Differentiation of univariate functions.

Definition 1. (Derivative of a univariate function) Let $f : \mathbb{R} \to \mathbb{R}$. We say that f is differentiable at a point x if the following limit exists

$$\lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \text{ with } h \in \mathbb{R}.$$

In [2]:

```
def differentiate(f, x, h=0.0001):
    return (f(x+h) - f(x)) / h

# Example usage:
def f(x):
    return x**2

x = 2
derivative = differentiate(f, x)
print("The derivative of f(x) = x^2 at x =", x, "is", derivative)

# This implementation uses the central difference formula for numerical differentiation, which is given by:
# f'(x) ≈ (f(x + h) - f(x - h)) / (2h)
# where 'h' is a small positive number. The default value of h is 0.0001, but you can adjust this to get more accurate results if needed.
```

```
The derivative of f(x) = x^2 at x = 2 is 4.0001000000078335
```

In [4]:

```
import math

def differentiate(f, x, h=0.0001):
    return (f(x+h) - f(x)) / h

# Example usage:
def f(x):
    return math.sin(x)

x = math.pi / 4
derivative = differentiate(f, x)
print("The derivative of f(x) = sin(x) at x =", x, "is approximately", derivative)

# In this example, we use the math library to compute the sine function.
# The function f(x) = sin(x) is then differentiated at x = π/4 using the differentiate function.
# The result should be approximately 0.7071067811865475, which is the cosine of π/4.

# Note that this implementation uses the central difference formula for numerical differentiation,
# so the results may not be very accurate for functions with sharp peaks or valleys. In such cases,
# you may need to use more advanced techniques, such as symbolic differentiation or automatic differentiation,
# to obtain more accurate results.
```

```
The derivative of f(x) = sin(x) at x = 0.7853981633974483 is approximately 0.7070714246681931
```

In [6]:

```
# how to get differentiating exp
# Importing library
import sympy as sym

# Declaring variables
x, y, z = sym.symbols('x y z')

# expression of which we have to find derivative
exp = x**3 * y + y**3 + z

# Differentiating exp with respect to x
derivative1_x = sym.diff(exp, x)
print('derivative w.r.t x: ',
      derivative1_x)

# Differentiating exp with respect to y
derivative1_y = sym.diff(exp, y)
print('derivative w.r.t y: ',
      derivative1_y)
```

```
derivative w.r.t x:  3*x**2*y
derivative w.r.t y:  x**3 + 3*y**2
```

Power Rule

1. Power Rule

In general : $f'(x^n) = nx^{(n-1)}$

Example, Function we have : $f(x) = x^5$

It's derivative will be : $f'(x) = 5x^{(5-1)} = 5x^4$

In [30]:
```
import sympy as sym

#Power rule
x = sym.Symbol('x')
f = x**5
derivative_f = f.diff(x)
derivative_f
```

Out[30]: $5x^4$

Product Rule

Let u(x) and v(x) be differentiable functions. Then the product of the functions u(x)v(x) is also differentiable.

(uv)′ = u′ v + uv′

In [31]:
```
#Product Rule
x = sym.Symbol('x')
f = sym.exp(x)*sym.cos(x)
derivative_f = f.diff(x)
derivative_f
```

Out[31]: $-e^x \sin(x) + e^x \cos(x)$

Chain Rule

The chain rule calculate the derivative of a composition of functions.

- Say, we have a function h(x) = f(g(x))
- Then according to chain rule: h′(x) = f ′(g(x)) g′(x)
- Example: f(x) = cos(x**2)

In [33]:
```
#Chain Rule
x = sym.Symbol('x')
f = sym.cos(x**2)
derivative_f = f.diff(x)
derivative_f
```

Out[33]: $-2x \sin(x^2)$

Higher-order derivatives

In [8]:
```
# Finding second derivative
# of exp with respect to x
exp = x**3 * y + y**3 + z
derivative2_x = sym.diff(exp, x, 2)
print('second derivative w.r.t. x: ',
      derivative2_x)

# Finding second derivative
# of exp with respect to y
derivative2_y = sym.diff(exp, y, 2)
print('second derivative w.r.t. y: ',
      derivative2_y)
```

```
second derivative w.r.t. x:  6*x*y
second derivative w.r.t. y:  6*y
```

In [17]:
```
import numpy as np
from scipy.misc import derivative

def higher_order_derivative(func, order, var=0, point=[]):
    args = point[:]
    def wraps(x):
        args[var] = x
        return func(*args)
    return derivative(wraps, point[var], n=order, dx=1e-6)

# Example 1
def f(x):
    return x**3 + 2*x**2 + 3*x + 4

print("Second derivative of f at x=1:", higher_order_derivative(f, 2, 0, [1]))


# Example 2
def g(x, y):
    return x**3 + y**3

print("Second partial derivative of g wrt x at (1, 2):", higher_order_derivative(g, 2, 0, [1, 2]))
print("Second partial derivative of g wrt y at (1, 2):", higher_order_derivative(g, 2, 1, [1, 2]))
```

```
Second derivative of f at x=1: 9.99733629214461
Second partial derivative of g wrt x at (1, 2): 6.000533403494046
Second partial derivative of g wrt y at (1, 2): 12.002843163827492
```

Partial differentiation

A partial derivative is a function's derivative that has two or more other variables instead of one variable. Because the function is dependant on several variables, the derivative converts into the partial derivative.

For example, where a function f(b,c) exists, the function depends on the two variables, b and c, where both of these variables are independent of each other. The function, however, is partially dependant on both b and c. Therefore, to calculate the derivative of f, this derivative will be referred to as the partial derivative. If you differentiate the f function with reference to b, you will use c as the constant. Otherwise, if you differentiate f regarding c, you will take b as the constant instead.

In [18]:
```
from sympy import symbols, cos, diff

a, b, c = symbols('a b c', real=True)
f = 5*a*b - a*cos(c) + a**2 + c**8*b

#differntiating function f in respect to a
print(diff(f, a))
```

```
2*a + 5*b - cos(c)
```

Gradients

The gradient of a function simply means the rate of change of a function. We will use numdifftools to find Gradient of a function.

In [19]:
```
pip install numdifftools
```

```
Collecting numdifftools
  Downloading numdifftools-0.9.41-py2.py3-none-any.whl (100 kB)
Requirement already satisfied: scipy>=0.8 in d:\anacondasucks\lib\site-packages (from numdifftools) (1.7.1)
Requirement already satisfied: numpy>=1.9 in d:\anacondasucks\lib\site-packages (from numdifftools) (1.20.3)
Installing collected packages: numdifftools
Successfully installed numdifftools-0.9.41
Note: you may need to restart the kernel to use updated packages.
```

In [20]:
```
# Input : x^4+x+1
# Output :Gradient of x^4+x+1 at x=1 is  4.99

# Input :(1-x)^2+(y-x^2)^2
# Output :Gradient of (1-x^2)+(y-x^2)^2 at (1, 2) is  [-4.  2.]

import numdifftools as nd


g = lambda x:(x**4)+x + 1
grad1 = nd.Gradient(g)([1])
print("Gradient of x ^ 4 + x+1 at x = 1 is ", grad1)

def rosen(x):
    return (1-x[0])**2 +(x[1]-x[0]**2)**2

grad2 = nd.Gradient(rosen)([1, 2])
print("Gradient of (1-x ^ 2)+(y-x ^ 2)^2 at (1, 2) is ", grad2)
```

```
Gradient of x ^ 4 + x+1 at x = 1 is  4.999999999999998
Gradient of (1-x ^ 2)+(y-x ^ 2)^2 at (1, 2) is  [-4.  2.]
```

In [21]:
```python
import numpy as np

def partial_derivative(func, var=0, point=[]):
    args = point[:]
    def wraps(x):
        args[var] = x
        return func(*args)
    return derivative(wraps, point[var], dx=1e-6)

def gradient(func, point=[]):
    return [partial_derivative(func, var, point) for var in range(len(point))]

# Example 1
def f(x, y):
    return x**2 + y**2

print("Partial derivative of f wrt x at (1, 2):", partial_derivative(f, 0, [1, 2])) # 2.0
print("Partial derivative of f wrt y at (1, 2):", partial_derivative(f, 1, [1, 2])) # 4.0
print("Gradient of f at (1, 2):", gradient(f, [1, 2])) # [2.0, 4.0]

# Example 2
def g(x, y, z):
    return x**2 + y**2 + z**2

print("Partial derivative of g wrt x at (1, 2, 3):", partial_derivative(g, 0, [1, 2, 3])) # 2.0
print("Partial derivative of g wrt y at (1, 2, 3):", partial_derivative(g, 1, [1, 2, 3])) # 4.0
print("Partial derivative of g wrt z at (1, 2, 3):", partial_derivative(g, 2, [1, 2, 3])) # 6.0
print("Gradient of g at (1, 2, 3):", gradient(g, [1, 2, 3])) # [2.0, 4.0, 6.0]
```

```
Partial derivative of f wrt x at (1, 2): 2.000000000279556
Partial derivative of f wrt y at (1, 2): 4.000000000115023
Gradient of f at (1, 2): [2.000000000279556, 4.000000000115023]
Partial derivative of g wrt x at (1, 2, 3): 2.000000000279556
Partial derivative of g wrt y at (1, 2, 3): 3.9999999996709334
Partial derivative of g wrt z at (1, 2, 3): 6.000000000838668
Gradient of g at (1, 2, 3): [2.000000000279556, 3.9999999996709334, 6.000000000838668]
```

Gradients of vector-valued functions

In [26]:
```python
import numpy as np

def higher_order_derivative(func, order, var=0, point=[]):
    args = point[:]
    def wraps(x):
        args[var] = x
        return func(*args)
    return derivative(wraps, point[var], n=order, dx=1e-6)

def gradient(func, point=[]):
    return np.array([higher_order_derivative(func, 1, var, point) for var in range(len(point))])

# Example 1
def f(x, y):
    return np.array([x**2 + y**2, x + y])

print("Gradient of f at (1, 2):\n", gradient(f, [1, 2]))


# Example 2
def g(x, y):
    return np.array([np.sin(x), np.cos(y)])

print("Gradient of g at (np.pi/2, 0):\n", gradient(g, [np.pi/2, 0]))
```

```
Gradient of f at (1, 2):
 [[2. 1.]
 [4. 1.]]
Gradient of g at (np.pi/2, 0):
 [[0. 0.]
 [0. 0.]]
```

We can use the numpy.gradient() function to find the gradient of an N-dimensional array. For gradient approximation, the function uses either first or second-order accurate one-sided differences at the boundaries and second-order accurate central differences in the interior (or non-boundary) points.

In [27]:
```python
# Example

# create list
x1 = [7, 4, 8, 3]
x2 = [2, 6, 5, 9]
# convert the lists to 2D array using np.array
f = np.array([x1, x2])

# compute the gradient of an N-dimensional array
# and store the result in result
result = np.gradient(f)

print(result)
```

```
[array([[-5.,  2., -3.,  6.],
       [-5.,  2., -3.,  6.]]), array([[-3. ,  0.5, -0.5, -5. ],
       [ 4. ,  1.5,  1.5,  4. ]])]
```

Jacobian matrix in PyTorch

Introduction:

The Jacobian is a very powerful operator used to calculate the partial derivatives of a given function with respect to its constituent latent variables. For refresher purposes, the Jacobian of a given function $f :^n \to^m$ with respect to a vector $\mathbf{x} = \{x_1, ..., x_n\} \in^n$ is defined as

$$J_f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} & \cdots & \frac{\partial f}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

Suppose we have a vector $\mathrm{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$ and a function $\mathrm{f}(\mathrm{x}) = \mathrm{f}(\mathrm{x}_1, x_2, x_3) = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \end{bmatrix} = \begin{bmatrix} x_1 + x_2 \\ x_1 \times x_3 \\ x_2^3 \end{bmatrix}$. To calculate the Jacobian of f with respect to $\mathbf{x}$, we can use the above-mentioned formula to get

$$J_f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} & \frac{\partial f}{\partial x_3} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} \\ \frac{\partial f_3}{\partial x_1} & \frac{\partial f_3}{\partial x_2} & \frac{\partial f_3}{\partial x_3} \end{bmatrix} = \begin{bmatrix} \frac{\partial(x_1+x_2)}{\partial x_1} & \frac{\partial(x_1+x_2)}{\partial x_2} & \frac{\partial(x_1+x_2)}{\partial x_3} \\ \frac{\partial(x_1 \times x_3)}{\partial x_1} & \frac{\partial(x_1 \times x_3)}{\partial x_2} & \frac{\partial(x_1 \times x_3)}{\partial x_3} \\ \frac{\partial x_2^3}{\partial x_1} & \frac{\partial x_2^3}{\partial x_2} & \frac{\partial x_2^3}{\partial x_3} \end{bmatrix} =$$

$$\begin{bmatrix} 1 & 1 & 0 \\ x_3 & 0 & x_1 \\ 0 & 3 \times x_2^2 & 0 \end{bmatrix}$$

To achieve the same functionality as above, we can use the **jacobian()** function from Pytorch's **torch.autograd.functional** utility to compute the Jacobian matrix of a given function for some inputs.

In [42]:
```
pip install torch
```

```
Requirement already satisfied: torch in d:\anacondasucks\lib\site-packages (1.13.1)
Requirement already satisfied: typing-extensions in d:\anacondasucks\lib\site-packages (from torch) (4.4.0)
Note: you may need to restart the kernel to use updated packages.
```

In [43]:
```
from torch.autograd.functional import jacobian
from torch import tensor

#Defining the main function
def f(x1,x2,x3):
    return (x1 + x2, x3*x1, x2**3)

#Defining input tensors
x1 = tensor(3.0)
x2 = tensor(4.0)
x3 = tensor(5.0)

#Printing the Jacobian
print(jacobian(f, (x1,x2,x3)))
```

```
((tensor(1.), tensor(1.), tensor(0.)), (tensor(5.), tensor(0.), tensor(3.)), (tensor(0.), tensor(48.), tensor(0.)))
```

Taylor series

A Taylor series is a series expansion of a function about a point. A one-dimensional Taylor series is an expansion of a real function $f(x)$ about a point $x = a$ is given by

$$f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f^{(3)}(a)}{3!}(x-a)^3 + \ldots + \frac{f^{(n)}(a)}{n!}(x-a)^n + \ldots. \quad (1)$$

If $a = 0$, the expansion is known as a Maclaurin series.

Taylor's theorem (actually discovered first by Gregory) states that any function satisfying certain conditions can be expressed as a Taylor series.

We can combine these terms in a line of Python code to estimate e^2 The code below calculates the sum of the first five terms of the Taylor Series expansion of e^x, where x=2

In [37]:
```
# We can combine these terms in a line of Python code to estimate e^2
#  The code below calculates the sum of the first five terms of the Taylor Series expansion of e^x, where x=2


import math

x = 2
e_to_2 = x**0/math.factorial(0) + x**1/math.factorial(1) + x**2/math.factorial(2) + x**3/math.factorial(3) + x**4/math.factorial(4)
print(e_to_2)
```

```
7.0
```

Our Taylor Series approximation of e^2 was calculated as 7.0. Let's compare our Taylor Series approximation to Python's math.exp() function. Python's math.exp() function raises e to any power. In our case, we want to use math.exp(2) because we want to calculate e^2

In [38]:
```
# Our Taylor Series approximation of e^2 was calculated as 7.0. Let's compare our Taylor Series approximation to Python's math.exp() function.
# Python's math.exp() function raises e to any power. In our case, we want to use math.exp(2) because we want to calculate e^2

print(math.exp(2))
```

```
7.38905609893065
```

Our Taylor Series approximation 7.0 is not that far off the calculated value 7.389056... using Python's exp() function.

Then lets try to use a for loop to calculate the difference between the Taylor Series expansion

In [40]:
```
import math

def func_e(x, n):
    e_approx = 0
    for i in range(n):
        e_approx += x**i/math.factorial(i)

    return e_approx
x = 5
for i in range(1,11):
    e_approx = func_e(x,i)
    e_exp = math.exp(x)
    e_error = abs(e_approx - e_exp)
    print(f'{i} terms: Taylor Series approx= {e_approx}, exp calc= {e_exp}, error = {e_error}')
```

```
1 terms: Taylor Series approx= 1.0, exp calc= 148.4131591025766, error = 147.4131591025766
2 terms: Taylor Series approx= 6.0, exp calc= 148.4131591025766, error = 142.4131591025766
3 terms: Taylor Series approx= 18.5, exp calc= 148.4131591025766, error = 129.9131591025766
4 terms: Taylor Series approx= 39.33333333333333, exp calc= 148.4131591025766, error = 109.07982576924327
5 terms: Taylor Series approx= 65.375, exp calc= 148.4131591025766, error = 83.0381591025766
6 terms: Taylor Series approx= 91.41666666666667, exp calc= 148.4131591025766, error = 56.99649243590993
7 terms: Taylor Series approx= 113.11805555555556, exp calc= 148.4131591025766, error = 35.29510354702104
8 terms: Taylor Series approx= 128.61904761904762, exp calc= 148.4131591025766, error = 19.79411148352898
9 terms: Taylor Series approx= 138.30716765873015, exp calc= 148.4131591025766, error = 10.105991443846449
10 terms: Taylor Series approx= 143.68945656966488, exp calc= 148.4131591025766, error = 4.723702532911716
```

In [41]:
```
# Examples

def taylor_series(x, n):
    """
    Calculate the value of sin(x) using the Taylor series expansion.

    x: float, the value of x in radians
    n: int, the number of terms to use in the expansion

    Returns: float, the value of sin(x)
    """
    result = 0
    for i in range(n):
        result += ((-1)**i * x**(2*i + 1))/math.factorial(2*i + 1)
    return result

def compare_to_math_sin(x, n):
    """
    Compare the result of the Taylor series expansion to the built-in
    sin function from the math module.

    x: float, the value of x in radians
    n: int, the number of terms to use in the expansion

    Returns: float, the absolute difference between the two results
    """
    return abs(taylor_series(x, n) - math.sin(x))

# Example usage:
print("Sin(π/4) using 10 terms of the Taylor series:", taylor_series(math.pi/4, 10))
print("Difference from the built-in sin function:", compare_to_math_sin(math.pi/4, 10))
```

```
Sin(π/4) using 10 terms of the Taylor series: 0.7071067811865475
Difference from the built-in sin function: 1.1102230246251565e-16
```

In [ ]:
```
```