Lecture 4

Learning outcomes:

- *Numerical Calculations with NumPy:*
- *Numerical differentiation and integration*

Numerical differentiation

- **Numerical differentiation is the process of finding the numerical value of a derivative of a given function at a given point.**

- **Principles and rules of differentiation**

❖The process of finding derivatives of a function is known as differentiation. Several principles and rules govern the differentiation of functions. They allow us to differentiate various types of functions with ease.

Here are some key principles and rules:

1. Power rule: The power rule states that if f(x) = x^n, where n is a constant, then the derivative f'(x) is given by f'(x) = n * x^(n-1).

2. Constant rule: The derivative of a constant function is zero. If f(x) = c, where c is a constant, then f'(x) = 0.

3. Sum and difference rule: For functions f(x) and g(x), the derivative of their sum or difference is the sum or difference of their derivatives. If f(x) and g(x) are differentiable functions, then (f(x) ± g(x))' = f'(x) ± g'(x).

4. Product rule: The product rule allows us to differentiate the product of two functions. If f(x) and g(x) are differentiable functions, then the derivative of their product, f(x) * g(x), is given by (f(x) * g(x))' = f'(x) * g(x) + f(x) * g'(x).

5. Quotient rule: The quotient rule enables us to differentiate the quotient of two functions. If f(x) and g(x) are differentiable functions and g(x) ≠ 0, then the derivative of their quotient, f(x) / g(x), is given by [(f'(x) * g(x)) - (f(x) * g'(x))] / [g(x)]^2.

6. Chain rule: The chain rule allows us to differentiate composite functions. If y = f(g(x)), where both f and g are differentiable functions, then the derivative of y with respect to x can be calculated as dy/dx = (dy/dg) * (dg/dx), where dy/dg represents the derivative of y with respect to g, and dg/dx represents the derivative of g with respect to x.

- ✓Understanding these principles and rules provides a solid foundation for effectively calculating derivatives and applying them to various functions. They serve as building blocks for more complex derivative calculations and enable us to explore the intricacies of mathematical functions in a systematic manner.

Importance of calculating derivatives

- Derivatives enable us to determine the slope of a curve at any specific point. By examining the slope, we can discern whether a function is increasing or decreasing, identify its maximum or minimum points, and analyze its concavity.

- Such information is crucial for understanding the dynamics of physical systems, modeling real-world phenomena, optimizing processes, and making predictions in various fields. Python provides a versatile platform for performing derivative calculations.

Uses of derivatives

- The ability to calculate derivatives has far-reaching implications across numerous disciplines.

- In physics and engineering, derivatives are essential for analyzing motion, determining velocities and accelerations, and designing efficient systems.
- In economics and finance, derivatives aid in understanding the behavior of markets, optimizing investment strategies, and valuing financial instruments.
- Derivatives are fundamental in fields such as biology, chemistry, and medicine, where they help model rates of reactions, growth patterns, and physiological processes.

Derivative Functions in Python

- In the realm of mathematics and science, derivatives hold immense significance as they allow us to understand the rate of change of a function at any given point. Using a versatile tool like Python can help us leverage this mathematical model to map real-world data.

- Put simply, taking a Python derivative measures how a function responds to infinitesimally small changes in its input. It provides valuable insights into the behavior, trends, and characteristics of mathematical functions.

- If we have a function f(x), the derivative of that function at point x can be calculated as the limit of the difference quotient as h approaches zero:

➢ f'(a) = lim(h -> 0) [(f(a+b) - f(a))/h]

- However, computing derivatives can be a time−consuming and error−prone process when done by hand.

- Luckily, numerical computing libraries like NumPy can make this process much easier, enabling us to calculate derivatives quickly and accurately.

- So, we'll delve into the world of numerical differentiation and explore how to use NumPy's gradient function to compute the derivative of one−dimensional and multi−dimensional functions.

Overview of how to compute derivatives using NumPy

❑ *Step 1: Define the Function*

- The first step is to define the function you want to take the derivative of.
- Let's say we want to find the derivative of the function f(x) = x^2.
- We can define this function in NumPy using the following code:

```
import numpy as np

def f(x):
    return x**2
```

❑ *Step 2: Define the Domain*

- The next step is to define the domain of the function. In other words, you need to specify the values of x that you want to compute the derivative at.
- For example, let's say you want to compute the derivative of f(x) = x^2 at x = 2.
- You can define this domain using the following code:

```
x = 2
```

❑ *Step 3: Compute the Derivative*

- Once we have established the function and domain, NumPy's gradient function comes into play to calculate the derivative.
- You can call function np.gradient to find the derivative of function f(x) = *x2*.

➢The first argument is an array of function values, the second defines the spacing Δx for the evaluation. Here pass it as an array of x values, the differences will be calculated automatically.

- The beauty of the gradient function is its simplicity, requiring only two arguments: the function we wish to derive and the values of x where we want to compute the derivative.
- Let's take a closer look at how to utilize this function:

derivative = np.gradient(f(x), x)

- In this case, the gradient function will compute the derivative of f(x) = x^2 at x = 2. The output of the function will be a single value representing the value of the derivative at that point.

- But what if we want to compute the derivative of a function over a range of values? We can easily do an updation in our code to do that. Let's say we want to compute the derivative of f(x) = x^2 over the range x = [0, 1, 2, 3].
- We can do that by updating our code as follows:

```
x = np.array([0, 1, 2, 3])
derivative = np.gradient(f(x), x)
```

- In this case, the gradient function will compute the derivative of f(x) = x^2 at each point in the domain and return an array representing the values of the derivative at each point.

❑ Multi−dimensional Derivatives

- Another useful feature of the gradient function in NumPy is its ability to compute partial derivatives of multi−dimensional functions.
- This means that you can find the rate of change of a function with respect to each of its variables separately.
- To compute partial derivatives, all you need to do is define the function in terms of multiple variables and provide a list of values for each variable.
- NumPy's gradient function will then return an array of partial derivatives for each variable at each point in the domain.

- Let's say we want to compute the partial derivatives of the function f(x, y) = x^2 + y^2. We can define this function in NumPy as follows:

```
def f(x, y):
    return x**2 + y**2
```

- We can then define the domain as x = [1, 2, 3] and y = [4, 5, 6] using the following code:

```
x = np.array([1, 2, 3])
y = np.array([4, 5, 6])
dx, dy = np.gradient(f(x, y), x, y)
```

- ✓The output of this function will be two arrays representing the values of the partial derivatives with respect to x and y at each point in the domain.

Numerical methods for calculating derivatives

- Numerical differentiation methods provide an approximation of the derivative by computing the slope of a function based on a finite difference. These methods are particularly useful when an analytical expression for the function is not available or when dealing with complex functions.

- Here, we will explore the difference quotient method and four commonly used numerical differentiation techniques: forward difference, central difference, the Newton-raphson method, and the five-point stencil method.

Difference quotient

- The difference quotient method is a fundamental approach to numerical differentiation. It approximates the derivative of a function by calculating the slope between two nearby points. Given a function f(x), the difference quotient is defined as:

$$f'(x) \approx (f(x + h) - f(x)) / h$$

- Here, h is a small step size that determines the distance between the two points. By choosing a small enough h, we can obtain an approximation of the derivative at a specific point.

Forward difference

- The forward difference method approximates the derivative using the slope between two points, where the second point lies slightly ahead of the first. It can be expressed as:

$$f'(x) \approx (f(x + h) - f(x)) / h$$

- This method provides a simple and straightforward way to estimate the derivative, but it introduces some error due to the asymmetry of the difference.

Central difference

- The central difference method addresses the asymmetry issue of the forward difference method by considering the slopes on both sides of the point of interest. It calculates the derivative using the slopes between two points, where one lies slightly ahead and the other lies slightly behind the point.
- The formula for the central difference method is:

$$f'(x) \approx (f(x + h) - f(x - h)) / (2 * h)$$

- By averaging the slopes from both directions, the central difference method yields a more accurate estimation of the derivative.

- **Forward and Central difference.**

Forward Difference

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

Central Difference

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

Example: Forward and Central difference

```
def forward_difference(f, x, h=1e-5):
    return (f(x + h) - f(x)) / h

def central_difference(f, x, h=1e-5):
    return (f(x + h) - f(x - h)) / (2 * h)

# Function to differentiate
f = lambda x: x**2

# Point at which to differentiate
x = 1

forward_diff = forward_difference(f, x)
central_diff = central_difference(f, x)

print("Forward Difference Result:", forward_diff)
print("Central Difference Result:", central_diff)
```

✓ **Output**:

Forward Difference Result: 2.00001000001393

Central Difference Result: 2.000000000002

Root-Finding Algorithms: Newton-Raphson Method

- The Newton-Raphson method is an iterative root-finding algorithm for continuous and differentiable functions.
- **Formula**

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

Example: Root-Finding Algorithms: Newton-Raphson Method

```
def newton_raphson(f, df, x0, tol=1e-5, max_iter=100):
    x = x0
    for _ in range(max_iter):
        x_new = x - f(x) / df(x)
        if abs(x_new - x) < tol:
            return x_new
        x = x_new
    raise ValueError("Convergence not achieved")
# Function whose root is to be found
f = lambda x: x**2 - 2
# Derivative of the function
df = lambda x: 2*x
# Initial guess
x0 = 1
root = newton_raphson(f, df, x0)
print("Root found:", root)
```

✓ **Output**:

Root found: 1.4142135623746899

Five-point stencil method

- The five-point stencil method further improves the accuracy of the derivative approximation by incorporating additional neighboring points. It employs a weighted combination of the function values at five points: two on each side of the point of interest and the point itself.
- The formula for the five-point stencil method is:

$$f'(x) \approx (-f(x + 2h) + 8f(x + h) - 8f(x - h) + f(x - 2h)) / (12 * h)$$

- By utilizing a larger number of function values, the five-point stencil method reduces the error and provides a more precise estimate of the derivative.

Numerical integration

Implementing Numerical Integration

- Numerical integration is used to approximate the definite integral of a function when an analytical solution is difficult or impossible. We will discuss two common numerical integration methods: the trapezoidal rule and Simpson's rule.

- **Trapezoidal Rule**

➢The trapezoidal rule approximates the area under the curve as a series of trapezoids.

$$\int_a^b f(x)\,dx \approx \frac{b-a}{2}[f(a)+f(b)]$$

For a more accurate result, we use multiple intervals:

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{n} \frac{x_i - x_{i-1}}{2}[f(x_i)+f(x_{i-1})]$$

Example: Trapezoidal Rule

```
import numpy as np

def trapezoidal_rule(f, a, b, n):
    x = np.linspace(a, b, n+1)
    y = f(x)
    dx = (b - a) / n
    integral = (dx / 2) * np.sum(y[:-1] + y[1:])
    return integral

# Function to integrate
f = lambda x: x**2

# Integration limits
a, b = 0, 1

# Number of intervals
n = 100

result = trapezoidal_rule(f, a, b, n)
print("Trapezoidal Rule Result:", result)
```

✓ **Output**:

Trapezoidal Rule Result: 0.3333499999999999

NumPy trapz()

- The trapz() function computes the definite integral of a given array using the trapezoidal rule. It approximates the area under the curve defined by the input array using a series of trapezoids.
- The syntax of trapz() is:

  numpy.trapz(y, x = None, dx = 1.0, axis = -1)

- The trapz() function takes following arguments:

- y - input array containing the y-coordinates of the curve
- x (optional) - input array containing the x-coordinates of the curve
- dx (optional) - the spacing between the x-coordinates
- axis (optional) - the axis along which the integration is performed

- The numpy.trapz() function returns the approximate definite integral of the input array using the trapezoidal rule.

Compute Definite Integral Using np.trapz()

- **Example 1**

```
import numpy as np

# create an array of y-coordinates
y = np.array([1, 2, 3, 4, 5])

# compute the definite integral using numpy.trapz()
area = np.trapz(y)
print(area)
```

✓**Output**:

```
12.0
```

▪ **Example 2**

```
import numpy as np

# create an array of y-coordinates
y = np.array([2, 5, 7, 3, 6, 9, 1])
# compute the definite integral using numpy.trapz()
area = np.trapz(y)
print("Area under the curve:", area)
```

✓**Output**:

```
Area under the curve: 31.5
```

- In the above examples, we have the y array representing the y-coordinates of a curve.

- The np.trapz() function is used to calculate the definite integral of the curve, approximating the area under the curve using the trapezoidal rule.

- The resulting area is stored in the area variable and then printed.

Use of x and dx Argument in trapz()

- **Example 3**

```
import numpy as np
# create an array of y-coordinates
y = np.array([1, 2, 3, 4, 5])
# create an array of x-coordinates
x = np.array([0, 1, 2, 3, 4])
# specify the spacing between x-coordinates
dx = 0.5
```

```
# compute the definite integral using numpy.trapz() with optional arguments
area = np.trapz(y, x=x, dx=dx)
print("Area under the curve:", area)
```

✓**Output**:

```
Area under the curve: 12.0
```

- ✓Here, the x array is provided to specify the x-coordinates, and the dx argument is used to specify the spacing between the x-coordinates.
- ✓By providing x and dx argument, we can compute the definite integral using non-equally spaced x-coordinates and a specific spacing between the x-coordinates.

- **Example 4: trapz() With 2-D Array**

➢The axis argument defines how we can compute definite integrals of elements in a 2-D array.

✓If axis = None, the array is flattened and the definite integral of the flattened array is computed.

✓If axis = 0, the definite integral is calculated column-wise.

✓If axis = 1, the definite integral is calculated row-wise.

Let's see an example.

```
import numpy as np

# create a 2-D array
array1 = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# calculate the definite integral of the flattened array
result1 = np.trapz(array1.flatten())

print('Definite integral of the flattened array:', result1)
```

```
# calculate the definite integral column-wise (axis=0)
result2 = np.trapz(array1, axis=0)

print('\nDefinite integrals column-wise (axis=0):')
print(result2)

# calculate the definite integral row-wise (axis=1)
result3 = np.trapz(array1, axis=1)

print('\nDefinite integrals row-wise (axis=1):')
print(result3)
```

✓**Output**:

Definite integral of the flattened array: 40.0

Definite integrals column-wise (axis=0):
[8. 10. 12.]

Definite integrals row-wise (axis=1):
[4. 10. 16.]

- **Simpson's Rule**

➢ Simpson's rule provides a more accurate approximation by using parabolic segments.

$$\int_a^b f(x)\,dx \approx \frac{b-a}{6}[f(a) + 4f(\frac{a+b}{2}) + f(b)]$$

For multiple intervals:

$$\int_a^b f(x)\,dx \approx \frac{dx}{3}[f(x_0) + 4\sum_{\text{odd}} f(x_i) + 2\sum_{\text{even}} f(x_i) + f(x_n)]$$

Example: Simpson’s Rule

```
def simpsons_rule(f, a, b, n):
    if n % 2:
        n += 1 # n must be even
        x = np.linspace(a, b, n+1)
        y = f(x)
        dx = (b - a) / n
        integral = (dx / 3) * np.sum(y[0:-1:2] + 4*y[1::2] + y[2::2])
        return integral

result = simpsons_rule(f, a, b, n)
print("Simpson's Rule Result:", result)
```

✓**Output**:

Simpson's Rule Result: 0.3333333333333333