \frac{(x-x_1)}{(x_0-x_1)} + y_1 \cdot \frac{(x-x_0)}{(x_1-x_0)}$, in order to see the pattern, we could see the case with three samples: $y = y_0 \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} + y_1 \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} + y_2 \frac{(x-x_0)(x-x_1)}{(x_0-x_0)(x_0-x_1)}$.

To put more general form, we could write as: $p_m(x) = y_0 L_0^m(x) + y_1 L_1^m(x) + y_2 L_2^m(x) + \cdots + y_m L_m^m(x)$, where $L_i^m = \prod_{j=0,\, i\neq j}^{m} \frac{(x-x_j)}{(x_i-x_j)}$ is a polynomial of x with degree m.

Fine, then back to the Vandermonde matrix(denoted as $\mathbb{V}$), why we said that there are the same with Lagrange method? Let's first check its determinant:

$$\det(\begin{bmatrix} 1 & \cdots & x_0^m \\ \vdots & \ddots & \vdots \\ 1 & \cdots & x_m^m \end{bmatrix}) = \prod_{0 \le i < j \le m} (x_i - x_j).$$

, which gives us the inverse:

$$\mathbb{V}^{-1} = \mathcal{L} = \begin{bmatrix} L_{0,0} & \cdots & L_{0,m} \\ \vdots & \ddots & \vdots \\ L_{m,0} & \cdots & L_{m,m} \end{bmatrix}$$

, where $L_i^m = \sum_{j=0}^{m} L_{i,j} x^i$.

Then from here, we see their strong relationship, and actually they are the same.

2.1.2 How to implement with python

```python
def P(x, m):
    px = 0  # the initial value
    for i in range(m):
        Lmj = 1  # Lagrange factor
        for j in range(m):
            if i != j:
                Lmj *= (x - X[j]) / (X[i] - X[j])
        px += Y[i] * Lmj  # accumulated together
    return px
```

Example 2.1: Write down the polynomial interpolation function y(x) interpolating the points (0,4), (3,3) and (6,-2) in terms of Lagrange polynomials. There is no need to simplify your expressions. Hence calculate an interpolated value of y(2).

Solution:
$y(x) = 4\frac{(x-3)(x-6)}{18} - 3\frac{x(x-6)}{9} - 2\frac{x(x-3)}{18}$
Then y(2) $= \frac{34}{9}$.

```python
def y(x):
    return (2 / 9) * (x - 3) * (x - 6) - (1 / 3) * x * (x - 6) - (1 / 9) * x * (x - 3)

print(f"The␣predicted␣value␣is␣{y(2)}.")

# check

xData = [0, 3, 6]
yData = [4, 3, -2]
m = len(xData)

def P(x, m):
    px = 0  # the initial value
    for i in range(m):
        Lmj = 1  # Lagrange factor
        for j in range(m):
            if i != j:
                Lmj *= (x - xData[j]) / (xData[i] - xData[j])
        px += yData[i] * Lmj  # accumulated together
    return px

print(f"The␣value␣for␣checking␣is␣{P(2,m)}.")
```

Example 2.2: The function:

$$f(x) = \tanh(x)$$

can be approximated by the Lagrange polynomials.

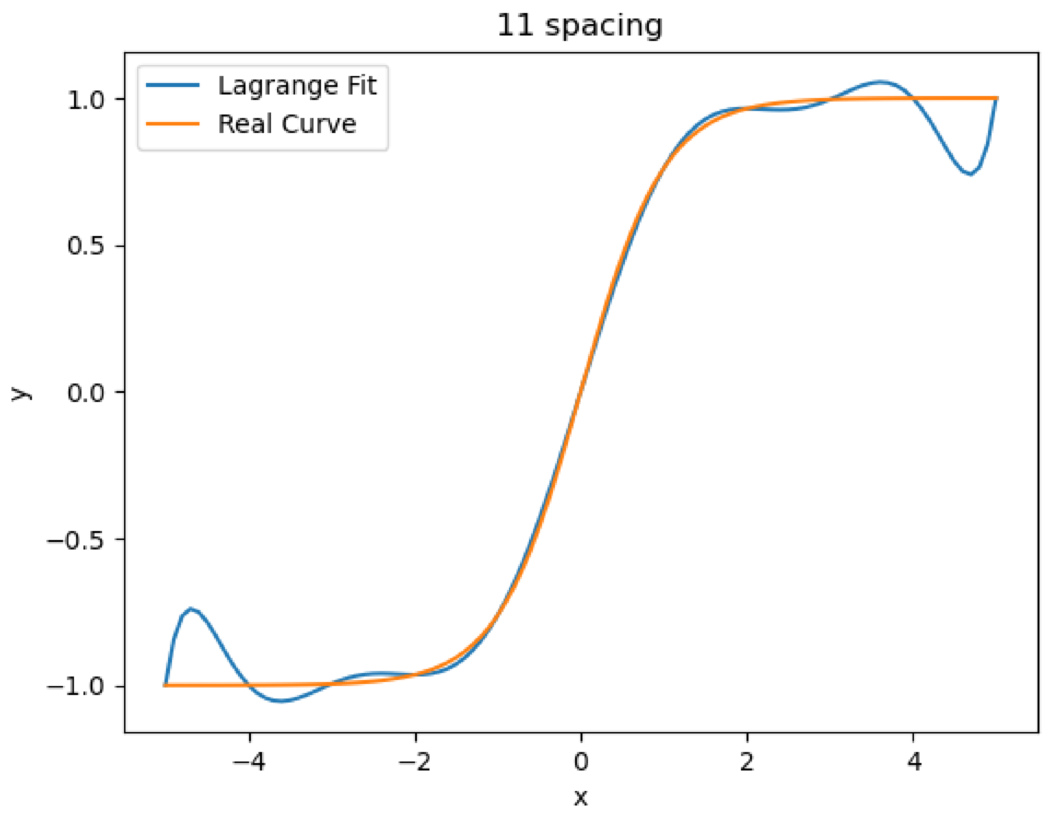
(a) Consider a set of data points (x, f(x)) given by 11 values of x equally spaced in the interval [-5, 5]. Let p(x) be the Lagrange interpolation function based on this 11 points. By writing a computer program, calculate f(x) and p(x) for 100 values of x in the same interval. Hence plot both functions on the same graph.

(b) Repeat the procedure by interpolating 21 equally spaced points in the same interval. Has the quality of the polynomial approximation been improved by interpolating more points on the function? Why?

Solution

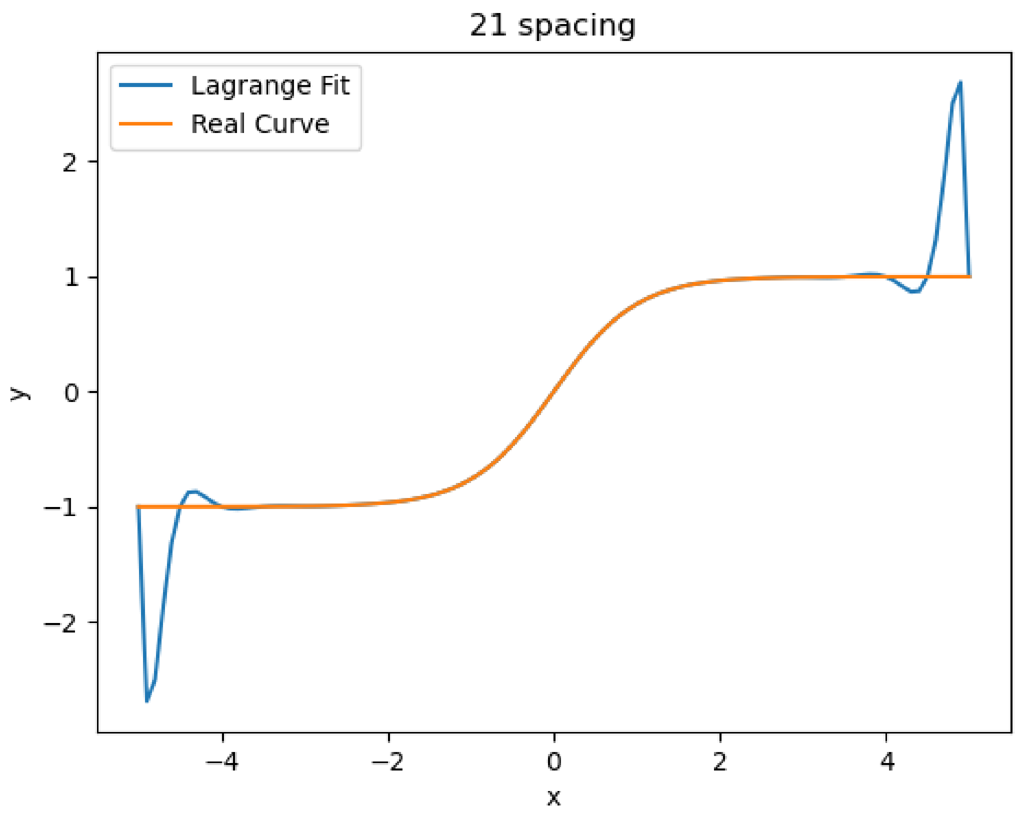
(a)

```python
def P(x, m, X, Y):
    px = 0  # the initial value
    for i in range(m):
        Lmj = 1  # Lagrange factor
        for j in range(m):
            if i != j:
                Lmj *= (x - X[j]) / (X[i] - X[j])
        px += Y[i] * Lmj  # accumulated together
    return px


def f(x):
    return tanh(x)


X_1 = [-5 + i for i in range(11)]
Y_1 = [f(i) for i in X_1]
m = len(X_1)

testX = [-5 + i / 10 for i in range(101)]
testY_1 = [P(i, m, X_1, Y_1) for i in testX]
realY = [f(i) for i in X_1]

plt.plot(testX, testY_1, label="Lagrange␣Fit")
plt.plot(testX, realY, label="Real␣Curve")
plt.title("11␣spacing")
plt.legend()
plt.show()
```



(b)

```
X_2 = [-5 + i/2 for i in range(21)]
Y_2 = [f(i) for i in X_2]
m = len(X_2)

testY_2 = [P(i, m, X_2, Y_2) for i in testX]

plt.plot(testX, testY_2, label="Lagrange Fit")
plt.plot(testX, realY, label="Real Curve")
plt.xlabel("x")
plt.ylabel('y')
plt.legend()
plt.title("21 spacing")
plt.show()
```



The approximation becomes even worse at the tail, the reason is that the change in the tail of tanh(x) is really slow, and which is really difficult to be learnt from 20ish order polynomial function, since the whole change rate is dominated by the middle body part, then the huge deviation at the tail appears in order to go through all the points.
Or more theoretical, we could think about, when more points used, higher order our polynomial is, then the error is more sensitive to the change, which is not what we want to see in the flat tail situation. In order to fix it, in principle we could choose more points in the flat tail, and for details, we could use Chebyshev nodes.

2.2 Linear Least Square Fit

In general, we define the least square as minimizing the loss function of square sum of residual, i.e the difference between the model function and observable value, $\mathcal{S}(c_0, c_1, \cdots, c_k) = \sum_{i=1}^{n}(p(x_i) - y_i)^2$, where $p(x) = c_0 p_0(x) + c_1 p_1(x) + \cdots + c_k p_k(x)$.
Therefore, we will check the below three cases: fit for straight line, polynomials and nonlinear one. Then next we will briefly introduce the weighted linear square, or to be specific, change our loss model to weighted one.

2.2.1 Fit to straight line

Derivation

We denote our targeted function as: $p(x) = c_0 + c_1 x$.
Then we could define our loss function: $\mathcal{S}(c_0, c_1) = \sum_{i=1}^{n}(p(x_i) - y_i)^2$, which is what we want to minimize:

$$\frac{\partial}{\partial c_0}\mathcal{S}(c_0, c_1) = \sum_{i=1}^{n} 2(c_0 + c_1 x_i - y_i) = 0$$
$$\frac{\partial}{\partial c_1}\mathcal{S}(c_0, c_1) = \sum_{i=1}^{n} 2x_i(c_0 + c_1 x_i - y_i) = 0$$

, which gives us the following results:

$$c_0 = \frac{\sum y_i \sum x_i^2 - \sum x_i \sum x_i y_i}{n \sum x_i^2 - (\sum x_i)^2}$$
$$c_1 = \frac{n \sum x_i y_i - \sum x_i \sum y_i}{n \sum x_i^2 - (\sum x_i)^2}$$

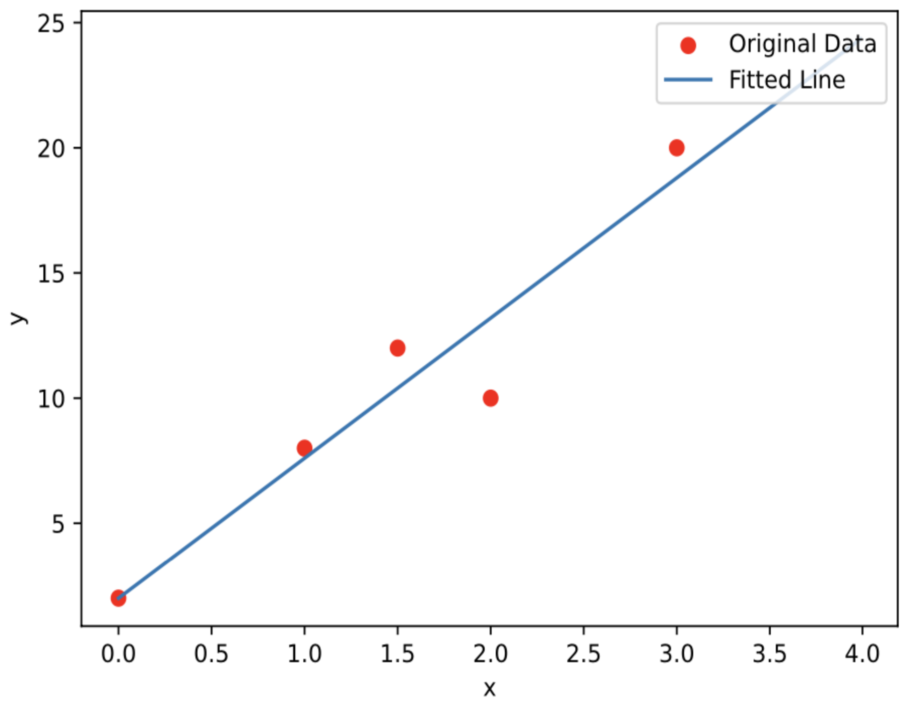
Implementaion

```
def LeastSquareLine(xData, yData):
    n = len(xData)
    sumX = sum(xData)
    sumY = sum(yData)
    cross = sum([xData[i]*yData[i] for i in range(n)])
    sumX2 = sum([xData[i]**2 for i in range(n)])

    c_0 = (sumY*sumX2 - sumX *cross)/(n*sumX2 - (sumX)**2)

    c_1 = (n*cross - sumX*sumY)/(n*sumX2 - (sumX)**2)

    return c_0, c_1
```

Example 2.3: Consider 5 data points, (x,y) = (0, 2), (1, 8), (1.5, 12), (2, 10) and (3, 20). Write a computer program which performs a least square fit of the data to a straight line. It should output the fitted parameters. Hence, write down the program output. Plot the data and the fitted line on the same graph.

Solution:

```
xData = [0, 1, 1.5, 2, 3]
yData = [2, 8, 12, 10, 20]


def LeastSquareLine(xData, yData):
    n = len(xData)
    sumX = sum(xData)
    sumY = sum(yData)
    cross = sum([xData[i] * yData[i] for i in range(n)])
    sumX2 = sum([xData[i] ** 2 for i in range(n)])
    c_0 = (sumY * sumX2 - sumX * cross) / (n * sumX2 - (sumX) ** 2)

    c_1 = (n * cross - sumX * sumY) / (n * sumX2 - (sumX) ** 2)

    return c_0, c_1


a, b = LeastSquareLine(xData, yData)

print(f"The␣estimated␣intercept␣is␣{a},␣and␣the␣predicted␣slope␣is␣{b
    }.")

# Mimic the process
xRange = [i / 100 for i in range((min(xData)) * 100, (max(xData) + 1)
    * 100)]
yRange = [a + b * x for x in xRange]

import matplotlib.pyplot as plt

plt.scatter(xData, yData, color="red")
plt.plot(xRange, yRange)
plt.xlabel("x")
plt.ylabel("y")
plt.legend(("Original␣Data", "Fitted␣Line"), loc="upper␣right")
plt.show()
```



Moreover, we could modify our loss function, to make it has something to do with our variance,

since variance denotes the uncertainty. Thus we could easily see that when variance is high, it may be influential point, then its weight should be low.
Hence: $\mathcal{S}(c_0, c_1) = \sum\limits_{i=1}^{n} (\frac{p_{c_0,c_1}(x_i) - y_i}{\sigma_i})^2$.

Example 2.4: A set of N data points $(x_i,\, y_i)$ for i = 1,2...N can be fitted to:

$$f(x) = a\sin(x)$$

,where a is a fitted parameter. Also, y_i has a r.m.s. error σ_i. On the basis of linear least square fit, derive a formula for a.

Solution:
We first define its weighted loss function as:

$$\mathcal{S}(a) = \sum_{i=1}^{N} (\frac{y_i - a\sin(x_i)}{\sigma_i})^2$$

, then we naturally want this to be as smallest as we could, we find its gradient, and set it to zero:

$$\frac{\partial}{\partial a}\mathcal{S}(a) = -2\sum_{i=1}^{N} \frac{y_i - a\sin(x_i)}{\sigma_i^2} \cdot \sin(x_i) = 0$$

, after some algebra, we get the weighted optimal parameter:

$$a = \sum_{i=1}^{N} \frac{y_i \sin(x_i)}{\sigma_i^2} \bigg/ \sum_{i=1}^{N} \frac{\sin^2(x_i)}{\sigma_i^2}.$$

2.3 Fit to Polynomials

Then more general, we also could fit this function by polynomials, our targeted function would become: $p(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_k x^k$.
Next, we minimize that:

$$\begin{aligned}\min_{c_0, c_1, \cdots, c_k} \mathcal{S}(c_0, c_1, \cdots, c_k) &= \sum (p(x_i) - y_i)^2 \\ &= \sum (c_0 + c_1 x_i + c_2 x_i^2 + \cdots + c_k x_i^k - y_i)^2\end{aligned}$$

, which gives us the below system of linear equations:

$$\begin{aligned}&\frac{\partial}{\partial c_0}\sum (c_0 + c_1 x_i + \cdots + c_k x_i^k - y_i)^2 \equiv c_0 n + c_1 \sum x_i + c_2 \sum x_i^2 + \cdots + c_k \sum x_i^k - \sum y_i = 0 \\ &\frac{\partial}{\partial c_1}\sum (c_0 + c_1 x_i + \cdots + c_k x_i^k - y_i)^2 \equiv c_0 \sum x_i + c_1 \sum x_i^2 + c_2 \sum x_i^3 + \cdots + c_k \sum x_i^{k+1} - \sum x_i y_i = 0 \\ &\qquad\qquad\qquad\qquad \cdots\cdots \\ &\frac{\partial}{\partial c_k}\sum (c_0 + c_1 x_i + \cdots + c_k x_i^k - y_i)^2 \equiv c_0 \sum x_i^k + c_1 \sum x_i^{k+1} + c_2 \sum x_i^{k+2} + \cdots + c_k \sum x_i^{k+k} - \sum x_i^k y_i = 0\end{aligned}$$

From this we put them into matrix form:

$$\begin{bmatrix} n & \sum x_i & \cdots & \sum x_i^k \\ \sum x_i & \sum x_i^2 & \cdots & \sum x_i^{k+1} \\ \vdots & \vdots & \ddots & \vdots \\ \sum x_i^k & \sum x_i^{k+1} & \cdots & \sum x_i^{k+k} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_k \end{bmatrix} = \begin{bmatrix} \sum y_i \\ \sum x_i y_i \\ \vdots \\ \sum x_i^k y_i \end{bmatrix} \Rightarrow$$

$$\begin{bmatrix} 1 & x_1 & \cdots & x_1^k \\ 1 & x_2 & \cdots & x_2^k \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & \cdots & x_n^k \end{bmatrix}^T \begin{bmatrix} 1 & x_1 & \cdots & x_1^k \\ 1 & x_2 & \cdots & x_2^k \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & \cdots & x_n^k \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_k \end{bmatrix} = \begin{bmatrix} 1 & x_1 & \cdots & x_1^k \\ 1 & x_2 & \cdots & x_2^k \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & \cdots & x_n^k \end{bmatrix}^T \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

, from which we see, this is just the normal equation of Least Square Estimation. However, this wouldn't give us too much help in solving numerically. At least, it enhances our intuition and imagination.

2.3.1 Implementation

```python
from numpy import array, matrix, linalg
def DesignMV(xData, yData, k):
    n = len(xData)

    DesignMatrix = [[sum([x ** (i + j) for x in xData]) for i in range
        (k + 1)] \
    for j in range(k + 1)]

    DesignVector = [sum([yData[i] * xData[i] ** j for i in range(n)])
        for j in range(k + 1)]

    return matrix(DesignMatrix), array(DesignVector)


def LeastSquarePoly(xData, yData, k):
    a, b = DesignMV(xData, yData, k)
    c = linalg.solve(a,b)
    return c
```

2.4 Exercise

(1) Inspired by Example 2.2 we could think deeper. Consider below theorem:

Theorem(Weierstrass Approximation Theorem). Let $f : [a, b] \to \mathbb{R}$ be continuous. Given $\epsilon > 0$, there exists a polynomial $p(x)$ satisfying

$$|f(x) - p(x)| < \epsilon$$

for all $x \in [a, b]$.

This theorem tells us that every continuous function could be approximated by polynomials. Then what is the relationship with this chapter?

Lemma: Let $f : [a, b] \to \mathbb{R}$ be continuous. Given $\epsilon > 0$, there exists a polygonal function ϕ satisfying

$$|f(x) - \phi(x)| < \epsilon$$

for all $x \in [a, b]$.

Proof:
We know that a continuous function over a compact set is uniformly continuous over that set. Given $\epsilon > 0$, apply uniform continuity on f to obtain some $\delta > 0$ and partition $[a, b]$ into uniform segments, with each segment length lower than δ. Define $\phi(x)$ at the endpoints of each segment to be equal to f, and to linearly interpolate between segment endpoints.
For any $x \in [a, b]$, let q be the nearest segment endpoint less than or equal to x, and r be the following segment endpoint. (If $x = a$ or $x = b$ then these aren't necessarily defined, but there is still an interval to work with, so there's nothing to worry about.) Since $|x - q| < \delta$ we have that

$$|f(x) - \phi(x)| < \epsilon.$$

We similarly also have $|f(q) - \phi(q)| < \epsilon$. Applying the triangle inequality leaves us with $|f(x) - \phi(x)| < \epsilon$ as desired.
Then you could following the below steps to prove the WAT:

(a) Fix $a \in [-1, 1]$ and sketch

$$h_a(x) = \frac{1}{2}(|x - a| + (x - a))$$

over $[-1, 1]$. Note that h_a is polygonal and satisfies $h_a(x) = 0$ for all $x \in [-1, a]$.

(b) Explain why we know $h_a(x)$ can be uniformly approximated with a polynomial on $[-1, 1]$.

(c) Let ϕ be a polygonal function that is linear on each subinterval of the partition

$$-1 = a_0 < a_1 < a_2 < \cdots < a_n = 1.$$

Show there exist constants $b_0, b_1, \ldots, b_{n-1}$ so that

$$\phi(x) = \phi(-1) + b_0 h_{a_0}(x) + b_1 h_{a_1}(x) + \cdots + b_{n-1} h_{a_{n-1}}(x)$$

for all $x \in [-1, 1]$.

(d) Complete the proof of WAT for the interval $[-1, 1]$, and then generalize to an arbitrary interval $[a, b]$.

3 Numerical Derivative

3.1 Forward & Backward Approximation

This chapter deals with numerical approximations of derivatives. The first question that comes to mind is: why do we need to approximate derivatives at all? After all, we do know how to analytically differentiate every function. Nevertheless, there are several reasons why we still need to approximate derivatives:

1. Even if there exists an underlying function that we need to differentiate, we might know its values only at a sampled data set without knowing the function itself.
2. There are some cases where it may not be obvious that an underlying function exists and all that we have is a discrete data set. We may still be interested in studying changes in the data, which are related, of course, to derivatives.
3. There are times when exact formulas are available, but they are very complicated to the point that an exact computation of the derivative requires a lot of function evaluations. It might be significantly simpler to approximate the derivative instead of computing its exact value.
4. When approximating solutions to ordinary (or partial) differential equations, we typically represent the solution as a discrete approximation that is defined on a grid. Since we then have to evaluate derivatives at the grid points, we need to be able to come up with methods for approximating the derivatives at these points, and again, this will typically be done using only values that are defined on a lattice. The underlying function itself (which in this case is the solution of the equation) is unknown.

A simple approximation of the first derivative is

$$f'(x) \approx \frac{f(x+h)-f(x)}{h}, \quad (3.1)$$

where we assume that $h > 0$. What do we mean when we say that the expression on the right-hand side of (3.1) is an approximation of the derivative? For linear functions, (3.1) is actually an exact expression for the derivative. For almost all other functions, (3.1) is not the exact derivative.
Let's compute the approximation error. We write a Taylor expansion of $f(x+h)$ about x, i.e.,

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(\xi), \quad \xi \in (x, x+h). \quad (3.2)$$

For such an expansion to be valid, we assume that $f(x)$ has two continuous derivatives. The Taylor expansion (3.2) means that we can now replace the approximation (3.1) with an exact formula of the form

$$f'(x) = \frac{f(x+h)-f(x)}{h} - \frac{h}{2}f''(\xi), \quad \xi \in (x, x+h). \quad (3.3)$$

Since this approximation of the derivative at x is based on the values of the function at x and $x+h$, the approximation (3.1) is called a forward differencing or one-sided differencing. The approximation of the derivative at x that is based on the values of the function at $x - h$ and x, i.e.,

$$f'(x) \approx \frac{f(x) - f(x-h)}{h},$$

is called a backward differencing (which is obviously also a one-sided differencing formula). The second term on the right-hand side of (3.3) is the error term. Since the approximation (3.1) can be thought of as being obtained by truncating this term from the exact formula (3.3), this error is called the truncation error. The small parameter h denotes the distance between the two points x and $x+h$. As this distance tends to zero, i.e., $h \to 0$, the two points approach each other and we expect the approximation (3.1) to improve. This is indeed the case if the truncation error goes to zero, which in turn is the case if $f''(\xi)$ is well defined in the interval $(x, x+h)$. The "speed" at which the error goes to zero as $h \to 0$ is called the rate of convergence. When the truncation error is of the order of $O(h)$, we say that the method is a first order method. We refer to a method as a pth-order method if the truncation error is of the order of $O(h^p)$.

Example 3.1: Derive the Taylor series for $\cos(x)$ about x = 0 up to the x^6 term. By keeping only up to the x^4 term, estimate $\cos(0.5)$, where 0.5 is an angle expressed in radian. By comparing with a more accurate answer directly from your calculator, find the percentage error in the approximation.

Solution:
Firstly, we know the Taylor expansion of any function at $x = 0$ is:

$$f(x) = \sum_{n=0}^{\infty} \frac{1}{n!} f^{(n)}(0) x^n$$

Then applying this to $\cos(x)$ at $x = 0$ up to sixth order:

$$\cos(x) = 1 - \frac{1}{2!}x^2 + \frac{1}{4!}x^4 - \frac{1}{6!}x^6 + \mathcal{O}(x^8)$$

And then we leave the last order using the left approximated function for estimation:

$$\widehat{\cos(0.5)} = 1 - \frac{1}{2!}(0.5)^2 + \frac{1}{4!}(0.5)^4 = 0.8776041667$$

We could further compute the relative error:

$$\varepsilon = |\frac{\widehat{\cos(0.5)} - \cos(0.5)}{\cos(0.5)}| \approx 2.461851 \times 10^{-5}$$

, which shows that there are only 0.0025% relative error in the fourth order Taylor approximation.

Example 3.2: Expressions involving derivatives of function $f(x)$ can be evaluated using finite difference approximations.

(a) Starting from Taylor series expansions, derive

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2).$$

Find the constants a and m.

(b) Starting also from Taylor series expansions, derive

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + O(h^n).$$

Find the constants b and n.

(c) Based on the values of m and n, which of the approximations in (a) and (b) is better?

Solution:

(a) Taylor series:

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + \dots$$

Subtracting +ve and -ve versions:

$$\Rightarrow f(x+h) - f(x) = hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + \dots$$

$$\Rightarrow f(x+h) - f(x-h) = hf'(x) + h^2\left(\frac{1}{2}f''(x) + \frac{h}{6}f'''(x)\right)$$

$$\Rightarrow a = 4,\, m = 2$$

(b) +ve version:

$$f(x+h) - f(x) = \frac{f(x+h) - f(x)}{h} + \frac{h^2}{2}f''(x) + O(h^3)$$

-ve version:

$$f(x) - f(x-h) = \frac{f(x+h) - f(x)}{h} + \frac{h^2}{2}f''(x) + O(h^3)$$

Multiplying the two equations:

$$\Rightarrow (f(x+h) - f(x)) - (f(x) - f(x-h)) = \frac{(f(x+h) - f(x))^2}{(h^2)(2)} + \frac{(h^2/3)(f''(x))}{h^2} + \dots = [f'(x)]^2 + O(h^n)$$

$$\Rightarrow b = 1,\, n = 2$$

(c) Since $m = n$, the two methods are equally good.

3.2 Centered Approximation

It is possible to write more accurate formulas than (3.3) for the first derivative. For example, a more accurate approximation for the first derivative that is based on the values of the function at the points $f(x-h)$ and $f(x+h)$ is the centered differencing formula

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}. \quad (3.4)$$

Let's verify that this is indeed a more accurate formula than (3.1). Taylor expansions of the terms on the right-hand side of (3.4) are

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(\xi_1),$$
$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(\xi_2).$$

Here $\xi_1 \in (x, x+h)$ and $\xi_2 \in (x-h, x)$. Hence,

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{h^2}{12}[f'''(\xi_1) + f'''(\xi_2)],$$

which means that the truncation error in the approximation (3.4) is

$$-\frac{h^2}{12}[f'''(\xi_1) + f'''(\xi_2)].$$

If the third-order derivative $f'''(x)$ is a continuous function in the interval $[x-h, x+h]$, then the intermediate value theorem implies that there exists a point $\xi \in (x-h, x+h)$ such that

$$f'''(\xi) = \frac{1}{2}[f'''(\xi_1) + f'''(\xi_2)].$$

Hence,

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{h^2}{6}f'''(\xi), \quad (3.5)$$

which means that the expression (3.4) is a second-order approximation of the first derivative. In a similar way, we can approximate the values of higher-order derivatives. For example, it is easy to verify that the following is a second-order approximation of the second derivative:

$$f''(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}. \quad (3.6)$$

To verify the consistency and the order of approximation of (3.6), we expand

$$f(x \pm h) = f(x) \pm hf'(x) + \frac{h^2}{2}f''(x) \pm \frac{h^3}{6}f'''(x) + \frac{h^4}{24}f^{(4)}(\xi_\pm).$$

Here, $\xi_- \in (x-h, x)$ and $\xi_+ \in (x, x+h)$. Hence,

$$\frac{f(x+h) - 2f(x) + f(x-h)}{h^2} = f''(x) + \frac{h^2}{24}[f^{(4)}(\xi_-) + f^{(4)}(\xi_+)] = f''(x) + \frac{h^2}{12}f^{(4)}(\xi),$$

where we assume that $\xi \in (x-h, x+h)$ and that $f(x)$ has four continuous derivatives in the interval. Hence, the approximation (3.6) is indeed a second-order approximation of the derivative, with a truncation error that is given by

$$-\frac{h^2}{12} f^{(4)}(\xi), \quad \xi \in (x-h, x+h).$$

Example 3.3: A function $y(x)$ follows

$$\frac{dy}{dx} = \cos(x+y)$$

and $y(0) = 0$. Using the Taylor expansion of $y(x)$ at $x = 0$, calculate $y(0.1)$ up to an error $O(x^3)$.

Solution:
Simply, we should find the second derivative of y:

$$\frac{d^2y}{dx^2} = \frac{d}{dx}\frac{dy}{dx} = -\sin(x+y)(1+\frac{dy}{dx})$$

Then, $\frac{d^2y}{dx^2}\Big|_{x=0} = 0$ *(1+0) = 0
By Taylor expansion:
$y(0.1) = y(0) + 0.1 \cdot \frac{dy}{dx} + \frac{0.1^2}{2} \cdot \frac{d^2y}{dx^2} + \mathcal{O}(x^3) = 0.1 + \mathcal{O}(x^3)$
Besides that, using the knowledge that you learned in AMA2111 you may verify that:
We set: $\mathcal{V} = x + y$, then $\frac{d\mathcal{V}}{dx} = 1 + \frac{dy}{dx}$, and we could transfer the above o.d.e:
$\frac{d\mathcal{V}}{dx} = 1 + \cos(\mathcal{V}) \implies \int \frac{1}{\cos(\mathcal{V})+1}\, d\mathcal{V} = \int dx \implies \int \frac{1}{2\cos^2(\mathcal{V})}\, d\mathcal{V} = \int dx$
Until now, the answer seems easy, we change variable by setting $u = \frac{\mathcal{V}}{2}$.
Then $\int \frac{1}{\cos^2(u)}\, du = \int dx \implies \tan(u) = x + C$, which tells us that: $y = 2\tan^{-1}(x) - x$, denote that y(0) = 0.

Example 3.4: A four-point formula for approximating the first derivative of a function $f(x)$ is

$$f'(x) \simeq \frac{-f(x+2\Delta x) + cf(x+\Delta x) - cf(x-\Delta x) + f(x-2\Delta x)}{12\Delta x},$$

where c is a constant.

(a) Find c.

(b) Consider $f(x) = x^4$. Using the above formula, estimate $f'(2)$ with $\Delta x = 1$.

(c) Now consider $f(x) = x^5$. Estimate $f'(2)$ with $\Delta x = 1$ similarly.

(d) Hence, by comparing with two-point formula for the first derivative which has an error of order Δx, state the order of error of the four-point formula. Briefly explain your answer.

Solution:
(a) Clearly, this four points algorithm should be better than two points, since if it is not, there is no need for us to use it.
We could start with the symmetry:

$$f(x+\Delta x) - f(x-\Delta x) = 2\Delta x f'(x) + \frac{1}{3}(\Delta x)^3 f'''(x) + \mathcal{O}((\Delta x)^5)$$

Then same thing for $2\Delta x$:

$$f(x+\Delta x) - f(x-\Delta x) = 4\Delta x f'(x) + \frac{8}{3}(\Delta x)^3 f'''(x) + \mathcal{O}((\Delta x)^5)$$

Since we want to cancel the third derivative of f, we could know c = 8, and then subtract the two equation will give us the answer.
(b) f'(2) $= \frac{-4^4+8\cdot 3^4-8\cdot 1^4+0^4}{12} = \frac{80}{3}$
(c) f'(2) $= \frac{-4^5+8\cdot 3^5-8\cdot 1^5+0^5}{12} = 76$
(d) Combining the two equation we could know:

$$f'(x) = \frac{-f(x+2\Delta x) + cf(x+\Delta x) - cf(x-\Delta x) + f(x-2\Delta x)}{12\Delta x} + \mathcal{O}((\Delta x)^4)$$

3.3 Implementation

```python
def derivative(f, x, h, type="forward"):
    if type == 'forward':
        return (f(x + h) - f(x)) / h
    elif type == 'backward':
        return (f(x) - f(x - h)) / h
    elif type == 'central':
        return (f(x + h) - f(x - h)) / (2 * h)
    else:
        return 'Invalid Type'


def dderivative(f, x, h):
    return (f(x + h) + f(x - h) - 2 * f(x)) / (h ** 2)


from math import sin, cos, pi

x = pi / 4
diffs = {}

for j in range(21):  # range is exclusive so range(21) will stop at 20
    h = 10 ** -j
    deriv = derivative(sin, x, h)
    exact = cos(x)
    diff = abs(deriv - exact)
    diffs[h] = diff

import matplotlib.pyplot as plt
```

```

ordered = sorted(diffs.items())
x, y = zip(*ordered)
plt.loglog(x, y)
plt.show()
```

3.4 Differentiation Via Interpolation

In this section, we demonstrate how to generate differentiation formulas by differentiating an interpolant. The idea is straightforward: the first stage is to construct an interpolating polynomial from the data. An approximation of the derivative at any point can then be obtained by a direct differentiation of the interpolant.
Please note that this part will not be covered in exam probably, just for extension.
We follow this procedure and assume that $f(x_0), \dots, f(x_n)$ are given. The Lagrange form of the interpolation polynomial through these points is

$$Q_n(x) = \sum_{j=0}^{n} f(x_j) l_j(x).$$

Here we simplify the notation and replace $l_n^i(x)$ which is the notation we used in Section ?? by $l_i(x)$. According to the error analysis of Section ??, we know that the interpolation error is

$$f(x) - Q_n(x) = \frac{1}{(n+1)!} f^{(n+1)}(\xi_n) \prod_{j=0}^{n} (x - x_j),$$

where $\xi_n \in (\min(x, x_0, \dots, x_n), \max(x, x_0, \dots, x_n))$. Since here we are assuming that the points $x_0, \dots, x_n$ are fixed, we would like to emphasize the dependence of ξ_n on x and hence replace the ξ_n notation by ξ_x. We have:

$$f(x) = \sum_{j=0}^{n} f(x_j) l_j(x) + \frac{1}{(n+1)!} f^{(n+1)}(\xi_x) w(x), \quad (3.7)$$

where

$$w(x) = \prod_{i=0}^{n} (x - x_i).$$

Differentiating the interpolant (3.7):

$$f'(x) = \sum_{j=0}^{n} f(x_j) l_j'(x) + \frac{1}{(n+1)!} f^{(n+1)}(\xi_x) w'(x) + \frac{1}{(n+1)!} w(x) \frac{d}{dx} f^{(n+1)}(\xi_x). \quad (3.8)$$

We now assume that x is one of the interpolation points, i.e., $x \in \{x_0, \dots, x_n\}$, say x_k, so that

$$f'(x_k) = \sum_{j=0}^{n} f(x_j) l_j'(x_k) + \frac{1}{(n+1)!} f^{(n+1)}(\xi_{x_k}) w'(x_k). \quad (3.9)$$

Now,

$$w'(x) = \sum_{i=0}^{n} \prod_{j=0, j\neq i}^{n} (x - x_j) = \sum_{i=0}^{n} \left[(x - x_0) \cdots (x - x_{i-1})(x - x_{i+1}) \cdots (x - x_n)\right].$$

Hence, when $w'(x)$ is evaluated at an interpolation point x_k, there is only one term in $w'(x)$ that does not vanish, i.e.,

$$w'(x_k) = \prod_{j=0, j\neq k}^{n} (x_k - x_j).$$

The numerical differentiation formula, (3.9), then becomes

$$f'(x_k) = \sum_{j=0}^{n} f(x_j) l_j'(x_k) + \frac{1}{(n+1)!} f^{(n+1)}(\xi_{x_k}) \prod_{j=0, j\neq k}^{n} (x_k - x_j). \quad (3.10)$$

We refer to the formula (3.10) as a differentiation by interpolation algorithm.
We demonstrate how to use the differentiation by interpolation formula (3.10) in the case where $n = 1$ and $k = 0$. This means that we use two interpolation points $(x_0, f(x_0))$ and $(x_1, f(x_1))$ and want to approximate $f'(x_0)$. The Lagrange interpolation polynomial in this case is

$$Q_1(x) = f(x_0) l_0(x) + f(x_1) l_1(x),$$

where

$$l_0(x) = \frac{x - x_1}{x_0 - x_1}, \quad l_1(x) = \frac{x - x_0}{x_1 - x_0}.$$

Hence,

$$l_0'(x) = \frac{1}{x_0 - x_1}, \quad l_1'(x) = \frac{1}{x_1 - x_0}.$$

We thus have

$$Q_1'(x_0) = f(x_0) \cdot \frac{1}{x_0 - x_1} + f(x_1) \cdot \frac{1}{x_1 - x_0} + \frac{1}{2} f''(\xi)(x_0 - x_1).$$

Here, we simplify the notation and assume that $\xi \in (x_0, x_1)$. If we now let $x_1 = x_0 + h$, then

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0)}{h} - \frac{h}{2} f''(\xi),$$

which is the (first-order) forward differencing approximation of $f'(x_0)$, (3.3).
We repeat the previous example in the case $n = 2$ and $k = 0$. This time

$$Q_2(x) = f(x_0) l_0(x) + f(x_1) l_1(x) + f(x_2) l_2(x),$$

with

$$l_0(x) = \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)}, \quad l_1(x) = \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)}, \quad l_2(x) = \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}.$$

Hence,

$$l_0'(x) = \frac{2x-x_1-x_2}{(x_0-x_1)(x_0-x_2)}, \quad l_1'(x) = \frac{2x-x_0-x_2}{(x_1-x_0)(x_1-x_2)}, \quad l_2'(x) = \frac{2x-x_0-x_1}{(x_2-x_0)(x_2-x_1)}.$$

Evaluating $l_j'(x)$ for $j = 0, 1, 2$ at x_0 we have

$$l_0'(x_0) = \frac{2x_0-x_1-x_2}{(x_0-x_1)(x_0-x_2)}, \quad l_1'(x_0) = \frac{x_0-x_2}{(x_1-x_0)(x_1-x_2)}, \quad l_2'(x_0) = \frac{x_0-x_1}{(x_2-x_0)(x_2-x_1)}.$$

Hence,

$$\begin{aligned} Q_2'(x_0) = f(x_0)\cdot\frac{2x_0-x_1-x_2}{(x_0-x_1)(x_0-x_2)} + f(x_1)\cdot\frac{x_0-x_2}{(x_1-x_0)(x_1-x_2)} + \\ f(x_2)\cdot\frac{x_0-x_1}{(x_2-x_0)(x_2-x_1)} + \frac{1}{6}f^{(3)}(\xi)(x_0-x_1)(x_0-x_2). \end{aligned}$$

Here, we assume $\xi \in (x_0, x_2)$. For $x_i = x + ih$, $i = 0, 1, 2$, equation becomes

$$Q_2'(x) = \frac{-3f(x)+4f(x+h)-f(x+2h)}{2h} + \frac{f^{(3)}(\xi)}{6}h^2,$$

which is a one-sided, second-order approximation of the first derivative.
Remark. In a similar way, if we were to repeat the last example with $n = 2$ while approximating the derivative at x_1, the resulting formula would be the second-order centered approximation of the first derivative

$$f'(x) = \frac{f(x+h)-f(x-h)}{2h} - \frac{1}{6}f^{(3)}(\xi)h^2.$$

3.5 Exercise

Richardson's extrapolation can be viewed as a general procedure for improving the accuracy of approximations when the structure of the error is known. While we study it here in the context of numerical differentiation, it is by no means limited only to differentiation, and we will get back to it later on when we study methods for numerical integration.
We start with an example in which we show how to turn a second-order approximation of the first derivative into a fourth-order approximation of the same quantity. We already know that we can write a second-order approximation of $f'(x)$ given its values in $f(x \pm h)$. In order to improve this approximation, we will need some more insight on the internal structure of the error. We therefore start with the Taylor expansions of $f(x \pm h)$ about the point x, i.e.,

$$f(x+h) = \sum_{k=0}^{\infty} \frac{f^{(k)}(x)}{k!}h^k,$$

$$f(x-h) = \sum_{k=0}^{\infty} \frac{(-1)^k f^{(k)}(x)}{k!} h^k.$$

Hence,

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \left(\frac{h^2}{3!} f^{(3)}(x) + \frac{h^4}{5!} f^{(5)}(x) + \ldots \right). \quad (1)$$

We rewrite (1) as:

$$L = D(h) + e_2 h^2 + e_4 h^4 + \ldots, \quad (2)$$

where L denotes the quantity that we are interested in approximating, i.e.,

$$L = f'(x),$$

and $D(h)$ is the approximation, which in this case is

$$D(h) = \frac{f(x+h) - f(x-h)}{2h}.$$

The error is:

$$E = e_2 h^2 + e_4 h^4 + \ldots,$$

where e_i denotes the coefficient of h^i in (1). The important property of the coefficients e_i is that they do not depend on h. We note that the formula:

$$L \approx D(h),$$

is a second-order approximation of the first derivative which is based on the values of $f(x)$ at $x \pm h$. We assume here that in general $e_i \neq 0$. In order to improve the approximation of L, our strategy will be to eliminate the term $e_2 h^2$ from the error. How can this be done? One possibility is to write another approximation that is based on the values of the function at different points. For example, we can write:

$$L = D(2h) + e_2 (2h)^2 + e_4 (2h)^4 + \ldots. \quad (3)$$

This, of course, is still a second-order approximation of the derivative. However, the idea is to combine (2) with (3) such that the h^2 term in the error vanishes. Indeed, subtracting the following equations from each other:

$$4L = 4D(h) + 4e_2 h^2 + 4e_4 h^4 + \ldots,$$

$$L = D(2h) + 4e_2 h^2 + 16 e_4 h^4 + \ldots,$$

we have:

$$L = \frac{4D(h) - D(2h)}{3} - 4e_4 h^4 + \ldots.$$

In other words, a fourth-order approximation of the derivative is:

$$f'(x) = \frac{-f(x+2h) + 8f(x+h) - 8f(x-h) + f(x-2h)}{12h} + O(h^4). \quad (4)$$

Note that (4) improves the accuracy of the approximation (1) by using more points.

This process can be repeated over and over as long as the structure of the error is known. For example, we can write (4) as:

$$L = S(h) + a_4 h^4 + a_6 h^6 + \dots, \quad (5)$$

where

$$S(h) = \frac{-f(x+2h) + 8f(x+h) - 8f(x-h) + f(x-2h)}{12h}.$$

(1) Hence, according to the above could you give a sixth-order approximation of the derivative by eliminating the term $a_4 h^4$?

Final Remark Once again we emphasize that Richardson's extrapolation is a general procedure for improving the accuracy of numerical approximations that can be used when the structure of the error is known. It is not specific to numerical differentiation.

4 Numerical ODE

4.1 Recall ODE

A differential equation expresses a set of constraints among the derivatives of an unknown function. Here's a simple example:

$$\frac{d}{dt}x(t) = ax(t)$$

This means that the derivative of the unknown function is equal to a times the unknown function itself. For compactness, the independent variable (t, here) is often omitted, and the notation $\frac{dx}{dt}$ is often abbreviated with a dot, like so: $\dot{x}$, or with a prime, like so: x'. Using these two notations, equation (1) becomes $\dot{x} = ax$ or $x' = ax$, respectively.
To solve a differential equation, you (generally) can't just integrate both sides. Rather, you have to find some function that satisfies the constraints expressed in that equation — in the example above, some function $x(t)$ whose derivative $x'(t)$ is equal to a constant multiple of the function itself. There is no general, foolproof way to do this unless the equation is very simple. Differential equations textbooks provide lots of suggestions about approaches, but there are many differential equations (DEs) that simply don't have analytic solutions — that is, solutions that you can write down. These equations can only be solved numerically, using the kinds of methods that are described in these notes.
Differential equations are interesting and useful to scientists and engineers because they "model" the physical world: that is, they capture the physics of a system, and their solutions emulate the behavior of that system. The class of differential equations that have no analytic solutions has a highly specific and interesting analog in the physical world: they model so-called chaotic systems. This means that numerical methods for solving ODEs are an essential tool for anyone (like me) who studies chaos.
An *ordinary differential equation* (ODE) has only one independent variable, and all derivatives in it are taken with respect to that variable. Most often, this variable is time t, but some books use x as the independent variable; pay careful attention to what the derivative is taken with respect to so you don't get confused.
Here's an example of where ODEs come from. Consider a mass m on a spring with spring constant k.
If you define x as the position of the end of the spring, as shown in the figure, and set $x = 0$ at the point where the end of the spring would naturally rest if it were unloaded (i.e., if $m = 0$), you can write the following force balance at the mass:

$$\Sigma F = ma$$

$$mg - kx = ma$$

Acceleration a is the second derivative of position: $a = x''$, so the equation becomes:

$$mg - kx = mx''.$$

The mg force is gravity; kx is Hooke's law for the force exerted by a spring. The signs of mg and kx are opposite because gravity pulls in the direction of positive x, and the spring pulls in the direction of negative x. Gathering terms and dividing through by m gives you the following ODE for the spring-mass system:

$$x''(t) + \frac{k}{m}x(t) - g = 0$$

Note that this ODE model does not include the effects of friction; if friction plays an important role in the system, this model is inadequate.

The independent variable t only appears implicitly in this equation — that is, hidden in the time dependence of the function $x(t)$. This means that the physics (the governing equations) of the system is not a function of time. If, on the other hand, someone were holding the top of the spring and moving it up and down in a sinusoidal pattern, the apparent gravity acting on the mass would change sinusoidally (much as the gravity changes in an elevator that is starting or stopping), and the equation would look like this:

$$x''(t) + \frac{k}{m}x(t) - g_0 \sin t = 0$$

The variable t appears explicitly in this ODE — by itself, and not wrapped in the brackets of a function like $x(t)$. Systems where this occurs are called *nonautonomous.*

The *order* of an ODE is the degree of the highest derivative in that equation. $x' = ax$ is a first-order ODE, $x''' - \tan x' = 2$ is a third-order ODE, and the spring-mass equation above is second order. An n-th order ODE can be transformed into n first-order ODEs (an n-th-order "ODE system"), and vice versa.

Example 4.1: The transformation requires the introduction of "helper" variables. For example, given:

$$x''' + 36 \log x' - x^2 + \sin 2t = 14,$$

Transform it into separate first order ODEs.

Solution:

1. Rewrite the ODE with the highest-order term by itself on the left-hand side:

$$x''' = 14 + x^2 - 36 \log x' - \sin 2t$$

2. Define $n - 1$ helper variables:

$$x_1' = x_2, \quad x_2' = x_3$$

3. Rewrite the equation using the helper variables, with no derivative signs on the right-hand side and only one on the left-hand side:

$$x_3' = 14 + x_1^2 - 36 \log x_2 - \sin 2t$$

4. Append the equations that define the helper variables:

$$x_1' = x_2, \quad x_2' = x_3, \quad x_3' = 14 + x_1^2 - 36 \log x_2 - \sin 2t$$

This set of first-order ODEs is equivalent to the original third-order ODE, as you can see by substituting the first two equations into the third.

4.2 Euler's Method

In this section, we derive and analyze the simplest numerical method for ODEs, the (forward) Euler's method. Through analyzing Euler's method, we introduce general concepts that are useful for understanding and analyzing all kinds of numerical methods.

Summary of Notation The following notation will be used throughout this section:

- $t_0, \dots, t_N$: the points where the approximate solution is defined.
- $y_j = y(t_j)$: the exact solution to the IVP at t_j.
- $\tilde{y}_j$: the approximation at t_j.
- τ_j: the truncation error in obtaining $\tilde{y}_j$.
- h or Δt: the "step size", defined as $t_j - t_{j-1}$ (if constant); otherwise h_j or Δt_j.

We seek a numerical solution $y(t)$ to the scalar IVP:

$$y' = f(t, y), \quad y(a) = y_0$$

(with $y \in \mathbb{R}$) up to a time $t = b$. The approximation will take the form of values $\tilde{y}_j$ defined on a grid:

$$a = t_0 < t_1 < \cdots < t_N = b,$$

such that:

$$\tilde{y}_j \approx y(t_j).$$

For convenience, denote by y_j the exact solution at t_j and let the "error" at each point be:

$$e_j = y_j - \tilde{y}_j.$$

It will be assumed that we have a free choice of the t_j's. This situation is illustrated in Figure 1.

4.2.1 Forward Euler's Method

Assume, for simplicity, that the step size (for time):

$$h = t_j - t_{j-1},$$

is constant. This is not necessary but is convenient.
Suppose that we have the exact value of $y(t)$. To get $y(t+h)$ from $y(t)$, expand $y(t+h)$ in a Taylor series and use the ODE to simplify the derivatives:

$$y(t+h) = y(t) + hy'(t) + O(h^2),$$
$$= y(t) + hf(t, y) + O(h^2).$$

This gives, for any point t_j, the formula:

$$y(t_j + h) = y(t_j) + hf(t_j, y(t_j)) + \tau_{j+1},$$

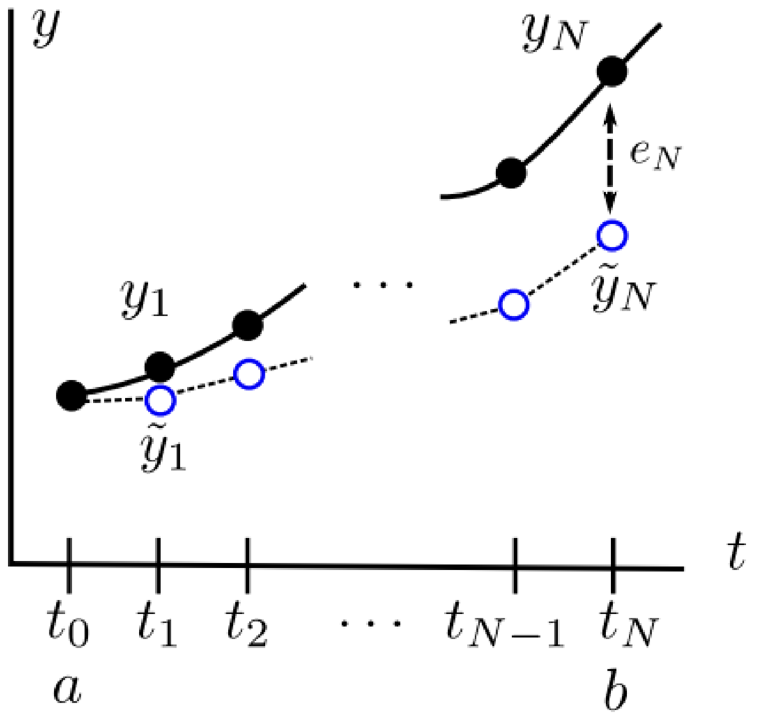


Figure 1: Numerical solution of an IVP forward in time from $t = a$ to $t = b$. The actual values are $y_1, \ldots, y_N$ and the estimated values are $\tilde{y}_1, \ldots, \tilde{y}_N$.

where τ_{j+1} is the *local truncation error* defined below. The important thing is that:

$$\tau_{j+1} = O(h^2).$$

Dropping the error term and iterating this formula, we derive the (forward) Euler's method:

$$\tilde{y}_{j+1} = \tilde{y}_j + hf(t_j, \tilde{y}_j).$$

Algorithm (Forward Euler's Method): The forward Euler's method for solving the IVP:

$$y' = f(t, y), \quad y(a) = y_0,$$

is given by:

$$\tilde{y}_{j+1} = \tilde{y}_j + hf(t_j, \tilde{y}_j).$$

The initial point is, ideally:

$$\tilde{y}_0 = y_0,$$

since the initial value (at t_0) is given. However, in practice, there may be some initial error in this quantity as well.

4.2.2 Local Truncation Error (LTE)

Definition (Local Truncation Error): The *local truncation error* τ_{j+1} is the error incurred in obtaining $\tilde{y}_{j+1}$ when the previous data $y_j = \tilde{y}_j$ is known exactly:

$$y_{j+1} = \tilde{y}_{j+1} + \tau_{j+1}.$$

In other words, it is the amount by which the exact solution fails to satisfy the equation given by the numeric method.
The LTE is "local" in the sense that it does not include errors created at previous steps.
For example, for Euler's method:

$$\tilde{y}_{j+1} = \tilde{y}_j + hf(t_j, \tilde{y}_j).$$

The LTE is the error between this and the true value of $\tilde{y}_{j+1}$ when $\tilde{y}_j = y_j$:

$$y_{j+1} = \tilde{y}_j + hf(t_j, \tilde{y}_j) + \tau_{j+1},$$

$$= y_j + hf(t_j, y_j) + \tau_{j+1}.$$

Notice that the total error is not just the sum of the truncation errors, because when starting the numeric method with y_0, at step j, f is evaluated at the approximation $\tilde{y}_j$, and $\tilde{y}_j \neq y_j$. The truncation error propagates through the iteration, as a careful analysis will show.

4.3 Implementation & Example

Example 4.2: A particle of mass $M = 1\,\mathrm{g}$ and total energy $E = 1.001\,\mathrm{J}$ moving in the $+x$ direction has a potential energy $V(x) = c/x$ at position x, where $c = 0.1\,\mathrm{Jm}$. The velocity x' is given by

$$x'(t) = \sqrt{\frac{2(E - V(x(t)))}{M}}$$

At $t = 0$, $x = 0.1\,\mathrm{m}$. Find $x(t)$ for $0 \leq t \leq 0.01\,\mathrm{s}$.

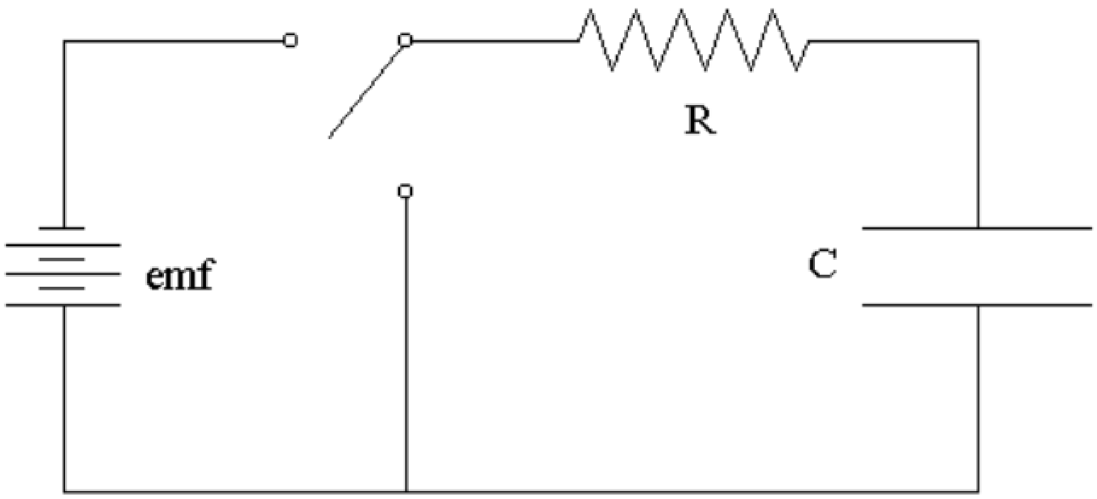
Solution: Putting

$$f(t, x) = \sqrt{\frac{2(E - V(x))}{M}}$$

```python
# Euler method for particle in a 1D potential
from math import *
import numpy as np

M = 1e-3       # mass in kg
E = 1.001      # total energy in J
c = 0.1        # potential energy parameter in Jm
T = 1e-2       # final time in s
dt= 0.001      # time step of Euler method
```

```
x = 0.1        # position in m

def f(t, x):
    V = c/x                          # 1D potential
    return sqrt( 2 * (E - V) /M )  # velocity

for t in np.arange(0, T, dt):
    print(t, x)             # output t, x(t)
    x = x + dt * f(t, x)     # Euler method iteration
```

Example 4.3: A resistor-capacitor (RC) circuit consists of a battery of voltage V_b, a resistor of resistance R, and a capacitor of capacitance C all connected in series. Using a switch, the battery can be disconnected from the circuit.
(a) Write down the evolution equations in the form of a first-order differential equations for the charge Q(t) in the capacitor for both cases (i) with and (ii) without the battery connected.
(b) Let V_b = 9V, R = 10^5 Ω, and C = 10^{-4}F. Initially at time t = 0, the capacitor is fully discharged (i.e. Q(0) = 0) and the battery is connected. At time t = 20s, the battery is disconnected. Write a computer program which applies Euler's method to calculate Q(t) for 0 ¡ t ¡ 40s. Use a time step Δt = 1s. Re-run your program with Δt = 0.1s. Plot Q(t) as a function of t for both values of Δt on the same graph. Submit the source listing of one of your programs.



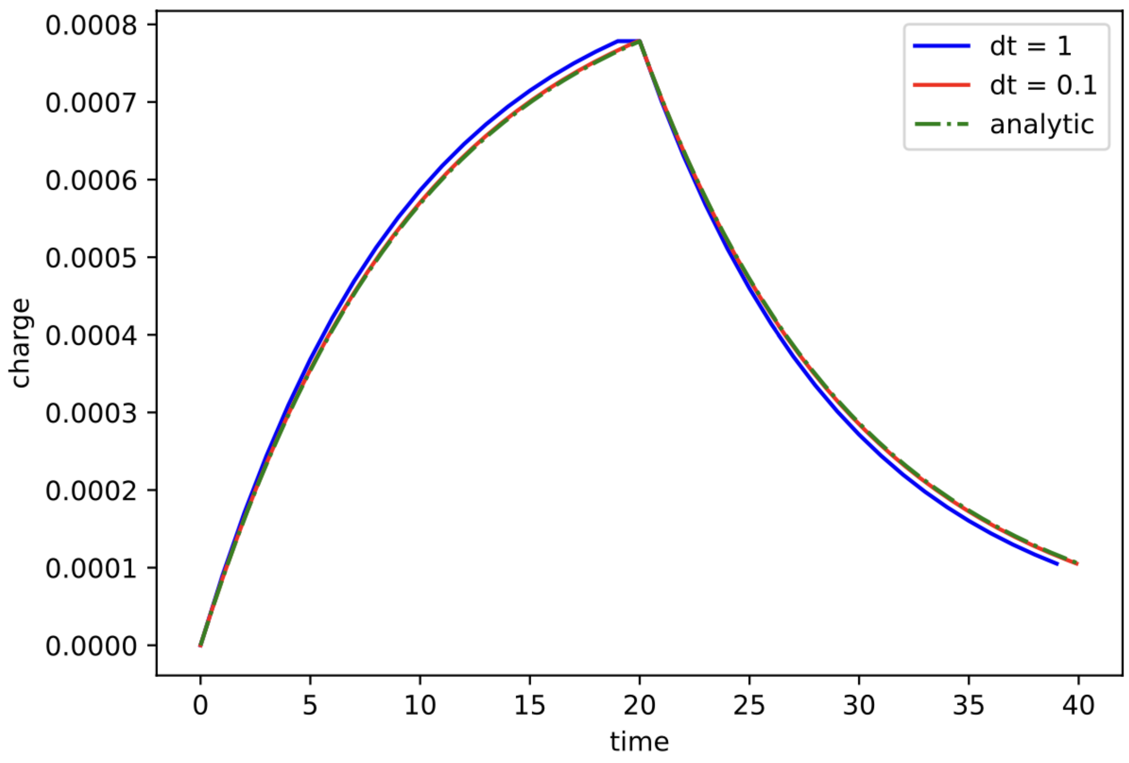
Solution:
(a) (i) Charging period: $V_b = V_R + V_C = Ri_C + \frac{Q(t)}{C} = R\frac{dQ(t)}{dt} + \frac{Q(t)}{C}$, which gives us the following differential equation: $\frac{dQ(t)}{dt} = \frac{V_b}{R} - \frac{Q(t)}{RC}$.
(ii) Discharging period: $V_C = \frac{Q(t)}{C} = V_R = -R\frac{dQ(t)}{dt}$, then $\frac{dQ(t)}{dt} = -\frac{Q(t)}{RC}$.
By simple algebra, we could solve the differential equation:

$$Q_1 = CV_b(1 - e^{\frac{-t}{RC}}), \quad t \in [0, 20]$$
$$Q_2 = Q_1(20) \cdot e^{\frac{-t}{RC}}, \quad t \in [20, 40]$$

(b)

```python
def EulerFirstOrder(df, y0, dt, interval):
    a, b = interval
    x, y = a, y0
    xRange, yChange = [], []
    while x < b:
        xRange.append(x)
        yChange.append(y)
        y += dt * df(y)
        x += dt
    return xRange, yChange


def f1(y, Vb=9, R=1e5, C=1e-4):
    return Vb / R - y / (R * C)


def f2(y, R=1e5, C=1e-4):
    return -y / (R * C)


dt1 = 1
dt2 = 0.1
itv1 = (0, 20)
itv2 = (20, 40)

x1, y1 = EulerFirstOrder(f1, 0, dt1, itv1)
x2, y2 = EulerFirstOrder(f2, y1[-1], dt1, itv2)
xList1 = x1 + x2
yList1 = y1 + y2

x3, y3 = EulerFirstOrder(f1, 0, dt2, itv1)
x4, y4 = EulerFirstOrder(f2, y3[-1], dt2, itv2)

xList2 = x3 + x4
yList2 = y3 + y4

from math import exp


def analytic1(t, Vb=9, R=1e5, C=1e-4):
    return C * Vb * (1 - exp(-t / (R * C)))


def analytic2(t, R=1e5, C=1e-4):
    Q_0 = analytic1(20) * exp(20 / (R * C))
    return Q_0 * exp(-t / (R * C))
```

```python


xReal = [i / 100 for i in range(4001)]
yReal = [analytic1(i) if i <= 20 else analytic2(i) for i in xReal]

import matplotlib.pyplot as plt

plt.plot(xList1, yList1, color='blue')
plt.plot(xList2, yList2, color='red')
plt.plot(xReal, yReal, color='green', linestyle='-.')
plt.xlabel("time")
plt.ylabel("charge")
plt.legend(("dt␣=␣1", "dt␣=␣0.1", "analytic"), loc="upper␣right")
plt.show()
```



Example 4.4: Forced and damped harmonic oscillator, now, assume that $M = 0.1\,\text{kg}$, $k = 10\,\text{N/m}$, $\lambda = 0.7\,\text{Ns/m}$, and $F(t) = 0$. Initially, $x = 0$ and $v = 0$. Find $x(t)$.

Solution: The equation of motion can be rewritten as:

$$x'' = f(t, x, v) = -\frac{k}{M}x + g - \frac{\lambda}{M}v$$

```python
# Euler method for damped oscillator
import numpy as np

```

```
M = 0.1       # mass
k = 10.       # spring constant
g = 9.8       # gravity
Lambda = 0.7 # damping coefficient
T = 1.        # final time
dt= 0.01      # time step of Euler method

x = 0.        # position  m
v = 0.        # velocity in m/s
eps=1e-12     # tolerance for machine error

def f(t, x, v):
    return -k/M*x +g - Lambda/M*v   # particle acceleration

for t in np.arange(0, T+eps, dt):
    print( t, x)              # output t, x(t)
    dx = dt * v               # Euler method iteration
    dv = dt * f(t, x, v)
    x += dx
    v += dv
```

4.4 Exercise

(1) Try to prove the below theorem:

Theorem (Convergence of Euler's method). Suppose:

1. The actual solution $y(t)$ satisfies $\max_{[a,b]} |y''| \le M$.
2. $f(t, y)$ is L-Lipschitz in y.

Let $\tilde{y}_j$ be the result of applying Euler's method starting at $(t_0 = a, \tilde{y}_0 = y_0)$. Then

$$|e_j| \le \frac{Mh}{2L}\left[e^{L(t_j - t_0)} - 1\right]$$

$$\max_{0 \le j \le n} |e_j| \le \frac{Mh}{2L}\left[e^{L(b-a)} - 1\right].$$

where $e_j = y_j - \tilde{y}_j$ is the error at t_j.

Backward Euler's method

The forward Euler's method is an *explicit* method: the approximation $\tilde{y}_{j+1}$ can be evaluated directly in terms of $\tilde{y}_j$ and other known quantities. In contrast, an *implicit* method is one where we are only given a (nonlinear) equation that is satisfied by $\tilde{y}_{j+1}$, and we have to solve for $\tilde{y}_{j+1}$.

The simplest example of an implicit method is backward Euler, which has the iteration

$$\tilde{y}_{j+1} = \tilde{y}_j + hf(t_{j+1}, \tilde{y}_{j+1}).$$

Note the function is evaluated at $\tilde{y}_{j+1}$ rather than $\tilde{y}_j$ as in the forward Euler's method. To solve for $\tilde{y}_{j+1}$, we can iterate Newton's method until convergence; $\tilde{y}_j$ makes for a good initial guess.
Using a "backward" Taylor expansion around t_{j+1} rather than around t_j, you can prove that the LTE is also $O(h^2)$.
This seems more complicated; why would you want to use the backward method? It turns out that it has better stability properties; we will come back to this point.

Algorithm (Backward Euler's method): The backward Euler's method is given by

$$\tilde{y}_{j+1} = \tilde{y}_j + hf(t_{j+1}, \tilde{y}_{j+1}).$$

Here, you can solve for $\tilde{y}_{j+1}$ using Newton's method with $\tilde{y}_j$ as an initial guess.

If Newton's method is used, the code must know f and $\partial f / \partial y$. At step j, we would like to set $\tilde{y}_{j+1}$ equal to the zero of

$$g(z) = z - \tilde{y}_j - hf(t_{j+1}, z).$$

We compute

$$g'(z) = 1 - hf_y(t_{j+1}, z)$$

so the Newton iteration is

$$z_{k+1} = z_k - \frac{g(z_k)}{g'(z_k)}.$$

This is iterated until convergence; then $\tilde{y}_{j+1}$ is set to the resulting z.
(2) Prove an analogue of Theorem in (1) for the backward Euler's method. You can assume that the numeric solution $\tilde{y}_{j+1}$ obtained from algorithm is exact.
Final Remark: If you are interested into chapter 1-4, you are recommended to take AMA3201 for more details.

5 Basic Statistics

5.1 Expectation & Variance

Firstly, this is not, for sure, a statistical inference course, and for our physicist, we only care about certain simple property of a random variable. Just expectation and variance. Therefore, we introduce this quantity briefly.
Let us first consider the expectation with discrete r.v. For expectation our first impression is just adding all items up then divided by its total number. However, this is not entirely correct in general, since, for most of the cases, the occurrence of samples is not uniform. For adding things up, then divided by its total number is assumed that all samples are equally likely to appear. i.e.

$$\widehat{\mathbb{E}(X)} = \frac{1}{n}\sum_{i=1}^{n} X_i = \mathbb{E}(X) \;\Leftrightarrow\; X_i \sim \mathcal{U}(\cdot,\cdot)$$

,where we assume X_i are identically and independent distributed (i.i.d).
Then clearly for the more general case, each X_i has its own probability to be a certain value, which is called the probability mass function. i.e.

$$P(X = X_i) = \alpha_i \;\Rightarrow\; \mathbb{E}(X) = \sum_{i=1}^{n} X_i P(X = X_i)$$

, where we should denote that for all sample spaces: $\mathcal{E} = \{X_1, \ldots, X_t\}$, the total probability measure should sum up to one. i.e. $\sum\limits_{y\in\mathcal{E}} P(X = y) = 1$. Then for this certain set of p.m.f., we say $X \sim f$, where f is just some probability mass function.
Remark: We could also do that in this way:

$$\begin{aligned}\mathbb{E}(X) &= \sum_{a\in\mathbb{E}} a\mathbb{P}(X = a) = \sum_{a\in\mathbb{E}} \int_0^a dt\, \mathbb{P}(X = a) \\ &= \int_0^a dt \sum_{a>t} \mathbb{P}(X = a) = \int_0^\infty \mathbb{P}(X > t)\, dt\end{aligned}$$

Example 6.1: If x is a positive r.v. follows certain p.m.f g, and we know that $g \propto e^{-\lambda x}$, what is the distribution function?

Solution:
Since $X \sim g$, where $g \propto e^{-\lambda x}$, then $\exists\, k \in \mathbb{R}$, s.t. $g = ke^{-\lambda x}$. And we know that the probability measure over whole space should be one. Then:

$$\int_0^\infty ke^{-\lambda x}dx = -\frac{k}{\lambda}e^{-\lambda x}\Big|_0^\infty = \frac{k}{\lambda} = 1$$

, which tells us that $g(x) = \lambda e^{-\lambda x}$. This, we will know later, is an exponential distribution.
Then we could consider the variance in discrete version; simply put, variance is the deviation of our r.v. from its mean/expectation, but in order to eliminate the canceling between positive

and negative difference, we tend to square it. To put that into formally mathematical language is as follows:

$$\text{Var}(X) = \mathbb{E}((X - \mathbb{E}(X))^2) = \mathbb{E}(X^2) - (\mathbb{E}(X))^2.$$

, here we used the linearity of expectation, that is, $\mathbb{E}(aX + bY) = a\mathbb{E}(X) + b\mathbb{E}(Y)$, where you should verify.

Then if a batch of rv is i.i.d. with a certain p.m.f., then $\text{Var}(\sum_{i=1}^{n} X_i) = \sum_{i=1}^{n} \text{Var}(X_i)$.

Then those quantity introduced above could be easily extended in a continuous way, but the pmf becomes a probability density function (pdf), and replace $\sum$ with $\int$.

Example 6.2: For the case shown in example 6.1, what is the expectation and variance of it?

Solution:

$$\mathbb{E}(X) = \int_0^\infty \lambda x e^{-\lambda x} dx = \frac{1}{\lambda}$$
$$\text{Var}(X) = \int_0^\infty \lambda x^2 e^{-\lambda x} dx - \frac{1}{\lambda^2} = \frac{1}{\lambda}$$

Last measurement is covariance, which tells us the **linear relationship** of two r.v., given below:

$$\text{Cov}(X, Y) = \mathbb{E}((X - \mathbb{E}(X))(Y - \mathbb{E}(Y))) = \mathbb{E}(XY) - \mathbb{E}(X)\mathbb{E}(Y).$$

Then for more general case of variance of sum of r.v. could be extended from: $\text{Var}(X+Y) = \text{Var}(X) + \text{Var}(Y) + 2\text{Cov}(X, Y)$.

5.2 Weak Law of Large Numbers

However, you may wonder why we know formally we can't just add them up, we still want to do that? Since by (weak) Law of Large Numbers (L.L.N), it tells us that:

Weak Law of Large Numbers: Let $X_1, X_2, \cdots, X_n$ be n i.i.d r.v. with mean μ. Let $\overline{X_n} = \frac{1}{n}\sum_{i=1}^{n} X_i$ be the sample mean. Then, $\overline{X_n}$ converges in probability to μ. i.e. $\forall \varepsilon > 0$, $\lim_{n\to\infty} \mathbb{P}[|\overline{X_n} - \mu| > \varepsilon] = 0$.

Before delving into the LLN, let's build some tools for us for convenience, however, fortunately, we only need to prove one inequality.

Chebyshev's Inequality: Let X be a random variable and $\varepsilon \in \mathbb{R}^+$, and with its $\mathbb{E}[X] = \mu$, Var(X) $= \sigma^2$, then we have: $\mathbb{P}[|X - \mu| \geq \varepsilon] \leq \frac{\sigma^2}{\varepsilon^2}$.

Proof: For X be continuous r.v.: $\sigma^2 = \int\limits_{-\infty}^{+\infty} (x-\mu)^2 f_X(x)dx \geq \int\limits_{|x-\mu|\geq\varepsilon} (x-\mu)^2 f_X(x)dx \geq \varepsilon^2 \int\limits_{|x-\mu|\geq\varepsilon} f_X(x)dx = \varepsilon^2\mathbb{P}[|X-\mu|\geq\varepsilon]$, then it is done.

Proof for weak LLN: By the property of the expectation and variance, there is no need to prove in detail that $\mathbb{E}[\overline{X_n}] = \mu$, $\text{Var}(\overline{X_n}) = \frac{\sigma^2}{n}$. Then using chebyshev's inequality, we have: $\lim\limits_{n\to\infty} \mathbb{P}[|\overline{X_n}-\mu| > \varepsilon] \leq \lim\limits_{n\to\infty} \frac{Var(\overline{X_n})}{\varepsilon^2} = \lim\limits_{n\to\infty} \frac{\sigma^2}{n\varepsilon^2} = 0$, then it is done.

Quick Remark: When we finish the little proof of weak LLN, that is $\overline{X_n} = \frac{1}{n}\sum\limits_{i=1}^{n} X_i \xrightarrow[n\to\infty]{\mathbb{P}} \mathbb{E}[X]$, we finally know that why many textbook will take arithmetic average to replace the expectation, although it is not rigorous. Or even a more interesting version: Statistics is replace expectation with average.

5.3 Program Estimation

We used one pass algorithm in order to avoid too much memory usage.

```python
import random

# set up the seed first for reuse
random.seed(33)


# get the rand sample we may change this by its p.d.f
def rand():
    return random.uniform(0, 1)


# define the related quantity
Size = 1000
sumX = 0
sumX2 = 0
cnt = 0

# Start to find mean/variance
while cnt < Size:
    sample = rand()
    sumX += sample
    sumX2 += sample ** 2
    cnt += 1

meanX = sumX / cnt
varX = sumX2 / cnt - meanX ** 2
```

Example 6.3: In the early 1600s, Galileo was asked to explain the fact that, although the number of triples of integers from 1 to 6 with sum 9 is the same as the number of such triples with sum 10, when three dice are rolled, a 9 seemed to come up less often than a 10 —supposedly in the experience of gamblers.** (a) Calculate the exact probabilities of getting respectively 9 and 10 when three dice are rolled. Can you conclude from your results that the gamblers were correct? (b) Write a program to simulate the roll of three dice a large number of times and keep track of the proportion of times that the sum is 9 and the proportion of times it is 10. Do your simulations support the results in part (a)?

Solution:
(a)

```python
# Calculate the theoretical result

Total = 0
CaseNine = 0
CaseTen = 0

for i in range(1, 7):
    for j in range(1, 7):
        for k in range(1, 7):
            Total += 1
            Sum = i + j + k
            if Sum == 9:
                CaseNine += 1
            elif Sum == 10:
                CaseTen += 1
```

For exact derivation: we denote the event A as getting an nine in one triplets of rolling, and B as getting a ten in one triplets of rolling. Ω as the number of cases of one triplets of rolling, equals to 6^3 apparently. Then, since:

$$9 = 1+2+6 = 1+3+5 = 1+4+4 = 2+2+5 = 2+3+4 = 3+3+3$$
$$10 = 1+3+6 = 1+4+5 = 2+2+6 = 2+3+5 = 2+4+4 = 3+3+4$$

Then:

$$\#A = 2\binom{3}{1} + 3\binom{3}{1}\binom{2}{1} + 1 = 25$$
$$\#B = 3\binom{3}{1} + 3\binom{3}{1}\binom{2}{1} = 27$$
$$P(A) = \frac{\#A}{\Omega} = \frac{25}{216} \approx 0.11574074074$$
$$P(B) = \frac{\#B}{\Omega} = \frac{27}{216} = 0.125$$

Gambler is correct, the real probability of getting ten in triplets is slightly higher than getting nine.

(b)

```python
from random import *


def RollDice():
    return randint(1, 6)


# Calculate the experimental result

NineCount = 0
TenCount = 0
time = int(1e5)

for t in range(time):
    x1, x2, x3 = RollDice(), RollDice(), RollDice()
    Sum = x1 + x2 + x3
    if Sum == 9:
        NineCount += 1
    elif Sum == 10:
        TenCount += 1
```

The result that program mimics agree quite well with theoretical one.

5.4 Exercise

(1) If $x_1, x_2, \cdots, x_n \sim f$, and are independent with each other, it is known that $\mathbb{E}(x_1) = \mu, \mathrm{Var}(x_1) = \sigma^2$, let's define: $\overline{X} = \frac{1}{n}\sum x_i$, then what is the unbiased estimation of the expectation and variance of $\overline{X}$.

(2) (a) Show that if (x_n) is a convergent sequence, then the sequence given by the averages

$$y_n = \frac{x_1 + x_2 + \cdots + x_n}{n}$$

also converges to the same limit.

(b) Give an example to show that it is possible for the sequence (y_n) of averages to converge even if (x_n) does not.

6 Uniform Distribution

6.1 What it is ?

From last chapter, we left a question: what is random.uniform(0,1)? It is actually a random number generator that gives us some number that uniformly distribution from 0 to 1. And after we know that, we could verify the estimated result with theoretical one:

$$\text{If } X \sim \mathcal{U}(0,1), \text{then} \int_0^1 P(X=x)dx = 1 \;\&\; P(X=x_i) = P(X=x_j)\; \forall\, x_i \neq x_j \in [0,1]$$
$$\text{Thus, we have: } P(X=x) = 1 \;\forall\, x \in [0,1]$$
$$\text{Then, } \mathbb{E}(X) = \int_0^1 xP(X=x)dx = \frac{x^2}{2}\Big|_0^1 = \frac{1}{2}, \text{Var}(X) = \frac{1}{12}.$$

And more generally, if $X \sim \mathcal{U}(a,b)$, then $P(X=x) \equiv f(x) = \frac{1}{b-a}, \forall\, x \in [a,b]$ and 0 otherwise.

Example 7.1: Show that: if $X \sim \mathcal{U}(a,b)$, then $\mathbb{E}(X) = \frac{b-a}{2}, \text{Var}(X) = \frac{(b-a)^2}{12}$.

Solution:

$$\mathbb{E}(X) = \int_a^b \frac{x}{b-a}dx = \frac{x^2}{2(b-a)}\Big|_a^b = \frac{b-a}{2}$$
$$\text{Var}(X) = \int_a^b \frac{x^2}{b-a}dx - \frac{(b-a)^2}{4} = \frac{a^2+b^2+ab}{3} - \frac{(b-a)^2}{4} = \frac{(b-a)^2}{12}$$

Example 7.2:
Let x ∈ [0, 1) be a uniform random number. Write down the probability distribution p(x) of x. Hence calculate analytically the probability that:
(a) x ¡ 0.3
(b) x = 1/2 exactly
(c) x = 0.500 up to 3 significant figures
(d) 0.7 ≤ x ≤ 0.9
(e) 0.7 ¡ x ¡ 0.9

Solution:
Since $x \sim \mathcal{U}[0,1)$, then generally, we could write down its pdf as: $p(x) = \mathbf{1}\{x \in [0,1)\}$, where $\mathbf{1}$ is indicator function.
(a) $P(x < 0.3) = \int_{-\infty}^{0.3} \mathbf{1}\{x \in [0,1)\} = \int_0^{0.3} \mathbf{1}\{x \in [0,1)\} = 0.3$.
(b) $P(x = \frac{1}{2}) = P(\frac{1}{2} \leq x \leq \frac{1}{2}) = \int_{\frac{1}{2}}^{\frac{1}{2}} \mathbf{1}\{\frac{1}{2} \in [0,1)\} = 0$.
(c) $P(x = 0.500) = P(0.4995 < x < 0.5005) = \int_{0.4995}^{0.5005} \mathbf{1}\{\frac{1}{2} \in [0,1)\} = 0.001$.
(d) $P(0.7 \leq x \leq 0.9) = \int_{0.7}^{0.9} \mathbf{1}\{x \in [0,1)\} = 0.2$.
(e) $P(0.7 < x < 0.9) = \int_{0.7}^{0.9} \mathbf{1}\{x \in [0,1)\} = 0.2$.

Example 7.3:
(1) Let x and y be independent random numbers in [0,1) following the uniform distribution. Calculate:
(a) the standard deviation σ_x of x.
(b) the standard deviation σ_z of z if z = x + y.

Solution:
(a)

$$\begin{aligned}\mathbb{E}[x] &= \int_0^1 x dx = \frac{x^2}{2}\Big|_0^1 = \frac{1}{2} \\ \mathrm{Var}(x) = \mathbb{E}[(x-\mathbb{E}[x])^2] &= \int_0^1 (x-\frac{1}{2})^2 dx = \frac{1}{3}(x-\frac{1}{2})^3\Big|_0^1 = \frac{1}{12} \\ \Rightarrow \sigma_x &= \sqrt{\mathrm{Var}(x)} = \frac{1}{\sqrt{12}}.\end{aligned}$$

(b)

$$\begin{aligned}\mathrm{Var}(z) &= \mathbb{E}[(x+y-\mathbb{E}[x+y])^2] = \mathbb{E}[((x-\mathbb{E}[x])+(y-\mathbb{E}[y]))^2] \quad \text{(By linearity)} \\ &= \mathbb{E}[(x-\mathbb{E}[x])^2] + \mathbb{E}[(y-\mathbb{E}[y])^2] + 2\mathbb{E}[(x-\mathbb{E}[x])(y-\mathbb{E}[y])] \quad \text{(Cov(x,y) = 0)} \\ &= \mathbb{E}[(x-\mathbb{E}[x])^2] + \mathbb{E}[(y-\mathbb{E}[y])^2] = \mathrm{Var}(x) + \mathrm{Var}(y) = \frac{1}{6}. \\ &\Rightarrow \sigma_z = \sqrt{\mathrm{Var}(z)} = \frac{1}{\sqrt{6}}.\end{aligned}$$

6.2 Linear Congruential Generator

However, how could we implement this in computer? For python, we just type random.uniform(a,b), it will give us, but if we really want to know that detail of implementing this, i think this subsection is for you.
The method is called **Linear Congruential Generator**, it is just a really simple recursive equation:

$$X_{n+1} = (aX_n + c) \bmod m$$

, here $a, c, m \in \mathcal{Z}$, and mod means means taking the remainder after division by m, sometime we use | to represent.
Then divided the outcome by m, we will get a fake uniform random number.

Example 7.4: We could set seed $X_0 = 33$, and $a = 3, c = 1, m = 7$. Then $\{X_n, n \geq 0\} = \{2, 0, 1, 4, 6, 5, 2, 0, \cdots\}$.

We could see that there is a pattern, and the period is 6.

Example 7.5: We could set seed $X_0 = 3$, and $a = 3, c = 1, m = 7$. Then $\{X_n, n \geq 0\} = \{3, 3, 3 \cdots\}$.

This is really worse case, since it just looks non-random at all - even a constant.
Therefore, we come up a really natural question, how to avoid that, or more advanced, how to make the period as high as possible. To know this, we first need to know what is the period of a linear congruential generator when x_0, a, c, m is given.

Lemma: Suppose p is prime and $j \geq 2$. Then, unless $p = j = 2$,

$$p^k \mid n \implies p^{k-j+2} \mid \binom{n}{j} \tag{1}$$

Furthermore,

$$2^k \mid n \implies 2^{k-1} \mid \binom{n}{2} \tag{2}$$

Proof: Unless $p = j = 2$, $j < p^{j-1}$. Thus, j has at most $j-2$ factors of p. Then (1) follows from the binomial identity

$$\binom{n}{j} = \frac{n}{j}\binom{n-1}{j-1}$$

(2) follows from

$$\binom{n}{2} = \frac{n}{2}(n-1)$$

Theorem Suppose that

(a) for all primes p, $p \mid m \implies p \mid a-1$

(b) $4 \mid m \implies 4 \mid a-1$

Then,

$$m \mid \frac{a^n - 1}{a-1} \implies m \mid n$$

Proof: Assume $m \mid \frac{a^n-1}{a-1}$. For simplicity of notation, let $r = a-1$. Then:

$$\frac{a^n - 1}{a-1} = \sum_{j=1}^{n} \binom{n}{j} r^{j-1} \tag{3}$$

For any odd $p \mid m$, assume that $p^k \mid n$ and $p^{k+1} \mid \frac{a^n-1}{a-1}$. Using (3), we get:

$$n \equiv -\sum_{j=2}^{n} \binom{n}{j} r^{j-1} \bmod p^{k+1} \tag{4}$$

The Lemma and the assumption that $p \mid m \implies p \mid r$ says that $p^{k-j+2}p^{j-1} = p^{k+1}$ divides each term in (4). Thus, $p^{k+1} \mid n$. Bootstrapping, we get that for any odd $p \mid m$,

$$p^k \mid \frac{a^n-1}{a-1} \implies p^k \mid n$$

If $2 \mid m$, then $2 \mid \frac{a^n-1}{a-1}$. Using (3), we get:

$$n \equiv -\sum_{j=2}^{n} \binom{n}{j} r^{j-1} \bmod 2 \tag{5}$$

The assumption that $p \mid m \implies p \mid r$ says that 2 divides each term in (6). Thus, $2 \mid n$; that is,

$$2 \mid \frac{a^n-1}{a-1} \implies 2 \mid n \tag{6}$$

If $4 \mid m$, then assume that $2^k \mid n$ and that $2^{k+1} \mid \frac{a^n-1}{a-1}$. Using (3), we get:

$$n \equiv -\sum_{j=2}^{n} \binom{n}{j} r^{j-1} \bmod 2^{k+1} \tag{7}$$

The Lemma and the assumption that $4 \mid m \implies 4 \mid r$ says that $2^{k-j+1}4^{j-1} = 2^{k+j-1}$ divides each term in (8). Since $j \geq 2$, we have $2^{k+1} \mid n$. Bootstrapping, we get that:

$$2^k \mid \frac{a^n-1}{a-1} \implies 2^k \mid n \tag{8}$$

(5) and either (7) or (9) show that:

$$m \mid \frac{a^n-1}{a-1} \implies m \mid n \tag{9}$$

Suppose the sequence x_k is defined by the recurrence:

$$x_{k+1} = ax_k + b \tag{10}$$

then, inductively, we have:

$$x_k = a^k x_0 + \frac{a^k - 1}{a-1} b \tag{11}$$

Multiplying by $a-1$ and adding 1 yields:

$$\frac{a^{k_1}-1}{a-1} \equiv \frac{a^{k_2}-1}{a-1} \bmod m \implies a^{k_1} \equiv a^{k_2} \bmod m \tag{12}$$

Therefore, to investigate the periodicity of x_k, we look at the periodicity of $\frac{a^k-1}{a-1} \bmod m$. To maximize the range of x_k, we will assume that $(a,m) = (b,m) = 1$. This implies

$$\frac{a^{k_1}-1}{a-1} \equiv \frac{a^{k_2}-1}{a-1} \bmod m \implies \frac{a^{k_1-k_2}-1}{a-1} \equiv 0 \bmod m \tag{13}$$

$$\implies \frac{a^{k_1-k_2}-1}{a-1} \equiv 0 \bmod m \tag{14}$$

That is, the period of x_k is the smallest positive n for which

$$\frac{a^n-1}{a-1} \equiv 0 \bmod m \tag{15}$$

By the theorem above, $m \mid n$ and since there are only m residue classes $\pmod m$, we must have $n = m$. Thus,

Theorem: Suppose:

(a) for all primes p, $p \mid m \implies p \mid a-1$

(b) $4 \mid m \implies 4 \mid a-1$

(c) $\gcd(b,m) = 1$

Then the modular sequence defined by

$$x_{n+1} \equiv ax_n + b \bmod m$$

has period m.
Then practically, the mostly used generator called Park and Miller generator by setting: $a = 7^5 = 16807, c = 0, m = 2^{31} - 1 = 214783647$.

6.3 Program Implementation

Simply, we just replace the rand function with our own LCG:

```
# setting up the config
a = 7 ** 5
c = 0
m = 2 ** 31 - 1

# the recursive number
x0 = 33
x1 = x0
Size = 100

# check the mean and variance
sumX = 0
sumX2 = 0
cnt = 0

# Start to find mean/variance
while cnt < Size:
    # get the random number
    x1 = (a * x0 + c) % m
    x0 = x1
    sample = x0 / m

    # calculate the related quantity
    sumX += sample
    sumX2 += sample ** 2
    cnt += 1

meanX = sumX / cnt
varX = sumX2 / cnt - meanX ** 2
```

6.4 Exercise

(1) If $U \sim \mathcal{U}(0,1)$, what is the distribution of $\zeta = -\ln(1-U) = -\ln U$.

(2) For independent r.v. $U_1, U_2, \cdots \sim \mathcal{U}(0,1)$, give an estimate for the value of $\mathbb{E}(N)$, where

$$N = \min\left\{n : \sum_{i=1}^{n} U_i > 1\right\}$$

7 More Probability Distribution

After we learning the most important uniform distribution, we could also delve into other important distribution, and in the end, we could find out why in this numerical analysis course, we will first introduce uniform distribution. Let's first begin with the most fundamental one.

7.1 Binomial Distribution

If we roll a coin, we will have two outcome, and if we roll it for five successive times, then we will get five result in line, and in order to analyze this process, we name it as binomial process.
Rigorously, we denote the probability of success (head) as p, the number of tails as n, we say if $X \sim \text{Bino}(n, p)$, where $\mathbb{N} \ni n \geq 0$ then:

$$P(X = x) = \binom{n}{k} p^k (1-p)^{n-k}$$

, for this discrete r.v. we could easily assess its expectation and variance:

$$\mathbb{E}(X) = \sum_{i=1}^{n} k \frac{n!}{k!(n-k)!} p^k (1-p)^{n-k} = np \sum_{i=1}^{n} \frac{(n-1)!}{(k-1)!(n-k)!} p^{k-1} (1-p)^{n-k} = np$$
$$\text{Var}(X) = \mathbb{E}(X^2) - (\mathbb{E}(X))^2 = np(1-p).$$

7.1.1 Simulation

We know that Binomial distribution is rolling n binomial r.v., and we could use below way to simulate binomial r.v.:

```python
import random

p = 0.6
n = 100
cnt = 0
sumX = 0
sumX2 = 0


def binomial(p):
    if x := random.uniform(0, 1) < p:
        res = 1  # success
    else:
        res = 0  # fail
    return res


while cnt < n:
    res = binomial(p)
    sumX += res
    sumX2 += res ** 2
    cnt += 1
```

```

meanX = sumX / n
varX = sumX2 / cnt - meanX ** 2
```

Example 8.1: If we have a series of independent binomal r.v. with success rate 0.6, we denote n as the number of success in 100 runs. Write a program to simulate this process, hence find out the corresponding probability.

Solution:

```python
import random

# Parameters of the binomial distribution
N = 100          # Number of trials
p = 0.6          # Probability of success
num_simulations = int(1e8)  # Number of simulations

# Function to simulate a binomial process
def simulate_binomial(N, p):
    successes = 0
    for _ in range(N):
        # Simulate a Bernoulli trial with success probability p
        if random.uniform(0, 1) < p:
            successes += 1
    return successes

# Run simulations and count cases where successes are less than 40
count_less_than_40 = 0

for _ in range(num_simulations):
    n = simulate_binomial(N, p)  # Simulate one binomial random
        variable
    if n < 40:
        count_less_than_40 += 1

# Calculate the probability
probability = count_less_than_40 / num_simulations
```

Remark:
If you run this program, the output would be around: 0.000024
And recall the knowledge you learnt in AMA1611, you will use normal distribution to approximate this process. The corresponding probability is $\Phi((40.5 - 60)/\sqrt{100 * 0.6 * 0.4}) = \Phi(-3.98) = 0.00003$.

7.2 Poisson Distribution

First, we need to define counting process $N(t)$:

A process is said to be counting process if it satisfies:

1. $N(t) \geq 0$;
2. $N(t) \in \mathbb{N}$;
3. If $0 < s < t$, then $N(s) \leq N(t)$;
4. For $0 < s < t$, $N(t) - N(s)$ is the number of events that occur in the interval (s, t].

Then if this counting process have independent increments, that is to say the events occurred in disjoint interval are independent with each other, and stationary, that is to say distribution of the number of events that occur in any interval of time depends only on the length of the time interval. Then this process is said to be a Poisson process.

Formally, a counting process is said to be Poisson process with rate $\lambda > 0$ if it satisfies:

1. $N(0) = 0$;
2. $\{N(t), t \geq 0\}$ has independent increments;
3. $P(N(t+h) - N(t) = 1) = \lambda h + \mathcal{O}(h)$;
4. $P(N(t+h) - N(t) \geq 2) = \mathcal{O}(h)$.

Then for this process we could verify its p.m.f:
Denote $P_k(t) = P(N(t) = k)$, $p(h) = P(N(h) \geq 1) = 1 - P_0(h)$.
Then from (2) we have: $P_0(t+h) = P_0(t)P_0(h) = P_0(t) - P_0(t)p(h)$.
Taking the limits and from (3) & (4): $P_0'(t) = \lim\limits_{h \to 0} \frac{P_0(t+h) - P_0(t)}{h} = \lim\limits_{h \to 0} \frac{-P_0(t)p(h)}{h} = -\lambda P_0(t)$.
Same process for $P_1(t)$, we have: $P_1'(t) = -\lambda P_1(t) + \lambda P_0(t)$.
Then we could easily extend this to $k > 1$: $P_k'(t) = -\lambda P_k(t) + \lambda P_{k-1}(t)$.
With initial condition: $P_k(0) = 0 \; \forall \; k \in \mathbb{Z}$, we conclude:

$$P_k(t) = P(N(t) = k) = e^{-\lambda t} \frac{(\lambda t)^k}{k!}.$$

The reason why we first introduce the Binomial distribution is that, Poisson process could be approximated taking limit of Binomial. If we consider one time interval, then divided it into infinite small piece, at each time range, we thought the occurrence of event as a success in Binomial process!

Example 8.2:
A manuscript is sent to a typing firm consisting of typists **A**, **B**, and**C**. If it is typed by:

- A the number of errors made is Poisson with mean $\lambda_A = 2.6$,
- B the number of errors made is Poisson with mean $\lambda_B = 3.0$,
- C the number of errors made is Poisson with mean $\lambda_C = 3.4$.

The manuscript is assigned to a typist randomly, so each typist has a probability of $p_A = p_B = p_C = \frac{1}{3}$. Let X be the number of errors. We want to calculate: $\mathbb{E}(X), \operatorname{Var}(X)$.

Solution:
Let Z denote whether typist A, B, or C is assigned. We have

$$Z = \begin{cases} \text{A with probability } \frac{1}{3}; \\ \text{B with probability } \frac{1}{3}; \\ \text{C with probability } \frac{1}{3}. \end{cases}$$

$$\begin{aligned}\mathbb{E}(X) &= \mathbb{E}[\mathbb{E}(X \mid Z)] = \frac{1}{3}\mathbb{E}(X \mid Z = A) + \frac{1}{3}\mathbb{E}(X \mid Z = B) + \frac{1}{3}\mathbb{E}(X \mid Z = C) \\ &= \frac{1}{3}(2.6 + 3 + 3.4) = 3 \\ \mathbb{E}(X^2) &= \mathbb{E}[\mathbb{E}(X^2 \mid Z)] = \mathbb{E}\left[\operatorname{Var}(X \mid Z) + (\mathbb{E}(X \mid Z))^2\right] \\ &= \frac{1}{3}(2.6 + 3 + 3.4 + (2.6)^2 + (3)^2 + (3.4)^2) = 12.107\end{aligned}$$

We have used the fact that the variance of a Poisson random variable is equal to its mean. Therefore,

$$\operatorname{Var}(X) = \mathbb{E}(X^2) - (\mathbb{E}(X))^2 = 3.107.$$

Poisson Limit Theorem:
Let p_n be a sequence of real numbers in $[0,1]$ such that the sequence np_n converges to a finite limit λ. Then:

$$\lim_{n\to\infty} \binom{n}{k} p_n^k (1 - p_n)^{n-k} = e^{-\lambda}\frac{\lambda^k}{k!}$$

Proof:
Assume $\lambda > 0$ (the case $\lambda = 0$ is easier). Then

$$\lim_{n\to\infty} \binom{n}{k} p_n^k (1-p_n)^{n-k} = \lim_{n\to\infty} \frac{n(n-1)(n-2)\dots(n-k+1)}{k!} \left(\frac{\lambda}{n}(1+\mathcal{O}(1))\right)^k \left(1 - \frac{\lambda}{n}(1+\mathcal{O}(1))\right)^{n-k}$$

$$= \lim_{n\to\infty} \frac{n^k + \mathcal{O}(n^{k-1})}{k!} \frac{\lambda^k}{n^k} \left(1 - \frac{\lambda}{n}(1+\mathcal{O}(1))\right)^n \left(1 - \frac{\lambda}{n}(1+\mathcal{O}(1))\right)^{-k}$$

$$= \lim_{n\to\infty} \frac{\lambda^k}{k!} \left(1 - \frac{\lambda}{n}(1+\mathcal{O}(1))\right)^n$$

Since

$$\lim_{n\to\infty} \left(1 - \frac{\lambda}{n}(1+\mathcal{O}(1))\right)^n = e^{-\lambda}$$

this leaves

$$\binom{n}{k} p^k (1-p)^{n-k} \simeq \frac{\lambda^k e^{-\lambda}}{k!}$$

Lastly, you could verify that: if $X \sim \text{Pois}(\lambda)$, $\mathbb{E}(X) = \lambda, \text{Var}(X) = \lambda$.

7.2.1 Simulation

You may not understand this program at this stage if you are not reviewing student, but don't worry pay some additional time will help you to understand this.

```python
import random
import math


def poisson(lam):
    L = math.exp(-lam)  # Threshold for stopping
    k = 0  # Number of events
    p = 1  # Running product of uniform random variables

    while p > L:
        k += 1
        u = random.uniform(0, 1)  # Generate a uniform random number
        p *= u  # Update the product

    return k - 1  # Subtract 1 because the loop increments one extra
        time
```

7.3 Exponential Distribution

If you pay enough attention on the poisson process, you may notice that poisson process only tells you at time t, what is the total number. But it never tells you what happen inside the whole interval. Then a natural consideration should be what is the interarrival time? That is to say, what is the distribution of the time between any two successive occurrence of event? For that question, we could guess that all the interarrival time is i.i.d, which is an educated guess, since we know from the definition of Poisson process that it has independent increments. Let's denote $T_k, k > 0$ be the waiting time of k-1 th event and k th event.
Then if $k = 1$, note that $T_1 \leq t \iff N(t) \geq 1$, then:

$$P(T_1 \leq t) = P(N(t) \geq 1) = 1 - P(N(t) = 0) = 1 - e^{-\lambda t}$$

Differentiate it, we have:

$$P(T_1 = t) = \lambda t e^{-\lambda t}$$

Similar method would be adopted for $k > 1$, you would find that:

$$P(T_k = t) = \lambda t e^{-\lambda t},\ \forall\ k > 0.$$

And we know it is i.i.d with such density function, we name it as exponential distribution.

A continuous random variable X is said to have an exponential distribution with rate parameter ($\lambda\quad 0$), then $X \sim \text{expo}(\lambda)$, if the density is:

$$f(x) = \begin{cases} \lambda e^{-\lambda x} & , x \geq 0 \\ 0 & , \text{otherwise} \end{cases}$$

Simply, you could verify that: if $X \sim \text{expo}(\lambda)$, then $\mathbb{E}(X) = \frac{1}{\lambda}, \text{Var}(X) = \frac{1}{\lambda^2}$.
But before we move on, let's check some interesting details:
Exponential distribution has memoryless property, i.e.:

$$P(X > s + t | X > t) = \frac{P(X > s + t)}{P(X > t)} = \frac{e^{-\lambda(t+s)}}{e^{-\lambda t}} = e^{-\lambda s} = P(X > s)$$

That is to say, from any point we start, the process will forget the previous state, then kick off a new exponential process.

Example 8.3:
Let X be exponential with rate parameter λ. Find $\mathbb{E}(X \mid X > 1)$.

Solution:
The conditional density of X given $X > 1$ is

$$\begin{aligned} f_{X|X>1}(x) &= \frac{f(x) I(x > 1)}{P(X > 1)} \\ &= \frac{\lambda e^{-\lambda x} I(x > 1)}{e^{-\lambda}} \\ &= \lambda e^{-\lambda(x-1)} I(x > 1) \\ &= \begin{cases} \lambda e^{-\lambda(x-1)}, & \text{if } x > 1; \\ 0, & \text{otherwise.} \end{cases} \end{aligned}$$

Then

$$\mathbb{E}(X \mid X > 1) = \int_1^\infty x \lambda e^{-\lambda(x-1)}\, dx = 1 + \frac{1}{\lambda}.$$

And also the reverse is true, i.e. only non-negative continuous r.v. with memoryless property follows exponential distribution.
Proof:
Denote $G(t) = P(X > t)$, where $X \sim f$. Then memoryless property could be rewrite as $G(t+s) = G(t)G(s)$.
Then when $t = s = 1$, $G(2) = G(1)^2$, and this could be extend to $G(x) = G(1)^x$.
Then it shows that it should be that form: $G(x) = \cdot e^{\cdot x}$, then plug in the initial condition, we have: $G(x) = e^{-\lambda x}$, which means it follows exponential distribution.
If we sum up all the interarrival time, $S_n = \sum T_i \sim \Gamma(n, \lambda)$, i.e. $f_{S_n}(t) = \lambda e^{-\lambda t}\frac{(\lambda t)^{n-1}}{(n-1)!}$.
The above could be derived by noting that $N(t) \geq n \;\Leftrightarrow\; S_n \leq t$, you could verify by yourself.

7.3.1 Simulation

You will also know that when you finished the last sub session of this chapter.

```python
import random
import math

def exponential(lam):
    u = random.uniform(0, 1)  # Generate a uniform random number
    return -math.log(u) / lam
```

Example 8.4:
Verifying exponential random numbers:
Generate $N = 10000$ random numbers $x \geq 0$ following

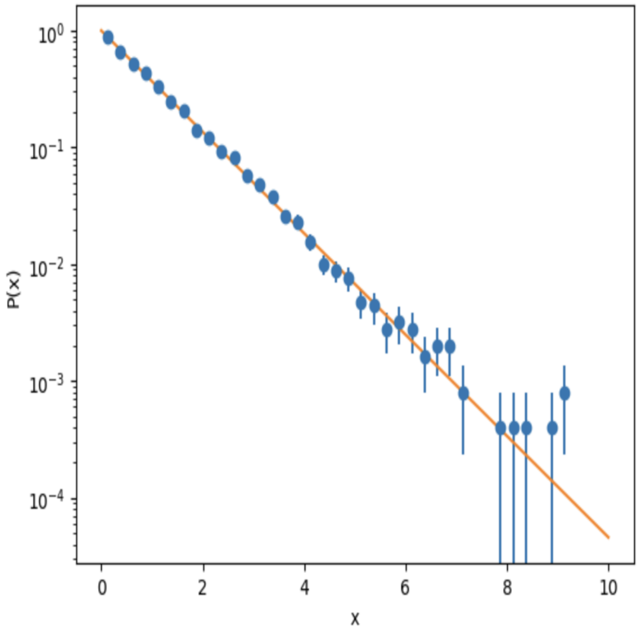
$$p(x) = \lambda e^{-\lambda x}$$

with $\lambda = 1$. As a consistency check, histogram the numbers to measure $p(x)$ and thus λ.

Solution:

```python
import numpy as np
from scipy.optimize import curve_fit
import matplotlib.pyplot as plt

### PART I: generate N exponential random numbers #################

Lambda = 1
N = 10000
r = np.random.uniform(0, 1, N)
x = -(1/Lambda) * np.log(r)

### PART II: Data Analysis ##########################################

# calculate P(x) of the random numbers by histogram
x_max = 10
n_bins = 40
```

```

n, x_edges = np.histogram(x, bins=n_bins, range=(0, x_max)) # counts &
     bin boundaries

X  = (x_edges[1:] + x_edges[:-1]) / 2  # x (midpoints of bins)
dx = x_edges[1] - x_edges[0]             # bin width
Px  = n / N / dx                    # P(x)
sigma_Px = n**0.5 / N / dx          # error of P(x)

# save histogram data to a file
data = np.column_stack((X, Px, sigma_Px))
np.savetxt('Px.txt', data, header= 'x \t P \t sigma_P', fmt='%g')

# plot P(x) vs x
plt.errorbar(X, Px, yerr=sigma_Px, fmt='o')  # plot errorbars
plt.yscale('log')                            # use semilog scales
plt.xlabel('x'); plt.ylabel('P(x)')

# fit data points with non-zero counts
def Fexp(x, a):
    return (1/a) * np.exp( -a * x)
popt, pcov = curve_fit(Fexp, X[n>0], Px[n>0], sigma=sigma_Px[n>0])
print('lambda =', *popt)

# plot fitted function
x_fit = np.linspace(0, x_max, 100)
y_fit = Fexp(x_fit, *popt)
plt.plot(x_fit, y_fit)
plt.show()
```



Example 8.5:
From the above analysis, we could know that the interarrival time also follows the exponential distribution. Then we could consider that for a restaurant, the customer's interarrival time follows the exponential distribution with parameter $\lambda = 0.1$. And at time t = 0, the restaurant will start, it can only serve one customer at a time, and the serving time is fixed that is 10. Could you simulate this process, and find the largest waiting number until t = 10000.

Solution:

```python
from math import *
import random

Lambda = 0.1


def interTime():
    u = random.uniform(0, 1)
    return -1 / Lambda * log(u)


tstop = 10000

InterArrivalTime = [0]

tnow = 0

while tnow < tstop:
    InT = interTime()
    if tnow + InT < tstop:
        tnow += InT
        InterArrivalTime.append(tnow)
    else:
        break

tau = 10


def checkin(t):
    n = len(InterArrivalTime)
    for i in range(1, n):
        if t - 1 <= InterArrivalTime[i] <= t:
            return True
    return False


t = 0
serving = False
queue = []
twait = -1
```

```
maxnum = 0

while t < tstop:
    if checkin(t):
        queue.append(1)
    if queue and not serving:
        serving = True
        twait = t + tau
    if t == twait:
        serving = False
        queue.pop()
    maxnum = max(maxnum, len(queue))
    t += 1

print(maxnum)
```

7.4 Normal Distribution & Central Limit Theorem

7.4.1 Normal Distribution

After we introduced those r.v., we come to the most important one. Why it is the most important? Next section will show that all i.i.d with large sample size with converge to normal distribution in distribution.

A continuous r.v. is said to follow normal distribution, denoted as $X \sim \mathcal{N}(\mu, \sigma^2)$, if:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}, \ \forall\, x \in \mathbb{R}$$

You may wonder, this seems really strange, does it integral to 1? Actually yes, but we could only evaluate this value explicitly.
We want to evaluate the integral $I = \int_{-\infty}^{\infty} e^{-x^2}$, since it is the main part:

$$I^2 = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} e^{-(x^2+y^2)}\, dy\, dx$$

Switching to polar coordinates:

$$= \int_0^{2\pi} \int_0^{\infty} re^{-(r^2\cos^2\theta + r^2\sin^2\theta)}\, dr\, d\theta$$

Simplifying:

$$= \int_0^{2\pi} \int_0^{\infty} re^{-r^2}\, dr\, d\theta = \int_0^{2\pi} \left[-\frac{e^{-r^2}}{2} \right]_0^{\infty} d\theta$$

$$= \int_0^{2\pi} \left(0 - \left(-\frac{1}{2} \right) \right) d\theta = 2\pi \left(\frac{1}{2} \right) = \pi$$

Then we know that: $\int_{-\infty}^{\infty} \frac{1}{\sqrt{\pi}} e^{-x^2} = 1$, here $\mu = 0, \sigma^2 = \frac{1}{2}$.

Then by simple integration by parts, we could verify below in general:

$$\mathbb{E}(X) = \mu, \mathrm{Var}(X) = \sigma^2$$

7.4.2 Central Limit Theorem

The Central Limit Theorem:
Let $X_1, X_2, \cdots, X_n$ be i.i.d. r.v. with $\mathbb{E}[X_i] < \infty$, and $\mathrm{Var}(X_n) < \infty$, then the new r.v.(arithmetic standardised mean): $\mathcal{Z}_n = \sqrt{n}\,(\frac{\overline{X_n}-\mu}{\sigma})$ converges in distribution to the standard normal r.v. as n goes to infinity, i.e. $\lim\limits_{n\to\infty} \mathcal{Z}_n \overset{(d)}{\to} \mathcal{N}(0,1)$.

Proof: We need to show the mgf of $\mathcal{Z}_n$ is equal to mgf of standardized Gaussian that's $e^{-\frac{t^2}{2}}$.
$M_{\mathcal{Z}_n}(t) = \mathbb{E}[e^{t\mathcal{Z}_n}] = \mathbb{E}[\frac{t}{\sigma\sqrt{n}} \sum\limits_{i=1}^{n}(x_i - \mu)] \overset{i.i.d}{=} \prod\limits_{i=1}^{n} \mathbb{E}[\frac{t}{\sigma\sqrt{n}}(x_i - \mu)]$.
Now, we just need to pay attention on the last term, since the r.v. are i.i.d. Let $Y_i = \frac{x_i - \mu}{\sigma}$,(it is centered new r.v. with $\mathbb{E}[Y_i] = 0, \mathbb{E}[Y_i^2] = 1$), $M_{Y_i}(\frac{t}{\sqrt{n}}) = \mathbb{E}[\frac{t}{\sqrt{n}}Y_i] \overset{Taylor}{=} 1 + M_{Y_i}^{(1)}(0)(\frac{t}{\sqrt{n}}) + \frac{1}{2}M_{Y_i}^{(2)}(0)(\frac{t}{\sqrt{n}})^2 + o(\frac{1}{n}) = 1 + \frac{t^2}{2n} + o(\frac{1}{n})$.
Then take its to limit: $\lim\limits_{n\to\infty} M_{\mathbf{Z_n}}(t) = \lim\limits_{n\to\infty} [1 + \frac{t^2}{2n} + o(\frac{1}{n})]^n = e^{\frac{t^2}{2}}$, which is the mgf of standard normal r.v.

7.4.3 Simulation

```
import random
from math import *

def normal():
  r1 = random.uniform(0,1)
  r2 = random.uniform(0,1)
  return sqrt(-2*log(r1))*sin(2*pi*r2)
```

Example 8.6: What is the p.d.f of below defined r.v.

$$x = \sqrt{-\ln r_1}\cos(2\pi r_2)$$

, where r_1 and r_2 are independent uniform deviates in [0,1).

Solution:
When we get this setting, a really natural way it to consider its conjugate: $y = \sqrt{-\ln r_1}\sin(2\pi r_2)$.
Let us firstly take its conjugate, $y = \sqrt{-\ln r_1}\sin(2\pi r_2)$, then make some transformation by simple algebra: $r_1 = e^{-(x^2+y^2)} \equiv g_1(x,y), r_2 = \frac{1}{2\pi}\tan^{-1}\frac{y}{x} \equiv g_2(x)$, if $x \neq 0$. Next the Jacobian matrix:

$$|J| = \begin{vmatrix} \frac{\partial g_1}{\partial x} & \frac{\partial g_1}{\partial y} \\ \frac{\partial g_2}{\partial x} & \frac{\partial g_2}{\partial y} \end{vmatrix} = \frac{1}{\pi} e^{-(x^2+y^2)}$$

Since $r_1 \times r_2 = [0,1) \times [0,1)$, then $x, y \in \mathbb{R}$, and:

$$f_{r_1,r_2} = f_{x,y}|J|^{-1} = 1 \quad \Leftrightarrow \quad f_{x,y} = \frac{1}{\pi} e^{-(x^2+y^2)}$$

Lastly, we could get our pdf of x:

$$f_x = \int_{-\infty}^{\infty} f_{x,y} dy = \frac{1}{\sqrt{\pi}} e^{-x^2}$$

Then we know $x \sim \mathcal{N}(0, \frac{1}{2})$.

7.5 Connection with Uniform Distribution

Have you wonder why we try to introduce the uniform distribution, and also try really hard to talk about the detail of implementation of uniform distribution? The reason is that, we could use uniform r.v. to generator all the r.v. we mentioned above.
We try to see the details of exponential and normal distribution.

7.5.1 Simulation of Exponential Distribution

If $U \sim \mathcal{U}(0,1)$, what is the distribution of $X = -\ln(1-U)$?
If we want to know the distribution of X, we could check its cumulative distribution function:

$$F(X \leq y) = F(-\ln(1-U) \leq y) = F(U \leq 1 - e^{-y}) = 1 - e^{y}$$

, since $1 - e^{-y} \in [0,1]$.
Then $P(X = y) = f(y) = \frac{d}{dy} F(X \leq y) = e^{-y}$, which means $X \sim \text{expo}(1)$.
Then generally, if we are interested in exponential distribution, we have $U \sim \mathcal{U}(0,1)$, we want $X \sim \text{expo}(\lambda)$, then $X = -\frac{1}{\lambda}\ln(1-U) = -\frac{1}{\lambda}\ln(U)$, since $1 - U, U \in [0,1]$.
Then to become more general, we could say that if we want to use uniform r.v. to simulate any p.d.f, we could first find its cumulative distribution function, then try to inverse it. i.e.

$$U = F(X) \sim \mathcal{U}(0,1) \implies X = F^{-1}(U) \sim f$$

This is called Inverse Transform Sampling.

Example 8.7:
Derive a formula for generating a random number x following the distribution:

$$p(x) = \begin{cases} x \exp(-\frac{1}{2}x^2) & \text{for } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Solution:
for $y \geq 0$:
$P(y) = p(X \leq y) = \int_0^y x \exp(-\frac{1}{2}x^2)dx = -\exp(-\frac{1}{2}x^2)\Big|_0^y = 1 - e^{-\frac{1}{2}y^2}$
for $y < 0$:
$P(y) = 0$
Then let $r \sim \mathcal{U}(0,1)$, we take: $r = 1 - e^{-\frac{1}{2}y^2}$, by simple algebra:

$$y = \sqrt{\ln \frac{1}{(1-r)^2}} \geq 0$$

7.5.2 Simulation of Normal distribution

We gain the really powerful tool from last subsection, now we try to simulate it with ITS:

$$U = \int_{-\infty}^{y} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

But, wait, how to find its inverse? What is the explicit form of this integral? We don't know !!!
Then how could we do it? One way is that we could use Rejection Sampling, but we don't want to come up with new concept and method. We could recall the **Example 8.6**, then it comes with easy that.
If we want to find the standarized one, it is really easy to extend from the above example: $x = \sqrt{-2\ln U_1}\cos(2\pi U_2) \sim \mathcal{N}(0,1)$, where $U_1 \perp U_2 \sim \mathcal{U}(0,1)$. And it is the same for its conjugate, i.e. $x = \sqrt{-2\ln U_1}\sin(2\pi U_2) \sim \mathcal{N}(0,1)$.
The above mention method is called Box-Muller algorithm.

7.6 Exercise

(1) Let $X_n \sim \text{Bin}(n, p_n)$ where $p_n \to 0$ and $np_n \to \infty$. Show that:

$$\frac{X_n - np_n}{\sqrt{np_n}} \xrightarrow{d} N(0,1) \text{ as } n \to \infty.$$

(2) Independent trials, resulting in one of the outcomes 1, 2, and 3, with respective probabilities p_1, p_2, and p_3 $\left(\sum_{i=1}^{3} p_i = 1\right)$ are performed. Let N be the number of trials needed until the initial outcome has occurred exactly 3 times. For instance, if the trial results are 3, 2, 1, 1, 2, 3, 3, ..., then $N = 7$. Find $\mathbb{E}(N)$.

(3) For example 8.6, the time complexity of **checkin** function is $\mathcal{O}(\text{len(InterArrivalTime)})$, which is too slow, could you come up a modified way to improve it to $\mathcal{O}(1)$. And can you come up with a solution that only need one **while loop**?

(4) Consider similar method for Box-Muller, how to simulate such distribution using uniform only, $f_X(x) = \frac{\beta^\alpha}{\Gamma(\alpha)} x^{\alpha-1} e^{-\beta x}$ for $x > 0$.

(5) Could you show that convergence of Poisson r.v. to normal distribution when $\lambda \to \infty$.

Final Remark: If you're into above topics, you could go further in AMA2691 & AMA4688.

8 Brownian Motion

8.1 Limiting Random Walk

Consider the symmetric random walk, which in each time unit is equally likely to take a unit step either to the left or to the right.
This is a Markov chain with transition probabilities $P_{i,i+1} = \frac{1}{2} = P_{i,i-1}$ $(i = 0, \pm 1, \pm 2, \dots)$.
Suppose that we "speed up" the process by taking smaller and smaller steps in smaller and smaller time intervals. If we go to the limit, then what we obtain is a Brownian motion.
Suppose that in each Δt time unit, the process takes a step of size Δx either to the left or to the right with equal probabilities. Let $X(0) = 0$, and for $t > 0$, let $X(t)$ denote the position at time t:

$$X(t) = \Delta x(Y_1 + \dots + Y_{\lfloor t/\Delta t \rfloor}),$$

where Y_i are assumed to be independent with

$$P(Y_i = 1) = P(Y_i = -1) = \frac{1}{2}.$$

Here, $\lfloor t/\Delta t \rfloor$ is the largest integer less than or equal to $t/\Delta t$.

8.1.1 Expectation and Variance

We can easily see that

$$E\{X(t)\} = 0, \quad \text{and} \quad \text{Var}\{X(t)\} = (\Delta x)^2 \left\lfloor \frac{t}{\Delta t} \right\rfloor.$$

We now set Δx and Δt to go to zero. To obtain a nontrivial limiting process such that the variance $\text{Var}\{X(t)\}$ is well-defined, Δx and Δt must go to zero at rates such that the variance of the process is nonzero and finite.
This can be done with $\Delta x = \sigma\sqrt{\Delta t}$ for some positive constant σ. In this case, as $\Delta t \to 0$,

$$E\{X(t)\} = 0, \quad \text{and} \quad \text{Var}\{X(t)\} \to \sigma^2 t.$$

Example 9.1: A random walker in one dimension is located at position x = 0 at time t = 0. At each time step, it moves a unit distance in the +x direction with probability p. Otherwise, it remain stationary during the step. Calculate analytically the average position $\bar{x}(t)$ and its standard deviation $\sigma_x(t)$ at time t.

Solution:
Denotes d_i as the displacement of i th movement, then clearly, we have:

$$\mathbb{E}(d_i) = 1 \cdot p + 0 \cdot (1-p) = p$$
$$\bar{x}(t) = \mathbb{E}(\sum_{i=1}^{t} d_i) = \sum_{i=1}^{t} p = tp$$

Then simply we also know that:

$$\mathrm{Var}(d_i) = \mathbb{E}(d_i^2) - (\mathbb{E}(d_i))^2 = p - p^2$$
$$\mathrm{Var}(x(t)) = \mathrm{Var}(\sum_{i=1}^{t} d_i) = \sum_{i=1}^{t} \mathrm{Var}(d_i) = tp(1-p)$$

Then it tells us: $\sigma_x(t) = \sqrt{\mathrm{Var}(x(t))} = \sqrt{tp(1-p)}$.

8.1.2 Properties of the Process

By the construction of $X(t)$, the following seems reasonable:

1. $X(t)$ is normal with mean 0 and variance $\sigma^2 t$, because it is the limit of the sum of mean-zero, i.i.d. random variables.
2. $\{X(t), t \geq 0\}$ has independent increments, because the changes of value of the random walk in nonoverlapping time intervals are independent. Formally, for $t_1 < t_2 < \cdots < t_n$,

   $$X(t_n) - X(t_{n-1}), X(t_{n-1}) - X(t_{n-2}), \ldots, X(t_2) - X(t_1), X(t_1)$$

   are independent.
3. $\{X(t), t \geq 0\}$ has stationary increments, because the distribution of the change in position of the random walk over any time interval depends only on the length of the interval: the distribution of $X(t+s) - X(t)$ does not depend on t.

Example 9.2: Let $\{B_t\}$ be a standard Brownian motion. Calculate $\mathbb{E}[B_t(B_s)^2]$ for $s > t > 0$.

Solution:
Since $(B_s)^2 = (B_s - B_t + B_t)^2 = (B_s - B_t)^2 + (B_t)^2 + 2(B_s - B_t)B_t$, we have

$$B_t(B_s)^2 = B_t(B_s - B_t)^2 + (B_t)^3 + 2(B_s - B_t)(B_t)^2.$$

Because B_t is independent of $B_s - B_t$, we have

$$\mathbb{E}[B_t(B_s)^2] = \mathbb{E}[B_t]\mathbb{E}[(B_s - B_t)^2] + \mathbb{E}[(B_t)^3] + 2\mathbb{E}[(B_s - B_t)]\mathbb{E}[(B_t)^2].$$

Since

- $\mathbb{E}[B_t] = 0$,
- $\mathbb{E}[(B_t)^3] = 0$, and
- $\mathbb{E}[(B_s - B_t)] = 0$,

we conclude that

$$\mathbb{E}[B_t(B_s)^2] = 0 + 0 + 0 = 0.$$

Example 9.3: For a random walk, it has $\frac{1}{2}$ probability to move forward one unit, and $\frac{1}{4}$ for backward one unit, then the remaining chance stays the last position. Then what is the expectation and variance of the position at time t.

Solution:
We could denote increment as s_i. Then the movement at time t could be rewrite as: $d = \sum_{i=1}^{t} s_i$.
Clearly, $\mathbb{E}(s_i) = 1 \cdot \frac{1}{2} + 0 \cdot \frac{1}{4} - 1 \cdot \frac{1}{4} = \frac{1}{4}$, and $\mathbb{E}(s_i^2) = \frac{3}{4}$.
And we know that each s_i is independent with other, then:

$$\mathbb{E}(d) = \sum_{i=1}^{t} \mathbb{E}(s_i) = \frac{t}{4}$$

But for Variance, we need to first consider:

$$\mathbb{E}(d^2) = \mathbb{E}((\sum_{i=1}^{t} s_i)^2) = \sum_{i=1}^{t} \mathbb{E}(s_i^2) + t(t-1)\mathbb{E}(s_i s_j) = \frac{3t}{4} + \frac{t(t-1)}{16} = \frac{t^2 + 11t}{16}$$

, where $i \neq j$.
Then $\text{Var}(d) = \mathbb{E}(d^2) - (\mathbb{E}(d))^2 = \frac{11t}{16}$.

8.2 Formal Definition

Definition of Brownian Motion A stochastic process $\{X(t), t \geq 0\}$ is said to be a **Brownian motion process** if

1. $X(0) = 0$;
2. $\{X(t), t \geq 0\}$ has stationary and independent increments;
3. For every $t > 0$, $X(t)$ is normally distributed with mean 0 and variance $\sigma^2 t$.

When $\sigma = 1$, the process is called a **standard Brownian motion**. Any Brownian motion $X(t)$ can be converted to the standard process by letting

$$B(t) = \frac{X(t)}{\sigma}.$$

8.2.1 Properties of Brownian Motion

A Brownian motion, as the limit of the random walk, is continuous, i.e.,

$$\lim_{h \to 0} \{X(t+h) - X(t)\} = 0$$

with probability 1. A plausibility argument of this result is obtained by noting that $X(t+h) - X(t)$ has mean 0 and variance h, so it seems reasonable that $X(t+h) - X(t)$ converges to 0, thus yielding continuity. Interestingly, although $X(t)$ is continuous in t, it is nowhere differentiable. To see why, note that $fracX(t+h) - X(t)h$ has mean 0 and variance $1/h$. The variance of the ratio tends to ∞ as $h \to 0$, so it is not surprising that it does not converge.

8.2.2 Density Function

As $X(t)$ is normal with mean 0 and variance t, its density function is

$$f_t(x) = \frac{1}{\sqrt{2\pi t}} e^{-\frac{x^2}{2t}}.$$

By the independent and stationary increment property, for $t_1 < \cdots < t_n$,

$$X(t_1), X(t_2) - X(t_1), \ldots, X(t_n) - X(t_{n-1})$$

are independent, and $X(t_k) - X(t_{k-1})$ is normal with mean 0 and variance $t_k - t_{k-1}$. For any two time points t_1 and t_2 s.t. $t_2 > t_1$, the covariance between $X(t_1)$ and $X(t_2)$ is

$$\mathrm{Cov}(X(t_2), X(t_1)) = t_1.$$

Example 9.4: The movements of a stock price can be modeled very approximately by a one-dimensional random walk. Let x_t be the closing stock price at the t th trading day. We assume that x_{t+1} is a Gaussian random number with mean x_t and standard deviation v(t)x_t , where v(t) denotes volatility of the stock.

(a) According to the definitions above, how is the volatility related to the risk and profit of an investment?

b The current closing price of a stock is \$10.5. Write a computer program which simulate the price movement and output the price after 10 trading days. Assume a constant volatility v = 4% during the period.

Solution:
(a) For any fixed x_t, when the volatility is high, the standard deviation of x_{t+1} is also high. It means that the risk and profit will also become more uncertain. Or in other word, there is more chance of getting either really high and extremely low.
(b)

```python
from random import random
from math import log, cos, sqrt, pi
from matplotlib import pyplot as plt


def Gaussian(mu, var):
    while 1:
        if (r1 := random()) and (r2 := random()):
            nml = sqrt(-2 * log(r1)) * cos(2 * pi * r2)
            return nml * sqrt(var) + mu


x0 = 10.5
vol = 0.04
smDay = 10
dayLine = list(range(smDay + 1))
priceList = [x0] * (smDay + 1)
```

```python

for day in range(smDay):
    mu = x0
    var = vol * x0
    x0 += Gaussian(mu, var)
    priceList[day + 1] = x0

print(f"The␣10␣th␣closed␣price␣is␣${round(priceList[-1],2)}")

plt.plot(dayLine, priceList)
plt.xlabel('day')
plt.ylabel('Closed␣Price')
plt.show()
```

8.3 Program Details

We define that Diffusion coefficient, D (related to volatility in stock price): $D = \frac{1}{2}\lim_{t\to\infty}\frac{\text{MSD}}{t} = \frac{1}{2}\lim_{t\to\infty}\frac{\text{Var}(X_t)}{t} = \frac{1}{2}\sigma^2$

Example 9.5: A particle carries out a correlated random walk in 1D with a probability p=0.1 of keeping the previous direction at each time step. Otherwise, the direction is flipped. Perform a walk with 1e6 steps. Hence, calculate MSD for duration $\tau = 1$ to 1000 using moving time window averaging approach. Estimate the diffusion coefficient.

Solution:

```python
### generate correlated random walk and measure diffusion coefficient
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

p = 1/10  # probability of keeping previous direction
t_max = 1000000  # no. of steps of random walk

x = np.zeros(t_max) # x stores time sequence of positions
x[0] = 0 # initial position
dir = (-1)**np.random.randint(0,1) # initial direction

# random walk
for j in range(1,t_max):
    if np.random.rand() > p:
      dir *= -1  # change direction
    x[j] = x[j-1] + dir  # a step of walk

# plotting trajectory
plt.figure()
plt.plot(x)
plt.xlabel('time');  plt.ylabel('x')
plt.show()

# msd during time tau, using a moving time window averaging approach
tau = 1            # minimum tau
tau_max = 1000     # maximum tau
inc_factor = 1.2  # next value of tau increased by this factor
tau_array = np.array([])
msd_array = np.array([])

while tau < tau_max:
  r2sum = 0
  cnt = 0
  for t in range(t_max - tau): # scan through the whole walk
    r2sum += (x[t + tau] - x[t])**2  # squared displacement
    cnt += 1
  msd = r2sum/cnt
  tau_array = np.append(tau_array, tau) # store tau in array
  msd_array = np.append(msd_array, msd) # store msd in array
  tau = round(tau*inc_factor+1) # increase tau

# plotting msd against tau
plt.figure()
plt.loglog(tau_array, msd_array)
plt.xlabel('tau');   plt.ylabel('MSD')

### find diffusion coefficient
tau_min = 100 # msd proportional to tau for tau in [tau_min, tau_max]
ind = tau_array > tau_min  # index of array elements for calculating D
D_array = np.divide(msd_array[ind], tau_array[ind])/2 # find D for
    each tau
D = np.mean(D_array)  # average over D for various values of tau
print(D)
msd_predict = 2*D*tau_array # check prediction of msd based on D
plt.loglog(tau_array[ind], msd_predict[ind], color='red') # plot for
    chekcing
plt.show()
```

8.4 Exercise

(1) Consider the distribution of $X(s)|X(t) = B$, where $X(u)$ is SBM, $s < t$.

(2) Let $X(t)$ be a standard Brownian motion. Fix a value $x > 0$ and a time $t > 0$. Consider the collection of sample paths $X(u)$ for $u \geq 0$ with $X(0) = 0$ and for which $X(t) > x$. Since $X(u)$ is continuous and $X(0) = 0$, there exists a (random) time τ at which the Brownian motion $X(u)$ first attains the value x. Or formally, $\tau_x = \{u \geq 0; X(u) = x\}$, what is the distribution of τ_x?

(3) Define new process $X^*(u) = X(u)$for $u \leq \tau$, and $-x + \{X(u) - x\}$for $u > \tau$., then show that $\mathbb{P}(X^*(u) > x) = \mathbb{P}(X(u) > x)$.

Final Remark: AMA3658 & AMA4380 will provide more details of BM.