RAMAIAH
Institute of Technology

Python Lab Manual for Second Semester Engineering Mathematics Courses

Department of Mathematics, Ramaiah Institute of Technology, Bengaluru - 54

List of Programs

Sl. No.	Program
1	Introduction to Python
2	Introduction to Symbolic Computations
3	Taylor Series Expansion for Single-Variable Functions
4	Newton-Raphson Method for Single-Variable Functions
5	Taylor Series Expansion for Two-Variable Functions
6	Newton-Raphson Method for Two-Variable Functions
7	Finding Maxima and Minima of Two-Variable Functions
8	Solving ODEs using Euler's and Modified Euler's Method
9	Solving ODEs using the Runge-Kutta 4th Order Method
10	Solving Higher-Order ODEs

Lab 01: Introduction to Python

1.1 Installation and Introduction to IDE

Python Installation

Many PCs will have python already installed. To check if you have python installed on a Windows PC, search in the start bar for Python or run the following on the Command Line (cmd.exe):

```
C:\Users\UserName>python --version
```

If not installed, install from https://www.python.org/downloads/

It is difficult to write and edit lengthy code through the command line. Therefore, many IDE's have been developed to make coding easier. Examples include:

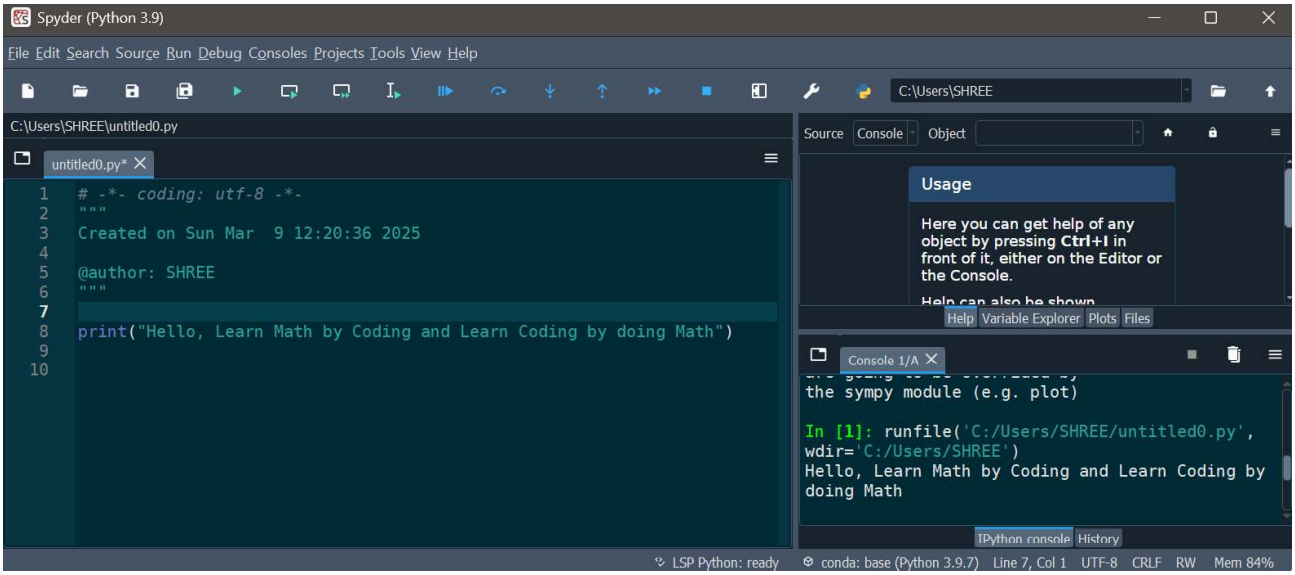
1. Spyder
2. Jupyter Notebook
3. PyCharm, etc.

Spyder

Install Spyder from https://www.spyder-ide.org

The opening page of Spyder (Scientific Python Development Environment) typically includes the following components:

1. Editor Pane (Top Left) – Where you write and edit Python scripts (.py files).
2. IPython Console (Bottom Right) – An interactive Python shell to run code and view outputs.
3. Variable Explorer (Top Right Tab) – Displays all defined variables and their values.
4. File Explorer (Top Right Tab) – Allows navigation through project directories.
5. Help Pane (Bottom Right Tab) – Provides documentation for Python functions and libraries.
6. Toolbar & Menu Bar (Top) – Contains options for running, debugging, and managing projects. The default layout provides a user-friendly interface for writing, testing, and debugging Python code efficiently.



Let's begin by writing simple basic codes in the **Python Console**:

1.2 Data Types

In Python, the data type is set when you assign a value to a variable:

Example	Data Type
`x = "Mathematics"`	`str`
`x = 20`	`int`
`x = 20.5`	`float`
`x = 1j`	`complex`
`x = ["apple", "banana", "cherry"]`	`list`
`x = {"name": "Ram", "age": 36}`	`dict`
`x = True`	`bool`

You can get the data type of any object by using the `type()` function:

In [1]:
```
x = 2j
print(type(x))
my_dict = {"a": 1, "b": 2}
print("Before adding entries:", my_dict)
# Adding two entries
my_dict["x"] = [1,2,3]
```

```
my_dict["y"] = [2,3,4]
print("After adding entries:",my_dict)
print(type(my_dict))
```

```
<class 'complex'>
Before adding entries: {'a': 1, 'b': 2}
After adding entries: {'a': 1, 'b': 2, 'x': [1, 2, 3], 'y': [2, 3, 4]}
<class 'dict'>
```

Observations:

In a Python dict (dictionary), any type of data can be stored with a corresponding key. In the example above, `a, b, x, y` are keys, and their associated values are the stored data. When solving multivariable equations, solutions are often represented as dictionaries. For instance, the solution to $(x-3)(x-4)(y-4)(y-7)=0$ is $x=3,4$ and $y=4,7$. In Python, solving this equation would yield a dictionary like `[{x: 3}, {x: 4}, {y: 4},{y: 7}]`.

1.3 Python Arithmetic Operators

Arithmetic operators are used with numeric values to perform common mathematical operations: `x=10` `y=5` `b=3`

Operator	Name	Example	Output
`+`	Addition	`x + y`	`15`
`-`	Subtraction	`x - y`	`5`
`*`	Multiplication	`x * y`	`50`
`/`	Division	`x / y`	`2.0`
`**`	Exponentiation	`x ** y`	`100000`
`%`	Modulus	`x % b`	`1`
`//`	Floor Division	`x // b`	`3`

1.4 `print()` Statement in Python

Input (Code)	Output
`print("x =", x)`	`x = 10`
`print(f"x={x}")`	`x=10`

f-strings (`f"x={x}"`) are the most recommended for modern Python

1.5 Introduction to Packages in Python

In Python, a package is a collection of modules that help organize and reuse code efficiently. It allows users to access pre-written functions for mathematical operations, data visualization, and more.

Python Packages and Their Uses

Package	Purpose
NumPy	Numerical computing, arrays, and linear algebra
SymPy	Symbolic mathematics (algebra, calculus, etc.)
Math	Standard mathematical functions
Matplotlib	Data visualization

Importing the packages in the program

Examples of Using Python Packages with Objects

Package	Import as	Method	Code	Output
NumPy	`import numpy as np`	`sin()`	`np.sin(2)`	`0.909297426825682`
SymPy	`import sympy as sp`	`sin()`	`sp.sin(2)`	`sin(2)`
Math	`import math as mt`	`sqrt()`	`mt.sqrt(16)`	`4.0`
Matplotlib	`import matplotlib.pyplot as plt`	`plot()`	`plt.plot([0,1], [0,1])`	*(Graph Output)*

1.6 Input in Python

The `input()` function in Python allows users to enter data during program execution. It reads input as a **string** by default.

Syntax:

```
variable = input("Prompt message: ")
```

Examples of `input()` in Python

Code	User Input	Output
`name = input("Enter your name: ")`	`MSR`	`name` stores `"MSR"`
`age = input("Enter your age: ")`	`25`	`age` stores `25` as an string
`age = int(input("Enter your age: "))`	`25`	`age` stores `25` as an integer
`num = float(input("Enter a number: "))`	`3.14`	`num` stores `3.14` as a float

`int(),float(),str()` are called as type casting constructors. Example:

1.7 Lists

A **list** is an ordered, mutable collection that can store multiple data types. It is defined using square brackets `[]`.

Example: `my_list = [1, "sin(x)", 3.14]`

Common List Operations

Operation	Code	Explanation	Output
Access element	`my_list[1]`	Print the second element	`"sin(x)"`
Append element	`my_list.append("new")`	Adds `"new"` at end	`[100, "sin(x)", 3.14, "new"]`

*Note: To see the output of modify, append, slice and remove operation, one has to print the modified list using print command

Input a list

In [2]:
```
#1. Getting input as `[1,2,3,4]`
my_list1 = input("Enter the list: ")
print(my_list1)

#2. Getting input as 1 2 3 4 and then converting it to list format
my_list2 = input("Enter the list: ").split()
print(my_list2)
```

```
Enter the list: [1,2,3,4]
[1,2,3,4]
Enter the list: 1 2 3 4
['1', '2', '3', '4']
```

Observarions

1. The way of giving input for `my_list2` will be easy compared to `my_list1`

2. Elements in the `my_list2` are strings, one can not perform any arithematical operations
3. To make it numeric, we have to convert the list to integer or float using `map` method as follows

In [3]:
```
my_list2=list(map(int,my_list2))
print(my_list2)
```

```
[1, 2, 3, 4]
```

Observations

- `map()` returns a map object, not a list. To convert it into a list, use `list()`.
- If we want to convert the input to float then use `list(map(float,my_list2))`

Example: Read 2 4 6 8 and append 5 to it and then print the length of the list and maximum number

In [4]:
```
list1=list(map(int,input("Enter the numbers: ").split()))
list1.append(5)
print(list1)
print(f"Length: {len(list1)}")
print(f"Maximum Value: {max(list1)}")
```

```
Enter the numbers: 2 4 6 8
[2, 4, 6, 8, 5]
Length: 5
Maximum Value: 8
```

1.8 Loops in Python

The while Loop

With the while loop we can execute a set of statements as long as a condition is true.

Example:

In [5]:
```
i=1
while i<=5:
    print(f"i = {i}: Still inside the while loop")
    i+=1
print(f"i = {i}: Exited the while loop")
```

```
i = 1: Still inside the while loop
i = 2: Still inside the while loop
i = 3: Still inside the while loop
i = 4: Still inside the while loop
i = 5: Still inside the while loop
i = 6: Exited the while loop
```

Observations

- The colon (`:`) is used after the `while` statement to indicate the start of the loop body. It tells Python that the following indented block belongs to the loop.
- Python uses indentation (spaces or tabs) to define blocks of code.
- `i+=1` is short form for incrementing `i` by 1 i.e., `i=i+1`

The for Loop

A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string)

Example:

In [6]:
```
x=["A","B","C"]

print("Traditional way")
for i in range(len(x)):
    print(x[i])
```

```
print("Python way")
for i in x:
    print(i)
```

```
Traditional way
A
B
C
Python way
A
B
C
```

Observations:

- `range(len(x))` generates indices `[0, 1, 2]` , `x[i]` is used to access elements and iterates through index value
- Iterates directly over list elements. No need for indexing (i takes values "A", "B", "C" directly).
- Direct iteration is more readable and efficient than the traditional approach.

1.9 Conditional Statements

Python supports the following conditional statements:

- Equals: a == b
- Not Equals: a != b
- Less than: a < b
- Less than or equal to: a <= b
- Greater than: a > b
- Greater than or equal to: a >= b

Examples:

In [7]:

```
a = int(input("Enter a: "))
b = int(input("Enter b: "))
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")
```

```
Enter a: 100
Enter b: 200
b is greater than a
```

Output 1:	Output 2:	Output 2:
Enter a: 100	Enter a: 100	Enter a: 100
Enter b: 45	Enter b: 500	Enter b: 100
a is greater than b	b is greater than a	a and b are equal

1.10 User defined functions

A function is a reusable block of code that executes only when called and can be invoked multiple times.

In [8]:

```
def checkEven(i):
    if i % 2 == 0:
        print(f"{i} is an even number")
    else:
        print(f"{i} is an odd number")

for i in range(1,6):
    checkEven(i)
```

```
1 is an odd number
2 is an even number
3 is an odd number
```

```
4 is an even number
5 is an odd number
```

Observations:

- Calling the function → The function `checkEven(i)` is invoked inside the loop, printing the message each time.
- `range(1,6)` → Generates numbers from 1 to 5, ensuring the function runs five times.

Example: Define functions to Read a matrix and Print the matrix:

1.11 Matrices

In [10]:

```python
def readMatrix(r):
    matrix = [list(map(int, input().split())) for _ in range(rows)]
    return matrix


def printMatrix(m):
    print("Given Matrix: ")
    for row in m:
        print(*row)

#Main Program
rows = int(input("Enter number of rows: "))
matrix = readMatrix(rows)
printMatrix(matrix)
print("This is how we access the cell matrix[i][j]:")
print(f"First row second column of the matrix is: {matrix[0][1]}")
```

```
Enter number of rows: 3
1 2 3
2 3 4
3 4 5
Given Matrix:
1 2 3
2 3 4
3 4 5
This is how we access the cell matrix[i][j]:
First row second column of the matrix is: 2
```

Observations

- The matrix is stored as a nested list (list of lists).Each row is a list, and all rows are stored inside another list.
- The function `readMatrix(r)` takes the number of rows (`r`) as input. It reads `r` lines, where each line is split into integers and stored as a list.
- `*row` is for Clean Matrix Printing, which removes list brackets and prints space-separated values.
- The program doesn't explicitly asks the number of columns, but it could be calculated using `len(matrix[0])` .
- `_` A throwaway variable, used when we don't need the loop index. `_` is the only commonly used special character as a throwaway variable instead of any letters (A-Z, a-z).
- To calculate determinant, trace and inverse of the given matrix `matrix` , we have to convert the list of lists to a `NumPy` array using the following code

`np_matrix = np.array(matrix)`

- This allows efficient vectorized calculations. Using `np_matrix` one can calculate the standard matrix operations as follows:
- np.linalg.det(np_matrix) → Computes determinant.
- np.trace(np_matrix) → Computes trace (sum of diagonal elements).
- np.linalg.inv(np_matrix) → Computes inverse (if determinant ≠ 0).

In [11]:

```python
import numpy as np

matrix = [[4, 3], [3, 2]]

np_matrix = np.array(matrix)  # Convert to NumPy array

# Now we can compute determinant, trace, and inverse
determinant = np.linalg.det(np_matrix)
```

```
trace = np.trace(np_matrix)
inverse = np.linalg.inv(np_matrix)

print("Determinant:", determinant)
print("Trace:", trace)
print("Inverse:")
for _ in inverse:
    print(*_)
```

```
Determinant: -1.0
Trace: 6
Inverse:
-2.0 3.0
3.0 -4.0
```

Lab 02: Introduction to Symbolic Computations

2.1 Defining Symbols

`import sympy as sp`

To define one symbol: `x = sp.Symbol('x')`

To define more symbols: `x, y = sp.symbols('x y')`

2.2 Standard Symbolic Computations in Python

Operation	Code	Output
Expanding Expressions	`sp.expand((x + y) ** 2)`	`x**2 + 2*x*y + y**2`
Factoring Expressions	`sp.factor(x**2 - y**2)`	`(x - y)(x + y)`
Solving Equations	`sp.solve(x**2 - 4, x)`	`[-2, 2]`
Differentiation	`sp.diff(x**3 + 2*x, x)`	`3*x**2 + 2`
Higher-Order Derivatives	`sp.diff(x**4 + 3*x**2, x, 2)`	`12*x**2 + 6`
Partial Derivative	`sp.diff(x**2*y + y, y)`	`x**2 + 1`
Integration	`sp.integrate(x**2, x)`	`x**3/3`
Substituting Values	`(x**2 + y).subs(x, 2)`	`4 + y`
Pretty Printing	`sp.pprint(((x + y) ** 2)/(x-y))`	Print the expression in more readable format

Example: Given `f(x,y) = x^2*y^2-x*y^2-6*y^2`, Perform the following things in order

1. Factor this expression
2. Differentiate w.r.t y twice
3. solve the obtained expression for x

In [12]:

```
import sympy as sp

x,y=sp.symbols('x y')

f=x**2*y**2 - x*y**2 - 6*y**2
print("Given function: ")
sp.pprint(f)
fctr_f = sp.factor(f)
print("Factorization: ", fctr_f)
fyy = sp.diff(f,y,2)
print("Derivative w.r.t. y: ", fyy)
sol = sp.solve(fyy,x)
print("Solution: x=", sol)
```

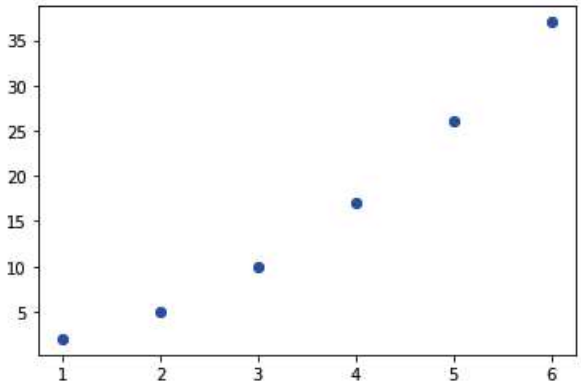
```
Given function:
 2  2      2      2
x ·y  - x·y  - 6·y
Factorization:  y**2*(x - 3)*(x + 2)
Derivative w.r.t. y:  2*(x**2 - x - 6)
Solution: x= [-2, 3]
```

2.3 Plotting

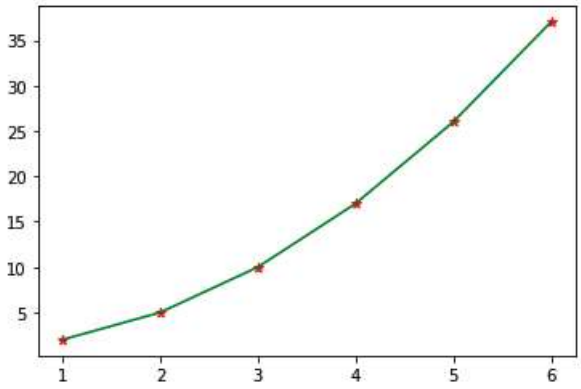
Data plot (Scatter Plot)

A scatter plot visualizes the relationship between two variables by displaying data points on a Cartesian plane. It helps identify trends, correlations, or patterns in data.

In [13]:
```
import matplotlib.pyplot as plt
import numpy as np

x = [1,2,3,4,5,6]
y = [2,5,10,17,26,37]
plt.scatter(x, y, color='b', marker='o')
plt.show()
```



In [14]:
```
import matplotlib.pyplot as plt
import numpy as np

x = [1,2,3,4,5,6]
y = [2,5,10,17,26,37]
#Draw the points
plt.scatter(x, y, color='r', marker='*')
# Connects points
plt.plot(x, y, color='g', linestyle='-')
plt.show()
```



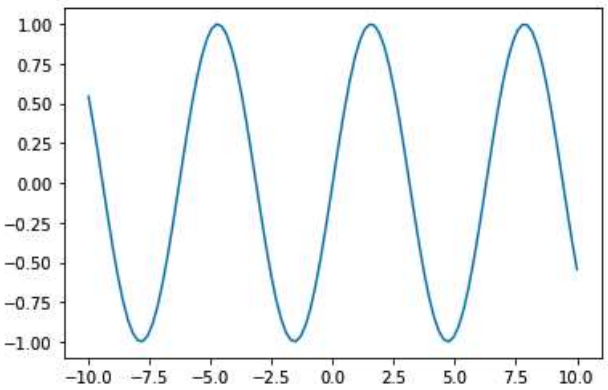
Standard Colors and Markers in Matplotlib

Color Code	Color Name	Marker Code	Marker Type
`b`	Blue	`.`	Point
`g`	Green	`o`	Circle
`r`	Red	`s`	Square
`c`	Cyan	`^`	Triangle (Up)
`m`	Magenta	`v`	Triangle (Down)
`y`	Yellow	`<`	Triangle (Left)
`k`	Black	`>`	Triangle (Right)
`w`	White	`*`	Star
		`+`	Plus
		`x`	Cross
		`D`	Diamond
		`p`	Pentagon
		`h`	Hexagon

Plotting Functions

Functions can be plotted using `matplotlib` by defining an array of x-values and computing corresponding y-values using `numpy`'s `array`

In [15]:
```
x = np.linspace(-10, 10, 100)
y = np.sin(x)

plt.plot(x, y)
plt.show()
```

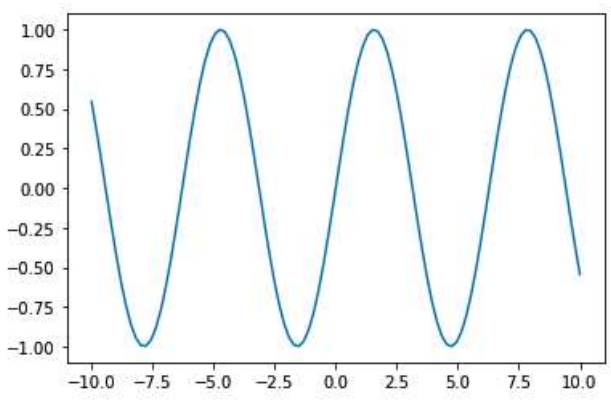


Observations

- `np.linspace(start, stop, num_points)` generates num_points (100 in this case) evenly spaced values between -10 and 10.
- `plt.show()` displays the plotted figure. It is necessary to render the plot when working in scripts or certain environments
- If `y = sin(x)` is sympified (i.e., `y = sp.sin(x)` where `sp` from `sympy`), it remains in symbolic form and won't generate a numerical array. To evaluate it for plotting, use `lambdify`

In [16]:
```
import numpy as np
import sympy as sp
import matplotlib.pyplot as plt

x = sp.symbols('x')
sym_y = sp.sin(x)
lam_y = sp.lambdify(x, sym_y, 'numpy')  # Converts to a NumPy-compatible function

xV = np.linspace(-10, 10, 100)
```

```
yV = lam_y(xV)  # Evaluates at x points

plt.plot(xV, yV)
plt.show()
```



Observations

- `lambdify(variables, expression, modules)` where
  1. `variables`: The symbolic variable(s) that will be replaced with numerical values (e.g., x or (x, y)).
     a. For multiple variables, pass them as a tuple `(x, y)`.
  2. `expression`: The SymPy symbolic expression to be converted into a numerical function.
  3. `modules`: Specifies which numerical library (`numpy`, `math`, `mpmath`, etc.) should be used for evaluation.
     a. Use `numpy` for efficient array-based evaluations.
     b. Use `math` if working with single values.
     c. Use `mpmath` if you need higher precision calculations.
- From now on: `sym_` represents symbolic functions, which remain in an unevaluated mathematical form. `lam_` represents lambdified expressions, which can be evaluated numerically for computations and plotting.

Customization

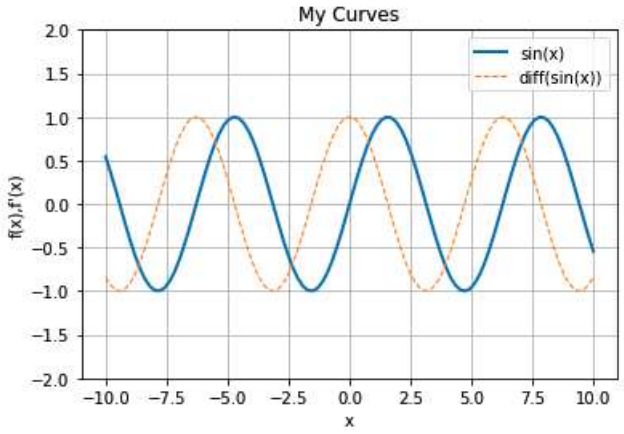
Matplotlib allows customization of plots, including colors, markers, line styles, gridlines, and labels.

Feature	Code / Function	Description
Inside plt.plot()	`plt.plot(x, y)`	Plots y-values against x-values
	`linestyle="dashed"`	Dashed line (`--`)
	`linestyle="solid"`	Solid line (`-`)
	`linestyle="dotted"`	Dotted line (`:`)
	`linestyle="dashdot"`	Dash-dot line (`-.`)
	`linewidth=2`	Adjusts line thickness
	`alpha=0.5`	Sets transparency (0 = transparent, 1 = solid)
Label	`label="My Curve"`	Adds label for legend
Plot Functions	`plt.grid()`	Enables grid lines
	`plt.legend()`	Displays legend if `label` is used
	`plt.xlabel("X-axis")`	Sets x-axis label
	`plt.ylabel("Y-axis")`	Sets y-axis label
	`plt.title("Title")`	Sets plot title
	`plt.xlim(a, b)`	Sets x-axis limits from `a` to `b`
	`plt.ylim(a, b)`	Sets y-axis limits from `a` to `b`
	`plt.axhline(0, color='r', linewidth=2)`	Draws a horizontal line at `y=0` with color and thickness

Example: Plot sin(x) and its derivative graph using all the possible customizations

In [17]:

```
import numpy as np
import sympy as sp
import matplotlib.pyplot as plt

x = sp.symbols('x')
sym_y1 = sp.sin(x)
lam_y1 = sp.lambdify(x, sym_y1, 'numpy')  # Converts to a NumPy-compatible function

sym_y2 = sp.diff(sym_y1,x)
lam_y2 = sp.lambdify(x, sym_y2, 'numpy')  # Converts to a NumPy-compatible function

xV = np.linspace(-10, 10, 100)
y1V = lam_y1(xV)  # Evaluates at x points
y2V = lam_y2(xV)  # Evaluates at x points

plt.plot(xV, y1V, label="sin(x)",linestyle="solid",linewidth=2)
plt.plot(xV, y2V, label="diff(sin(x))",linestyle="dashed",linewidth=1)

plt.grid()
plt.legend()
plt.ylim(-2, 2)
plt.xlabel("x")
plt.ylabel("f(x),f'(x)")
plt.title("My Curves")

plt.show()
```



Lab 03: Taylor Series Expansion for Single-Variable Functions

Level 01 - Values Comparision

In [21]:

```
import sympy as sp
import math
import numpy as np
import matplotlib.pyplot as plt

x=sp.Symbol('x')

#Get f(x),n,a
fx=input("Enter the function: ")
a=sp.sympify(input("Enter the point:")) #Converting to float will not work for pi, so sympify
n=list(map(int,input("Enter the 3 different orders(>2):").split()))


#Call taylor_series function
series_fx1=sp.series(fx,x,a,n[0]).removeO()
series_fx2=sp.series(fx,x,a,n[1]).removeO()
```

```
series_fx3=sp.series(fx,x,a,n[2]).removeO()

#Print All the expansions
print(f"\nTayler series expansion of order {n[0]}:")
print(series_fx1)
print(f"Tayler series expansion of order {n[1]}:")
print(series_fx2)
print(f"Tayler series expansion of order {n[2]}:")
print(series_fx3)

#Lambdify the functions
original_fx=sp.lambdify(x,fx,"numpy")
lam_fx1=sp.lambdify(x,series_fx1,"numpy")
lam_fx2=sp.lambdify(x,series_fx2,"numpy")
lam_fx3=sp.lambdify(x,series_fx3,"numpy")

x0=float(sp.sympify((input("Enter the point x0 to find f(x0): "))).evalf())
print(f"\nExact value of f(x) at {x0:.4f}: {original_fx(x0):.4f}")
print(f"Approximation of f(x) of order {n[0]} at {x0:.4f}: {lam_fx1(x0):.4f}")
print(f"Approximation of f(x) of order {n[1]} at {x0:.4f}: {lam_fx2(x0):.4f}")
print(f"Approximation of f(x) of order {n[2]} at {x0:.4f}: {lam_fx3(x0):.4f}")
```

```
Enter the function: sin(x)
Enter the point:3*pi/4
Enter the 3 different orders(>2):3 5 7

Tayler series expansion of order 3:
-sqrt(2)*(x - 3*pi/4)**2/4 - sqrt(2)*(x - 3*pi/4)/2 + sqrt(2)/2
Tayler series expansion of order 5:
sqrt(2)*(x - 3*pi/4)**4/48 + sqrt(2)*(x - 3*pi/4)**3/12 - sqrt(2)*(x - 3*pi/4)**2/4 - sqrt(2)*(x - 3*pi/4)/2 + sqr
t(2)/2
Tayler series expansion of order 7:
-sqrt(2)*(x - 3*pi/4)**6/1440 - sqrt(2)*(x - 3*pi/4)**5/240 + sqrt(2)*(x - 3*pi/4)**4/48 + sqrt(2)*(x - 3*pi/4)**
3/12 - sqrt(2)*(x - 3*pi/4)**2/4 - sqrt(2)*(x - 3*pi/4)/2 + sqrt(2)/2
Enter the point x0 to find f(x0): pi/6

Exact value of f(x) at 0.5236: 0.5000
Approximation of f(x) of order 3 at 0.5236: 0.8156
Approximation of f(x) of order 5 at 0.5236: 0.4226
Approximation of f(x) of order 7 at 0.5236: 0.5071
```

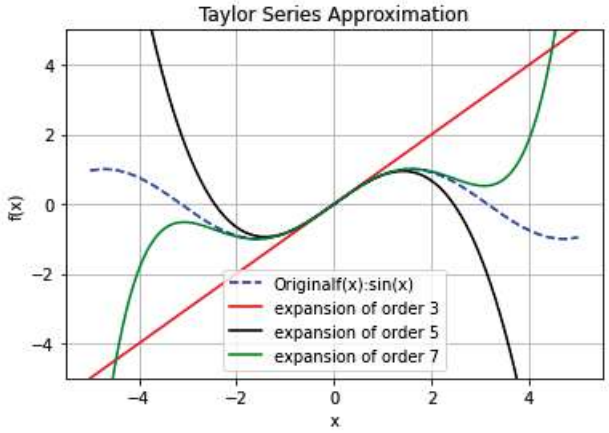
Level 02: Graph Comparision

In [22]:

```
import sympy as sp
import math
import numpy as np
import matplotlib.pyplot as plt

x=sp.Symbol('x')

#Get f(x),n,a
fx=input("Enter the function: ")
a=sp.sympify(input("Enter the point:"))
n=list(map(int,input("Enter the 3 different orders(>2):").split()))

#Call taylor_series function
series_fx1=sp.series(fx,x,a,n[0]).removeO()
series_fx2=sp.series(fx,x,a,n[1]).removeO()
series_fx3=sp.series(fx,x,a,n[2]).removeO()

#Print All the expansions
print(f"Tayler series expansion of order {n[0]}:")
print(series_fx1)
print(f"Tayler series expansion of order {n[1]}:")
print(series_fx2)
print(f"Tayler series expansion of order {n[2]}:")
print(series_fx3)

#Plotting
#Get the range of x and create x Array
lL=float(input("Enter the lower limit of range: "))
uL=float(input("Enter the upper limit of range: "))
xV=np.linspace(lL, uL,200)
```

```
#Lambdify the functions
original_fx=sp.lambdify(x,fx,"numpy")
lam_fx1=sp.lambdify(x,series_fx1,"numpy")
lam_fx2=sp.lambdify(x,series_fx2,"numpy")
lam_fx3=sp.lambdify(x,series_fx3,"numpy")

#Plot all the expansions
plt.plot(xV,original_fx(xV),'blue',label=f"Originalf(x):{fx}",linestyle="dashed")
plt.plot(xV,lam_fx1(xV),'red',label=f"expansion of order {n[0]}")
plt.plot(xV,lam_fx2(xV),'black',label=f"expansion of order {n[1]}")
plt.plot(xV,lam_fx3(xV),'green',label=f"expansion of order {n[2]}")

#Customization
plt.ylim(min(original_fx(xV)) - 4, max(original_fx(xV)) + 4) #Optional to set the y range
plt.xlabel("x")
plt.ylabel("f(x)")
plt.title("Taylor Series Approximation")
plt.legend()
plt.grid()
plt.show()
```

```
Enter the function: sin(x)
Enter the point:0
Enter the 3 different orders(>2):3 5 7
Tayler series expansion of order 3:
x
Tayler series expansion of order 5:
-x**3/6 + x
Tayler series expansion of order 7:
x**5/120 - x**3/6 + x
Enter the lower limit of range: -5
Enter the upper limit of range: 5
```



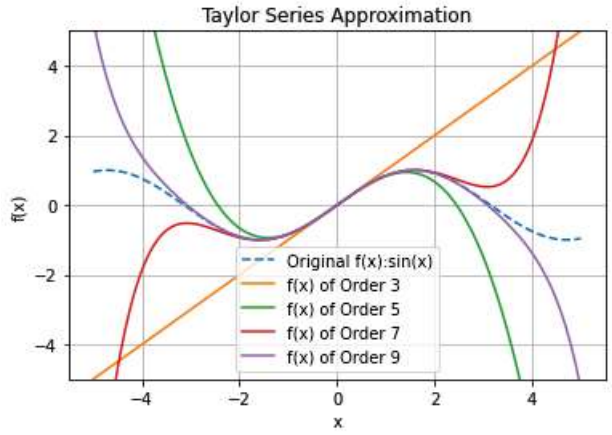
Level 03: Generalized with Convergence analysis

In [23]:

```
import sympy as sp
import math
import numpy as np
import matplotlib.pyplot as plt

#Defining symbolic variable
x=sp.Symbol('x')

#Get f(x), a and different orders
fx=input("Enter the function: ")
a=sp.sympify(input("Enter the point: "))
order_list=list(map(int,input("Enter the order list(>2 & space separated):").split()))
print("\n")

#find the series expansion for all the given orders
#(Not works for 1, since plotting constant function is different)
expanded_fxs=[]
expanded_fxs=[sp.series(fx,x,a,i).removeO() for i in order_list]

#Print all the expansions
for order, expansions in zip(order_list,expanded_fxs):
```

```
    print(f"\nTaylor series expansion upto order {order}:")
    print(expansions)

#Plotting
#Get the plotting range
lL=float(input("Enter the lower limit of range: "))
uL=float(input("Enter the upper limit of range: "))


#To plot get the numeric values of x and f(x)
xArray=np.linspace(lL, uL,200)
lambdified_fxs=[sp.lambdify(x,expr,"numpy") for expr in expanded_fxs]
original_fx=sp.lambdify(x,fx,"numpy")

#convergence Analysis
print("\nConvergence Analysis:")
for i,expr in enumerate(lambdified_fxs):
    for j in xArray:
        err=abs(expr(j)-original_fx(j))
        if err<0.1:
            roC=abs(j-a)
            cLL=a-roC
            cUL=a+roC
            print(f"Series of order {order_list[i]} is converged in the range ({cLL},{cUL})")
            break


#Plot all the graphs
plt.plot(xArray,original_fx(xArray),label=f"Original f(x):{fx}",linestyle="dashed")
for i, order in enumerate(order_list):
    plt.plot(xArray,lambdified_fxs[i](xArray),label=f"f(x) of Order {order}")
#Customization
plt.ylim(min(original_fx(xArray)) - 4, max(original_fx(xArray)) + 4) #Optional to set the y range
plt.xlabel("x")
plt.ylabel("f(x)")
plt.title("Taylor Series Approximation")
plt.legend()
plt.grid()
plt.show()
```

```
Enter the function: sin(x)
Enter the point: 0
Enter the order list(>2 & space separated):3 5 7 9



Taylor series expansion upto order 3:
x

Taylor series expansion upto order 5:
-x**3/6 + x

Taylor series expansion upto order 7:
x**5/120 - x**3/6 + x

Taylor series expansion upto order 9:
-x**7/5040 + x**5/120 - x**3/6 + x
Enter the lower limit of range: -5
Enter the upper limit of range: 5

Convergence Analysis:
Series of order 3 is converged in the range (-0.829145728643216,0.829145728643216)
Series of order 5 is converged in the range (-1.63316582914573,1.63316582914573)
Series of order 7 is converged in the range (-2.43718592964824,2.43718592964824)
Series of order 9 is converged in the range (-3.24120603015075,3.24120603015075)
```



Exercise

Find the Taylor's Series expansion of different orders > 2 for the following about the given point and compare the convergence by evaluating the function and its approximations at a given point

1. $\sqrt{1+\sin(x)}$ about the point $a = 0$ at $x = 1$
2. $\cos(x)$ about $x = \pi/3$ at $x = \pi/4$
3. $\sin^{-1}(x)$ about $x = 0$ at $x = 1$
4. $\log(x)$ about $x = 1$ at $x = 2$
5. e^{x^2} about $x = 0$ at $x = 1$

Try: Plot the approximations and compare the convergence of different orders

---

Lab 04: Newton-Raphson Method for Single-Variable Functions

Level 01: Without Graph

In [24]:

```
import sympy as sp
import numpy as np

# Define symbolic variable
x = sp.Symbol('x')

# Read function and compute derivative
f = sp.sympify(input("Enter the function f(x): "))
fx = sp.diff(f, x)

# Convert to numerical functions for evaluation
lam_f = sp.lambdify(x, f, "numpy")
lam_fx = sp.lambdify(x, fx, "numpy")

# Get initial range
x0 = float(sp.sympify(input("Enter the initial Value x0: ")).evalf())

flag,i=0,1
x_roots=[x0]
while flag==0:
    x1 = x0 - lam_f(x0)/lam_fx(x0)
    x_roots.append(x1)
    print(f"Iteration {i}:- Approximate Root is {x1}")
    if abs(x_roots[i]-x_roots[i-1])<1e-5:
        print("-----------------------------------------")
        print(f"The root of the given equation : {x1}")
        flag = 1
    else:
```

```
        x0=x1
        i+=1
```

```
Enter the function f(x): 2*tan(x)-3*x
Enter the initial Value x0: 1.5
Iteration 1:- Approximate Root is 1.440249975867342
Iteration 2:- Approximate Root is 1.34537949685672
Iteration 3:- Approximate Root is 1.218858943719272
Iteration 4:- Approximate Root is 1.0894616009245244
Iteration 5:- Approximate Root is 1.0008893469514206
Iteration 6:- Approximate Root is 0.9703345803666928
Iteration 7:- Approximate Root is 0.9674266073304036
Iteration 8:- Approximate Root is 0.967402639787109
Iteration 9:- Approximate Root is 0.9674026381746997
----------------------------------------
The root of the given equation : 0.9674026381746997
```

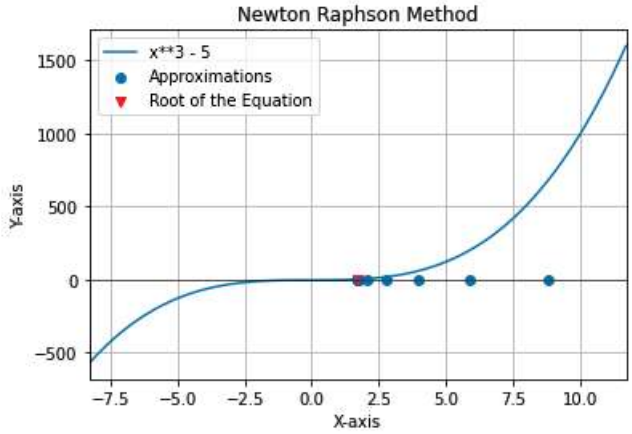
Level 02: With Graph

In [25]:

```
import sympy as sp
import numpy as np
import matplotlib.pyplot as plt

# Define symbolic variable
x = sp.Symbol('x')

# Read function and compute derivative
f = sp.sympify(input("Enter the function f(x): "))
fx = sp.diff(f, x)

# Convert to numerical functions for evaluation
lam_f = sp.lambdify(x, f, "numpy")
lam_fx = sp.lambdify(x, fx, "numpy")

# Get initial range
x0 = float(input("Enter the initial Value x0: "))

flag,i=0,1
x_roots=[x0]
while flag==0:
    x1 = x0 - lam_f(x0)/lam_fx(x0)
    x_roots.append(x1)
    print(f"Iteration {i}:- Approximate Root is {x1:.4f}")
    if abs(x_roots[i]-x_roots[i-1])<1e-5:
        print("----------------------------------------")
        print(f"The root of the given equation : {x1:.4f}")
        flag = 1
    else:
        x0=x1
        i+=1
#Plotting
xV=np.linspace(x1-10,x1+10,200)
plt.plot(xV,lam_f(xV),label=f"{f}")
plt.scatter(x_roots, [0]*len(x_roots), marker='o', label="Approximations")
plt.scatter(x1, 0,color='red', marker='v', label="Root of the Equation")
#Customization
plt.axhline(0, color='black', linewidth=0.5)  # Draw x-axis
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Newton Raphson Method")
plt.xlim(x1-10, x1+10)
plt.legend()
plt.grid(True)
plt.show()
```

```
Enter the function f(x): x**3-5
Enter the initial Value x0: 100
Iteration 1:- Approximate Root is 66.6668
Iteration 2:- Approximate Root is 44.4449
Iteration 3:- Approximate Root is 29.6308
Iteration 4:- Approximate Root is 19.7558
Iteration 5:- Approximate Root is 13.1748
Iteration 6:- Approximate Root is 8.7928
Iteration 7:- Approximate Root is 5.8834
```

```
Iteration 8:- Approximate Root is 3.9704
Iteration 9:- Approximate Root is 2.7527
Iteration 10:- Approximate Root is 2.0551
Iteration 11:- Approximate Root is 1.7647
Iteration 12:- Approximate Root is 1.7117
Iteration 13:- Approximate Root is 1.7100
Iteration 14:- Approximate Root is 1.7100
----------------------------------------
The root of the given equation : 1.7100
```



Exercise: Using Newton's Raphson method evaluate one of the real roots for the following equations:

1. $x^4 - 12x + 7, \ x_0 = 0$
2. $x + ln(x) = 3.375, \ x_0 = 1$
3. $2\tan(x) = 3x, \ x_0 = 1$
4. $x^3 - 5 = 0, \ x_0 = 100$

Lab 05: Taylor Series Expansion for Two-Variable Functions

In [26]:

```
import sympy as sp
import math

def taylor2(f,a,b):
    f=sp.sympify(f)
    a=float(a)
    b=float(b)
    fx=sp.lambdify([x,y],sp.diff(f,x),"numpy")
    fy=sp.lambdify([x,y],sp.diff(f,y),"numpy")
    fxx=sp.lambdify([x,y],sp.diff(f,x,2),"numpy")
    fxy=sp.lambdify([x,y],sp.diff(f,x,y),"numpy")
    fyy=sp.lambdify([x,y],sp.diff(f,y,2),"numpy")
    f=sp.lambdify([x,y],f,"numpy")
    return f(a,b)+(x-a)*fx(a,b)+(y-b)*fy(a,b)+(1/math.factorial(2))*((x-a)**2*fxx(a,b)+2*(x-a)*(y-b)*fxy(a,b)+(y-


x, y = sp.symbols('x y')
f=input("Enter the function f(x,y): ")
a,b=input("Enter the point a and b (space separated): ").split()
a=float(sp.sympify((a)).evalf())
b=float(sp.sympify((b)).evalf())
print(f"\nTaylor's Series expansion of {f} about the point ({a},{b}) up to 2nd order:")
print(taylor2(f,a,b))
```

```
Enter the function f(x,y): y*exp(x-1)-x*log(y)
Enter the point a and b (space separated): 1 1

Taylor's Series expansion of y*exp(x-1)-x*log(y) about the point (1.0,1.0) up to 2nd order:
1.0*x + 0.5*(x - 1.0)**2 + 0.5*(y - 1.0)**2
```

Exercise: Expand the following functions in powers of x and y up to second-degree terms

1. $\sin(x)\sin(y)$
2. $e^x \sin(y)$
3. $e^{x^2-y^2}$ **Exercise:** Expand the following functions about the mentioned points up to second-degree terms
4. $xy^2 + \cos(y)$ about $(1/\pi/2)$
5. $x^2y + 3y - 2$ about $(1, -2)$
6. x^y about $(1, 1)$

Lab 06: Newton-Raphson Method for Two-Variable Functions

In [27]:

```
import sympy as sp

# Define symbols
x, y, h, k = sp.symbols('x y h k')

# User input for functions
f = sp.sympify(input("Enter f(x, y): "))
g = sp.sympify(input("Enter g(x, y): "))

# Compute partial derivatives
dfx = sp.diff(f, x)
dfy = sp.diff(f, y)
dgx = sp.diff(g, x)
dgy = sp.diff(g, y)

# Initial guesses
x0,y0 = input("Enter the initial guess (x0,y0) (space separated): ").split()
x0,y0=float(x0),float(y0)

flag = 0
iteration = 0

while flag==0:
    # Construct equations
    EQ1 = dfx.subs({x: x0, y: y0}) * h + dfy.subs({x: x0, y: y0}) * k + f.subs({x: x0, y: y0})
    EQ2 = dgx.subs({x: x0, y: y0}) * h + dgy.subs({x: x0, y: y0}) * k + g.subs({x: x0, y: y0})

    # Solve for h and k
    solution = sp.solve([EQ1, EQ2], (h, k))

    # Update x and y values
    xn = x0 + solution[h]
    yn = y0 + solution[k]
    iteration += 1

    print(f"Iteration {iteration}: ({xn:.6f}, {yn:.6f})")

    # Check convergence
    if abs(f.subs({x: xn, y: yn})) <= 1e-5:
        flag = 1
    else:
        x0, y0 = xn, yn
```

```
Enter f(x, y): x**2-y**2-4
Enter g(x, y): y**2+x**2-16
Enter the initial guess (x0,y0) (space separated): 1 1
Iteration 1: (5.500000, 3.500000)
Iteration 2: (3.659091, 2.607143)
Iteration 3: (3.196005, 2.454256)
Iteration 4: (3.162456, 2.449494)
Iteration 5: (3.162278, 2.449490)
```

Exercise: Using Newton – Raphson iterative method, find a real root of the following equations:

1. $x^2 + y = 11; y^2 + x = 7, (x_0, y_0) = (3.5, -1.8)$
2. $x^3 = y + 100; y^3 = x + 150, (x_0, y_0) = (4.5, 5.3)$

Lab 07: Finding Maxima and Minima of Two-Variable Functions

Level 01: Without Graph

In [28]:

```
import sympy as sp
import numpy as np
import matplotlib.pyplot as plt

# Define symbolic variables
x, y = sp.symbols('x y')

# Read function and sympify
f = sp.sympify(input("Enter the function f(x, y): "))
# Calculate first derivatives
fx = sp.diff(f, x)
fy = sp.diff(f, y)

# Solve for stationary points
solutions = sp.solve([fx, fy], (x, y), dict=True)
# Get only real solutions
real_solutions = []
for sol in solutions:
    xi, yi = sol[x], sol[y]  # Extract x and y values
    if xi.is_real and yi.is_real:  # Check if both x & y are real
        real_solutions.append(sol)
# Compute second-order derivatives
fxx = sp.diff(fx, x)  #f_xx
fyy = sp.diff(fy, y)  #f_yy
fxy = sp.diff(fx, y)  #f_xy

# Check each stationary point
for sol in real_solutions :
    xi, yi = sol[x], sol[y]
        # Evaluate second-order derivatives at (xi, yi)
    A = fxx.subs(sol)
    B = fxy.subs(sol)
    C = fyy.subs(sol)
    # Check the condition
    Cond = A * C - B**2
    # Function value at (xi, yi)
    f_val = f.subs(sol)
    # Check for maxima, minima, or saddle point
    if Cond > 0 and A < 0:
        print(f"({xi}, {yi}) -> Maxima: The maximum value is {f_val}")
        print("\n")
    elif Cond > 0 and A > 0:
        print(f"({xi}, {yi}) -> Minima: The minimum value is {f_val}")
        print("\n")
    elif Cond < 0:
        print(f"({xi}, {yi}) -> Saddle point")
        print("\n")
    else:
        print(f"No conclusion can be drawn for ({xi}, {yi})")
        print("\n")
```

```
Enter the function f(x, y): x**3+y**3-3*x-12*y+20
(-1, -2) -> Maxima: The maximum value is 38


(-1, 2) -> Saddle point


(1, -2) -> Saddle point


(1, 2) -> Minima: The minimum value is 2
```

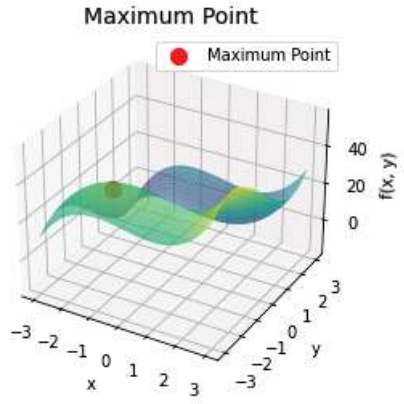
Level 02: With Graph

In [29]:

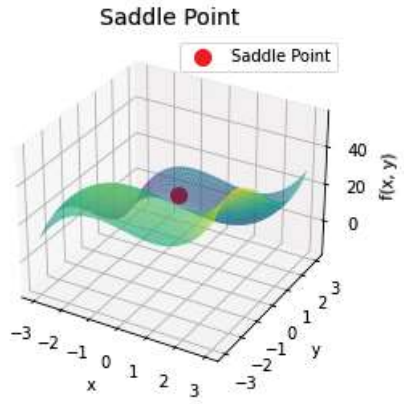
```python
import sympy as sp
import numpy as np
import matplotlib.pyplot as plt


# Function to plot f(x, y)
def plot_function(f, xi, yi, f_val, plot_title):
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')

    # Define plotting range
    X = np.linspace(-3, 3, 50)
    Y = np.linspace(-3, 3, 50)
    X, Y = np.meshgrid(X, Y)

    # Convert f(x, y) to a numerical function
    f_numeric = sp.lambdify((x, y), f, "numpy")
    Z = f_numeric(X, Y)

    # Plot surface
    ax.plot_surface(X, Y, Z, cmap="viridis", alpha=0.7)

    # Plot stationary point
    ax.scatter(float(xi), float(yi), float(f_val), color='r', s=100, label=plot_title)

    ax.set_title(plot_title, fontsize=14)
    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.set_zlabel("f(x, y)")
    ax.legend()
    plt.show()



# Define symbolic variables
x, y = sp.symbols('x y')

# Read function and sympify
f = sp.sympify(input("Enter the function f(x, y): "))
# Calculate first derivatives
fx = sp.diff(f, x)
fy = sp.diff(f, y)

# Solve for stationary points
solutions = sp.solve([fx, fy], (x, y), dict=True)
# Get only real solutions
real_solutions = []
for sol in solutions:
    xi, yi = sol[x], sol[y]  # Extract x and y values
    if xi.is_real and yi.is_real:  # Check if both x & y are real
        real_solutions.append(sol)
# Compute second-order derivatives
fxx = sp.diff(fx, x)  #f_xx
fyy = sp.diff(fy, y)  #f_yy
fxy = sp.diff(fx, y)  #f_xy

# Check each stationary point
```

```python
for sol in real_solutions :
    xi, yi = sol[x], sol[y]

    # Evaluate second-order derivatives at (xi, yi)
    A = fxx.subs(sol)
    B = fxy.subs(sol)
    C = fyy.subs(sol)

    # Check the condition
    Cond = A * C - B**2

    # Function value at (xi, yi)
    f_val = f.subs(sol)

    # Check for maxima, minima, or saddle point
    if Cond > 0 and A < 0:
        print(f"({xi}, {yi}) -> Maxima: The maximum value is {f_val}")
        plot_function(f, xi, yi, f_val, "Maximum Point")
        print("\n")
    elif Cond > 0 and A > 0:
        print(f"({xi}, {yi}) -> Minima: The minimum value is {f_val}")
        plot_function(f, xi, yi, f_val, "Minimum Point")
        print("\n")
    elif Cond < 0:
        print(f"({xi}, {yi}) -> Saddle point")
        plot_function(f, xi, yi, f_val, "Saddle Point")
        print("\n")
    else:
        print(f"No conclusion can be drawn for ({xi}, {yi})")
        print("\n")
```

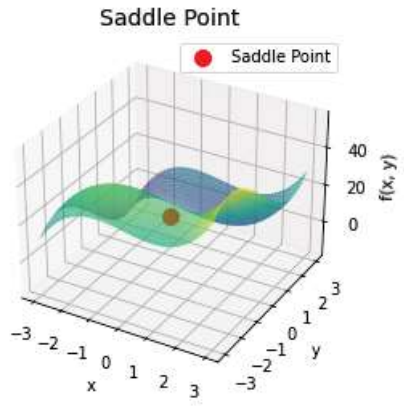
```
Enter the function f(x, y): x**3+y**3-3*x-12*y+20
(-1, -2) -> Maxima: The maximum value is 38
```



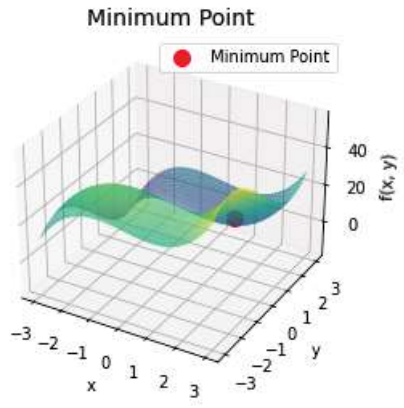

```
(-1, 2) -> Saddle point
```




```
(1, -2) -> Saddle point
```




```
(1, 2) -> Minima: The minimum value is 2
```



Exercise: Discuss the maxima and minima of the following functions:

1. $x^3 * y^2 * (1 - x - y)$
2. $x^3 + y^3 - 3xy$
3. $xy(1 - x - y)$

Lab 08: Solving ODEs using Euler's and Modified Euler's Method

Euler's Method

In [30]:

```python
import numpy as np
import sympy as sp

x, y = sp.symbols('x y')

f = sp.sympify(input("Enter f(x,y): "))
lam_f = sp.lambdify((x, y), f, 'numpy')

x0 = float(input("Enter the initial value of x: "))
y0 = float(input("Enter the initial value of y: "))
xn = float(input("Enter the value of x for the desired solution: "))
h = float(input("Enter the step size h: "))

xV = np.arange(x0, xn + h, h)
yV = np.zeros_like(xV)  #zeros_like(Array_V) creates a zero array of same dimension of Array_V
yV[0] = y0

for i in range(1, len(xV)):
```

```
    yV[i] = yV[i-1] + h * lam_f(xV[i-1], yV[i-1])
    print(f'y({xV[i]:.3f}) = {yV[i]:.3f}')
```

```
Enter f(x,y): x+y
Enter the initial value of x: 0
Enter the initial value of y: 1
Enter the value of x for the desired solution: 10
Enter the step size h: 1
y(1.000) = 2.000
y(2.000) = 5.000
y(3.000) = 12.000
y(4.000) = 27.000
y(5.000) = 58.000
y(6.000) = 121.000
y(7.000) = 248.000
y(8.000) = 503.000
y(9.000) = 1014.000
y(10.000) = 2037.000
```

Modified Euler's Method

In [31]:

```
import numpy as np
import sympy as sp

# Define symbols
x, y = sp.symbols('x y')

# Input function f(x, y)
sym_f = sp.sympify(input('Enter f(x,y): '))
f = sp.lambdify((x, y), sym_f, 'numpy')

# Input initial conditions
x0 = float(input('Enter the initial value of x: '))
y0 = float(input('Enter the initial value of y: '))
xn = float(input('Enter the value of x for the desired solution: '))
h = float(input('Enter the step size h: '))

# Define the range of x values
xV = np.arange(x0, xn + h, h)
yc = np.zeros(len(xV))
y = np.zeros((len(xV), 2000))
yc[0] = y0
y[0, 0] = y0

# Implement Euler's modified method
for i in range(1, len(xV)):
    j = 0
    # Prediction using Euler's Method
    y[i, j] = yc[i - 1] + h * f(xV[i - 1], yc[i - 1])
    print(f'\nPredicted value y({xV[i]:.4f})={y[i, j]:.4f}')

    # Correction using Modified Euler's Method
    converged = False
    while not converged:
        j += 1
        y[i, j] = yc[i - 1] + (h / 2) * (f(xV[i - 1], yc[i - 1]) + f(xV[i], y[i, j - 1]))
        print(f'Corrected value {j}: y({xV[i]:.4f})={y[i, j]:.4f}')

        if abs(y[i, j - 1] - y[i, j]) < 0.0001:  # Convergence check
            yc[i] = y[i, j]
            print(f'y({xV[i]:.4f})={yc[i]:.4f}\n')
            converged = True
```

```
Enter f(x,y): x+y
Enter the initial value of x: 0
Enter the initial value of y: 1
Enter the value of x for the desired solution: 0.4
Enter the step size h: 0.1

Predicted value y(0.1000)=1.1000
Corrected value 1: y(0.1000)=1.1100
Corrected value 2: y(0.1000)=1.1105
Corrected value 3: y(0.1000)=1.1105
```

```
y(0.1000)=1.1105


Predicted value y(0.2000)=1.2316
Corrected value 1: y(0.2000)=1.2426
Corrected value 2: y(0.2000)=1.2432
Corrected value 3: y(0.2000)=1.2432
y(0.2000)=1.2432


Predicted value y(0.3000)=1.3875
Corrected value 1: y(0.3000)=1.3997
Corrected value 2: y(0.3000)=1.4004
Corrected value 3: y(0.3000)=1.4004
y(0.3000)=1.4004


Predicted value y(0.4000)=1.5704
Corrected value 1: y(0.4000)=1.5839
Corrected value 2: y(0.4000)=1.5846
Corrected value 3: y(0.4000)=1.5846
y(0.4000)=1.5846
```

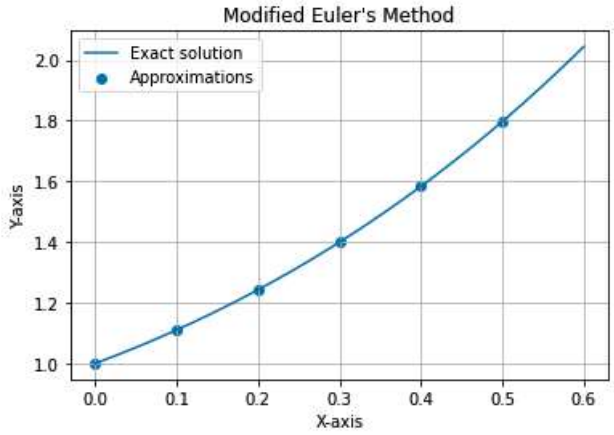
Level 2: With graph

In [32]:

```python
import sympy as sp
import numpy as np
import matplotlib.pyplot as plt

# Take user input
x, y = sp.symbols('x y')
str_f = input("Enter f(x,y) (e.g., x + y): ")
#f = sp.sympify(f_input)
x0 = float(input("Enter initial x0: "))
y0 = float(input("Enter initial y0: "))
xn = float(input("Enter final xn: "))
h = float(input("Enter step size h: "))

# Input function f(x, y)
f = sp.lambdify((x, y), str_f, 'numpy')

# Define the range of x values
xV = np.arange(x0, xn + h, h)
yc = np.zeros(len(xV))
y = np.zeros((len(xV), 2000))
yc[0] = y0
y[0, 0] = y0

# Implement Euler's modified method
for i in range(1, len(xV)):
    j = 0
    # Prediction using Euler's Method
    y[i, j] = yc[i - 1] + h * f(xV[i - 1], yc[i - 1])
    print(f'\nPredicted value y({xV[i]:.4f})={y[i, j]:.4f}')

    # Correction using Modified Euler's Method
    converged = False
    while not converged:
        j += 1
        y[i, j] = yc[i - 1] + (h / 2) * (f(xV[i - 1], yc[i - 1]) + f(xV[i], y[i, j - 1]))
        print(f'Corrected value {j}: y({xV[i]:.4f})={y[i, j]:.4f}')

        if abs(y[i, j - 1] - y[i, j]) < 0.001:  # Convergence check
            yc[i] = y[i, j]
            print(f'y({xV[i]:.4f})={yc[i]:.4f}\n')
            converged = True


#Plotting
y_str = sp.sympify(input("Enter the exact solution to compare: "))
lam_y = sp.lambdify(x,y_str, 'numpy')
```

```
xArray = np.linspace(x0, xn+h, 200) #To plot exact sol
plt.plot(xArray,lam_y(xArray),label="Exact solution")
plt.scatter(xV,yc , marker='o', label="Approximations")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Modified Euler's Method")
plt.legend()
plt.grid(True)
plt.show()
```

```
Enter f(x,y) (e.g., x + y): x+y
Enter initial x0: 0
Enter initial y0: 1
Enter final xn: 0.5
Enter step size h: 0.1

Predicted value y(0.1000)=1.1000
Corrected value 1: y(0.1000)=1.1100
Corrected value 2: y(0.1000)=1.1105
y(0.1000)=1.1105


Predicted value y(0.2000)=1.2316
Corrected value 1: y(0.2000)=1.2426
Corrected value 2: y(0.2000)=1.2432
y(0.2000)=1.2432


Predicted value y(0.3000)=1.3875
Corrected value 1: y(0.3000)=1.3997
Corrected value 2: y(0.3000)=1.4003
y(0.3000)=1.4003


Predicted value y(0.4000)=1.5703
Corrected value 1: y(0.4000)=1.5838
Corrected value 2: y(0.4000)=1.5845
y(0.4000)=1.5845


Predicted value y(0.5000)=1.7830
Corrected value 1: y(0.5000)=1.7979
Corrected value 2: y(0.5000)=1.7986
y(0.5000)=1.7986

Enter the exact solution to compare: 2*exp(x)-x-1
```



Exercise: Solve the following initial value problems using Euler's method

1. $\frac{dy}{dx} = 2 - 2y - e^{-4x}; y(0) = 1$ at $x = 5$ by taking $h = 0.5$
2. $e^y \frac{dy}{dx} + x^2 y^2 = 2\sin(3x); y(0) = 5$ at $x = 0.5$ by taking $h = 0.1$

Exercise: Solve the following initial value problems using Modified Euler's method

1. $\frac{dy}{dx} = \frac{2y}{x} + x^3; y(1) = 0.5$ at $x = 1.4$ by taking $h = 0.2$
2. $\frac{dy}{dx} = x + y^2; y(0) = 1$ at $x = 0.2$ by taking $h = 0.1$

Lab 09: Solving ODEs using the Runge-Kutta 4th Order Method

Level 1: Without Graph

In [33]:

```
import sympy as sp

x, y = sp.symbols('x y')

# Take user input
str_f = input("Enter f(x,y) (e.g., x + y): ")
x0 = float(input("Enter initial x0: "))
y0 = float(input("Enter initial y0: "))
xn = float(input("Enter final xn: "))
h = float(input("Enter step size h: "))

f = sp.lambdify((x, y), str_f, 'math')  # Convert string to numeric

step = 0

while x0 < xn:
    k1 = h * f(x0, y0)
    k2 = h * f(x0 + h/2, y0 + k1/2)
    k3 = h * f(x0 + h/2, y0 + k2/2)
    k4 = h * f(x0 + h, y0 + k3)

    y1 = y0 + (k1 + 2*k2 + 2*k3 + k4) / 6
    x1 = x0 + h

    print(f"\nStep : {step+1}")
    print(f"x0: {x0:.4f}, y0: {y0:.4f}")
    print(f"k1: {k1:.4f}, k2: {k2:.4f}, k3: {k3:.4f}, k4: {k4:.4f}")
    print(f"Solution y({x1:.4f}) = {y1:.4f}\n")
    x0, y0 = x1, y1
    step += 1
```

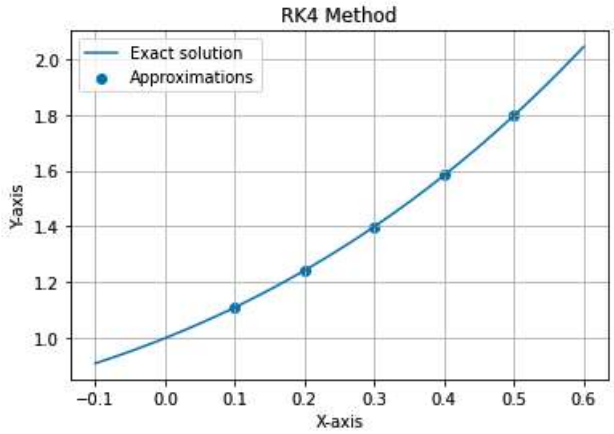
```
Enter f(x,y) (e.g., x + y): x+y
Enter initial x0: 0
Enter initial y0: 1
Enter final xn: 0.4
Enter step size h: 0.1

Step : 1
x0: 0.0000, y0: 1.0000
k1: 0.1000, k2: 0.1100, k3: 0.1105, k4: 0.1211
Solution y(0.1000) = 1.1103


Step : 2
x0: 0.1000, y0: 1.1103
k1: 0.1210, k2: 0.1321, k3: 0.1326, k4: 0.1443
Solution y(0.2000) = 1.2428


Step : 3
x0: 0.2000, y0: 1.2428
k1: 0.1443, k2: 0.1565, k3: 0.1571, k4: 0.1700
Solution y(0.3000) = 1.3997


Step : 4
x0: 0.3000, y0: 1.3997
k1: 0.1700, k2: 0.1835, k3: 0.1841, k4: 0.1984
Solution y(0.4000) = 1.5836
```

In [35]:

```
import sympy as sp
import numpy as np
import matplotlib.pyplot as plt

# Take user input
x, y = sp.symbols('x y')
str_f = input("Enter f(x,y) (e.g., x + y): ")
#f = sp.sympify(f_input)
x0 = float(input("Enter initial x0: "))
y0 = float(input("Enter initial y0: "))
xn = float(input("Enter final xn: "))
h = float(input("Enter step size h: "))

xV = np.linspace(x0-h, xn+h, 200) #To plot

f = sp.lambdify((x, y), str_f, 'math')  # Convert symbolic function to numeric
x_num = []
y_num = []
step = 0

while x0 < xn:
    k1 = h * f(x0, y0)
    k2 = h * f(x0 + h/2, y0 + k1/2)
    k3 = h * f(x0 + h/2, y0 + k2/2)
    k4 = h * f(x0 + h, y0 + k3)

    y1 = y0 + (k1 + 2*k2 + 2*k3 + k4) / 6
    x1 = x0 + h

    print(f"\nStep : {step+1}")
    print(f"x0: {x0:.4f}, y0: {y0:.4f}")
    print(f"k1: {k1:.4f}, k2: {k2:.4f}, k3: {k3:.4f}, k4: {k4:.4f}")
    print(f"Solution y({x1:.4f}) = {y1:.4f}\n")
    x0, y0 = x1, y1
    x_num.append(x1)
    y_num.append(y1)
    step += 1


#Plotting
y_str = sp.sympify(input("Enter the exact solution to compare: "))
lam_y = sp.lambdify(x,y_str, 'numpy')


plt.plot(xV,lam_y(xV),label=f"Exact solution")
plt.scatter(x_num,y_num , marker='o', label="Approximations")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("RK4 Method")
plt.legend()
plt.grid(True)
plt.show()
```

```
Enter f(x,y) (e.g., x + y): x+y
Enter initial x0: 0
Enter initial y0: 1
Enter final xn: 0.5
Enter step size h: 0.1

Step : 1
x0: 0.0000, y0: 1.0000
k1: 0.1000, k2: 0.1100, k3: 0.1105, k4: 0.1211
Solution y(0.1000) = 1.1103


Step : 2
x0: 0.1000, y0: 1.1103
k1: 0.1210, k2: 0.1321, k3: 0.1326, k4: 0.1443
Solution y(0.2000) = 1.2428


Step : 3
x0: 0.2000, y0: 1.2428
k1: 0.1443, k2: 0.1565, k3: 0.1571, k4: 0.1700
```

```
Solution y(0.3000) = 1.3997


Step : 4
x0: 0.3000, y0: 1.3997
k1: 0.1700, k2: 0.1835, k3: 0.1841, k4: 0.1984
Solution y(0.4000) = 1.5836


Step : 5
x0: 0.4000, y0: 1.5836
k1: 0.1984, k2: 0.2133, k3: 0.2140, k4: 0.2298
Solution y(0.5000) = 1.7974

Enter the exact solution to compare: 2*exp(x)-x-1
```



Exercise: Solve the following initial value problems using RK4 method

1. $\frac{dy}{dx} = \frac{2y}{x} + x^3; y(1) = 0.5$ at $x = 1.4$ by taking $h = 0.2$.(Exact solution is: $y = x^4/2$)
2. $\frac{dy}{dx} = 2 - 2y - e^{-4x}; y(0) = 1$ at $x = 1$ by taking $h = 0.5$. (Exact solution is: $1 - \frac{e^{-2x}+e^{-4x}}{2}$)
3. $\frac{dy}{dx} = \frac{x+3y}{2x}; y(1) = 1$ at $x = 1.3$ by taking $h = 0.1$ (Exact solution is: $2x^{3/2} - x$)

Lab 10: Solving Higher-Order ODEs

Level 1: General Solution

In [36]:

```python
import sympy as sp

# Define symbols
x = sp.symbols('x')
y = sp.Function('y')(x)  # Define y(x) explicitly as a function

# Take differential equation input
eq = sp.sympify(input("Enter the differential equation in terms of y: "), locals={'y': y, 'x': x, 'sp': sp})
# Solve the differential equation
solution = sp.dsolve(eq, y)

# Print the solution
print("\nSolution for the given Differential Equation:")
print(solution)
```

```
Enter the differential equation in terms of y: sp.diff(y,x,2)-y

Solution for the given Differential Equation:
Eq(y(x), C1*exp(-x) + C2*exp(x))
```

Level 2: Initial value/Boundary value Problems

In [37]:

```
import sympy as sp

# Define symbols
x = sp.symbols('x')
y = sp.Function('y')(x)  # Define y(x) explicitly as a function

# Take differential equation input
eq = sp.sympify(input("Enter the differential equation in terms of y: "), locals={'y': y, 'x': x, 'sp': sp})

# Read initial/boundary conditions
ics_count = int(input("Enter the number of initial/boundary conditions: "))
ics = {}

for _ in range(ics_count):
    condition = input("Enter condition (e.g., y.subs(x,0)=1 or y.diff(x).subs(x,0)=2): ")
    key_str, value_str = condition.split("=")  # Split at '='

    # Convert key & value into SymPy expressions, ensuring y(x) is correctly referenced
    key = sp.sympify(key_str.strip(), locals={'y': y, 'x': x, 'sp': sp})
    value = sp.sympify(value_str.strip())

    ics[key] = value  # Store in dictionary

# Solve the differential equation with initial conditions
solution = sp.dsolve(eq, y, ics=ics)

# Print the solution
print("\nSolution for the given Differential Equation:")
print(solution)
```

```
Enter the differential equation in terms of y: sp.diff(y,x,2)-y
Enter the number of initial/boundary conditions: 2
Enter condition (e.g., y.subs(x,0)=1 or y.diff(x).subs(x,0)=2): y.subs(x,0)=1
Enter condition (e.g., y.subs(x,0)=1 or y.diff(x).subs(x,0)=2): y.diff(x).subs(x,0)=2

Solution for the given Differential Equation:
Eq(y(x), 3*exp(x)/2 - exp(-x)/2)
```

Exercise: Solve the initial value/boundary value problems:

1. $(D^2 + D)y = 2 - 2x - x^2, y(0) = 8, y(2) = -1$
2. $y''' - y'' + 100y' - 100y = 0, y(0) = 4, y'(0) = 11, y''(0) = -299$ (Note: for nth order condition: `y.diff(x,n).subs(x,a)=b`)
3. $x^2\frac{d^2y}{dx^2} - 3x\frac{dy}{dx} + 4y = 0, y(1) = 2, y'(1) = 5$

In [ ]: