Doing Maths using SymPy

Dr. Susanta Mandal

Assistant Professor

Department of Mathematics

St. Paul's Cathedral Mission College

Contents

Chapter - 1

Symbolic Math using SymPy

1.1 Introduction

Python is composed of a core language along with an extensive library of additional software packaged in modules. Many of these modules are included with the standard Python distribution, offering extra functionality for handling various system tasks. Others provide more specialized features that might not be relevant to every user. These modules function like a library, allowing us to use them as needed. In Python, *external packages* (also known as *third-party packages* or *libraries*) are collections of pre-written code that we can install and use in our own projects. These packages are not part of Python's standard library, so they need to be installed separately.

1.2 Functions, methods, modules, and packages of python

In Python, modules, methods, functions, and packages serve distinct purposes in programming, though they often work together in code.

Here we will demonstrate briefly their differences, with examples:

(a) Function: A function is a reusable block of code that performs a specific task. Functions are defined using the def keyword. Python functions are extensively discussed in chapter-9. The following table highlights how to define and use functions within a python code.

Function Creation	Use of function in another place

`def greet(name):`
`    return f"Hello, {name}!"`
	`print(greet("Anna"))`
	# Output: Hello, Anna!

(b) Module: A module is a single Python file (with a .py extension) that contains definitions of variables, functions, and classes that can be used in other Python programs. Specially modules help organize code into separate, reusable files.

Module Creation	Importing and use of the module in another file

# create a python file 	
`math_operations.py`
`def user_ad(a, b):`
`    return a + b`
`def user_subt(a,b):`
`  return a - b`
	`import math_operations`
	`result1 = math_operations.user_ad(8,3)`
	# Output: 11
	`result2 =`
	`math_operations.user_subt(8,3)`
	# Output: 5

(c) Method: It is well-known that python is an object-based language. A method is a function that is associated with an object or class. Methods are defined inside classes and operate on instances of that class (they are essentially functions that belong to an object).

Class Creation	Use of Class

`def __init__(self, name):`
`        self.name = name`
	`def greet(self):`
	`        return f"Hello,`
	`{self.name}!"`
	`person = Person("Anna")`
	`print(person.greet())`
	
	# Output: Hello, Anna!

(d) Package: A package is a collection of modules organized in a directory structure. A directory is recognized as a package when it contains an `__init__.py` file (which may be empty or have initialization code). Packages allow us to organize multiple modules into a structured namespace.

Example: Suppose we have a package named my_math with the following structure:

```
my_math/
        __init__.py
        operations.py
        constants.py
```

- In operations.py, the actual code is given below:

# operations.py

```
def add(a, b):

                return a + b
```

- In constants.py, the actual code is given below:

  # constants.py

```
 PI = 3.14159
```

- To use the my_math package, the actual lines of code are given below:

```
from my_math import operations, constants
 result = operations.add(2, 3)      # Output: 5
 pi_value = constants.PI        # Output: 3.14159
```

1.3 Some Common External Packages

An overview of common external packages and their uses are presented in the following table:

Table: External packages and their Uses

Broad Are	Name of Packages	Descriptions
Data Science & Machine Learning	**NumPy**	NumPy provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays. It is essential for scientific computing.
	Pandas	Pandas is used for data manipulation and analysis, providing data structures like DataFrames to handle structured data. It's a must-have for data science tasks.
	Scikit-Learn	It is a machine learning library that includes simple and efficient tools for predictive data analysis, such as classification, regression, clustering, and dimensionality reduction.
	TensorFlow	This is an open-source deep learning framework developed by Google. It's widely used for building and training neural networks, especially for large-scale applications.
	PyTorch	PyTorch is another deep learning framework, popular in academia and industry for building and experimenting with neural networks. Known for its dynamic computation graph and ease of use.
	Matplotlib & Seaborn	It is a plotting library that makes it easy to create static, animated, and interactive visualizations in Python. Often used alongside Pandas for data visualization.

Web Development	**Flask**	It is a lightweight web framework suitable for building small to medium-sized applications. Its simplicity and flexibility make it popular for RESTful APIs.
	Requests	It simplifies HTTP requests in Python, making it easy to send and receive data from web APIs. Great for web scraping and REST API interactions.
Data Visualization	**Plotly**	It is popular for interactive, web-based visualizations.
	Bokeh	It creates interactive and real-time visualizations.

1.4 Installation of External Packages

External packages for python are usually installed via pip. The general instruction to install external packages are given below:

(a) Installation for single package:

```
pip install package_name
```

(b) Installation for a list of packages such as numpy, pandas and scikit-learn

```
pip install numpy pandas scikit-learn
```

Using external packages can significantly accelerate your development process, providing specialized functions and utilities that are widely used and tested by the community.

1.5 How to import Packages in Python Source Files

Packages are included using the import statement. We can import a whole package, a specific module from a package, or specific functions or variables.

- **Code for importing an entire package**: `import my_math`
- **Code for importing specific modules from a package**:

```
from my_math import specific_module_name
```

- **Code for importing specific functions or variables from a module within a package**:

```
from my_math.operations import add
from my_math.constants import PI
```

Each of these imports allows code reuse, modularity, and separation of concerns, helping make Python code more organized and maintainable.

1.6 Uses of External Package SymPy

SymPy is a powerful tool for symbolic computation in python. It is essentially a Python library for symbolic mathematics, offering robust tools for algebraic manipulation, calculus, equation solving, and many more. Its functionality is valuable across many fields, from elementary algebra, symbolic simplifications and equation solving to calculus, linear algebra, and plotting. SymPy also includes subpackages to deal with advanced and specialized topics such as statistics, category theory, quantum logic ect. Unlike numerical computation libraries like NumPy, SymPy focuses on exact, symbolic computation rather than approximate, numerical results.

This section introduces the basic functionality of the SymPy (SYMbolic PYthon) library. Unlike numerical computation, which deals with specific numbers, symbolic computation focuses on processing and transforming general variables. The SymPy homepage (https://www.sympy.org/) offers comprehensive and up-to-date documentation for the library. While symbolic calculations are significantly slower than floating-point operations (e.g., symbolic calculations with decimals), they are a valuable tool for preparing code and performing symbolic tasks. Occasionally, symbolic operations are used in simulations to derive the most efficient numerical code before execution.

1.7 Symbols and Symbolic Expressions

In formation or construction of mathematical expression, we need to define variable/ variables and this can be done using external package sympy in python. The symbol class in sympy is a fundamental building block in the SymPy library, representing symbolic variables in mathematical expressions. It allows us to define variables that can be manipulated algebraically, rather than being assigned a specific numerical value.

1.7.1. The way of defining variable/ variables

To use the symbol class, we first import it from the SymPy library such as:

```
from sympy import Symbol
```

The symbolic variables are then defined in the following way.

For single variable	For more than one variable

`from sympy import symbols`
`x = symbols('x')`
#Here, ‘x’ is the name of the symbolic variable.
	`from sympy import symbols`
	`x,y,... = symbols('x,y,...')`
	# Here, ‘x’ and “y” are the names of the symbolic variables.

We can use this symbol in algebraic expressions and perform various symbolic operations.

Remarks:

1. The name of a symbol is a string that identifies it such as:

```
z = Symbol('z')
print(z)

# Output: z
```

2. Symbols can have optional assumptions about their properties, such as being positive, real, or integer:

```
a = Symbol('a', positive=True)
print(a.is_positive)

# Output: True
```

3. Common assumptions include:

```
positive=True: # The symbol is strictly greater than zero
integer=True: # The symbol represents an integer,
real=True:  # The symbol is a real number
```

1.7.2. The way of defining symbolic expressions/ functions

We need to define different types of expressions/ functions to express physical problems or solve mathematical problems. We can create such expressions by defining symbolic variables. These types of symbolic expressions/ functions can be used to do many types of mathematical formulation and calculation in a broad sense. Let us show how to define symbolic expressions/ functions.

```
Code:
from sympy import *         # Importing everything from sympy
x= Symbol('x')               # way to define symbolic function
f=x*log(x**2+2)-sin(3*x)     # way to define symbolic function
print("Original symbolic function showing as: {}".format(f))

#Output:
Original symbolic function showing as: x*log(x**2 + 2) - sin(3*x)
```

The Symbolic expressions /functions can be used as an object in a python function or command. The symbolic function can be used to find limits at a point, differentiation of function, definite or indefinite integration etc.

1.7.3. Difference between a variable in core python and a symbolic variable in SymPy

Variables occur in both core Python and SymPy. The primary difference between variables in core Python and symbolic variables in SymPy lies in their nature and purpose. Python variables are used for storing and manipulating data, whereas SymPy symbols are abstract representations of mathematical variables used for symbolic computation.

Table: Differences between PythonVariables and SymPy Variables

Python Variables	SymPy Variables

It represents a reference to a value in memory.	It represents an abstract mathematical variable.
It is used for programming and data manipulation.	It is used for symbolic mathematics and algebra.
It requires a specific value (e.g., x = 15).	It does not require a value (e.g., x = Symbol('x')).
Operations are numerical or based on data types.	Operations are symbolic and algebraic.
x = 7; y = x + 1 (numerical result: 8)	x = Symbol('x'); expr = x + 1 (symbolic result: x + 1).
Python variable is a built-in functionality.	Symbolic variable requires the SymPy library (from sympy import Symbol).

Examples
1. Python Variable

Code:
```
x = 7           # Assignment of a value to the variable
y = x + 2       # Numerical computation
print(y)        # Printing the content

# Output:  9
```

2. Sympy Symbolic Variable

Code:
```
from sympy import Symbol   #Importing symbol class
x = Symbol('x')            # Defines a symbolic variable
y = x + 12                 # Creates a symbolic expression
print(y)                   # Printing the content of y

# Output: x + 12
```

1.8. Symbolic Computation vs. Numerical Computation

Numerical Operations with Python Variables are used to directly perform calculations using assigned values and limited to specific data types (int, float, etc.).

Example:

Code:
```
a = 3
b = 4
```

```
result = a * b
print(result)

# Output: 12
```

Symbolic Operations with SymPy Symbols are used to manipulate expressions without assigning specific values and these are useful for algebra, calculus, and equation solving.

Example:

Code:
```
from sympy import symbols, expand # Importing section
x, y = symbols('x y')             # Defining symbol
expr = (x + y)**2                 # Creating expression
expanded_expr = expand(expr)      # Use of expand
print(expanded_expr)              # Print the output

# Output: x**2 + 2*x*y + y**2
```

1.9. Use of Python Variables and Symbolic Variables

- **Python Variables are used in the following situations when we need**
  - general programming tasks.
  - specific numerical values for computations.
  - procedural or object-oriented logic.
- **SymPy Symbols are used in the following situations when we**
  - deal mathematical problem-solving including algebra, calculus, and equation solving.
  - deal with symbolic expressions or formulas.
  - prepare efficient numerical code through symbolic analysis.

1.10. Working with Symbolic expressions

After defining variable/ variables, we can construct various types of mathematical functions, expressions or equations. As for example,

Python Code	Output

`from sympy import symbols`
`x = symbols('x')`
`expr=x**5-7*x**3-2*x**2-19`
`print(expr)`
	`x**5-7*x**3-2*x**2-19`

`from sympy import symbols`
`x, y = symbols('x y')`
`fx=x**2*y-x*y**2+x*y-1`
`print(fx)`
	`x**2*y-x*y**2+x*y-1`

`from sympy import symbols`
`x, y = symbols('x y')`
`gx=x*log(x**2*y-5*x*y+3*x*y**2)`
`+cos(x+y)`
`print(gx)`
	`x*log(x**2*y-5*x*y+3*x*y**2)+co`
	`s(x+y)`

1.11. Mathematical Manipulation of Symbolic Expressions/ functions

The external package SymPy is a powerful Python library for symbolic mathematics, making it an excellent tool for tasks like simplifying and reducing the complex mathematical expressions in simplified forms, the factoring, expanding the mathematical expressions and substituting in expressions. We are going to demonstrate use of SymPy for these purposes:

11.1. Simplifying the Expressions: Simplification is often used in algebra and calculus to make equations more manageable or to reduce terms.

Example:

Code:
```
from sympy import symbols, simplify
x, y = symbols('x y')
expr = (x**2 + 2*x*y + y**2) / (x + y)
simplified_expr = simplify(expr)
print(simplified_expr)
```

Output:
```
x + y
```

Here, SymPy recognizes that the expression can be simplified to $x + y$ by canceling terms.

11.2. Factoring Expressions: To factorize expressions, SymPy uses the factor() function.
Example:

Code:
```
from sympy import symbols, factor
x, y = symbols('x y')
expr = x**2 - 5*x + 6
factored_expr = factor(expr)
print(f"Factored Expression: {factored_expr}")
```

Output:
```
Factored Expression: (x - 3)*(x - 2)
```

11.3. Expanding Expressions: To expand expressions, SymPy uses the expand() function.

Example:

Code:

```
from sympy import expand
expr = (x + 2)*(x - 3)
```

```
expanded_expr = expand(expr)
print(f"Expanded Expression: {expanded_expr}")

#Output: Expanded Expression: x**2 - x - 6
```

11.4. Substituting Values: To substitute values into an expression, SymPy uses the subs() method.

Example:

Code:
```
expr = x**2 + 2*x + 1
value_substituted = expr.subs(x, 3)   # Substitute x = 3
print(f"Expression after substitution: {value_substituted}")

# Output: 16
```

The expression $x^2 + 2x + 1$ is evaluated at x=3 yielding 16.

Example: Write symbolic expression $x^3 - 3x^2 + 4$ and then factor it and calculate the value after substitution of x=2.

Code:
```
from sympy import *
x = symbols('x')
expr = x**3 - 3*x**2 + 4
factored = factor(expr)
expanded = expand(factored)
substituted = factored.subs(x, 2)
print(f"Factored: {factored}, Expanded: {expanded},
Substituted (with x=2): {substituted}")
```

Output:
```
Factored: (x - 2)**2*(x + 1), Expanded: x**3 - 3*x**2 + 4,
Substituted ( with x=2): 0
```

11.5. Combining Operations: We can combine various operations for more complex workflows.

Example: Write an expression, factored it, then expand it and calculate the value for $x = 2$. Print the intermediate result separately.

Code:
```
from sympy import *
x = symbols('x')
expr = x**3 - 3*x**2 + 4
factored_expr = factor(expr)
expanded_expr = expand(factored_expr)
substituted_value = factored_expr.subs(x, 2)
```

```
print(f"Original Expression: {expr}")
print(f"Factored Expression: {factored_expr}")
print(f"Expanded Back: {expanded_expr}")
print(f"Substitution (x=2): {substituted_value}")
```

Output:

```
Original Expression: x**3 - 3*x**2 + 4
Factored Expression: (x - 2)**2*(x + 1)
Expanded Back: x**3 - 3*x**2 + 4
Substitution (x=2): 0
```

11.6. Working with Multiple Variables: SymPy also works seamlessly with multiple variables.

Example: Give an example to deal with expressions with two variables.

Code:

```
from sympy import symbols, factor, expand
x, y = symbols('x y')
expr = x**2 + 2*x*y + y**2
factored_expr = factor(expr)
expanded_expr = expand(factored_expr)
print(f"Factored Expression: {factored_expr}")
print(f"Expanded Expression: {expanded_expr}")
```

Output:

```
Factored Expression: (x + y)**2
Expanded Expression: x**2 + 2*x*y + y**2
```

1.12. Pretty Printing

Pretty Printing in SymPy refers to displaying mathematical expressions in a more human-readable or formatted manner, similar to how we see them in textbooks or on paper. SymPy provides the *pprint()* function and support for LaTeX rendering in Jupyter Notebooks to achieve this. Here are some use of *pprint()* presented below

12.1. Use of pprint() for Pretty Printing: The `pprint()` function in SymPy prints expressions in a structured format directly in the console.

Example 1: Give an example to deal with Simple Polynomial

Code

```
from sympy import symbols, pprint
x = symbols('x')
expr = x**2 + 2*x + 1
pprint(expr)
```

Output:

```
 2
x  + 2⋅x + 1
```

Example 2: Give an example to deal with Fraction

Code:
```
from sympy import Rational
expr = Rational(3, 4) + Rational(2, 5)
pprint(expr)
```
Output:
```
  23
──────
  20
```

Example 3: Give an example to deal with Nested Expression

Code:
```
from sympy import sin, cos
x = symbols('x')
expr = sin(x)**2 + cos(x)**2
pprint(expr)
```
Output:
```
2    2
sin (x) + cos (x)
```
12.2. Use of init_printing() for Interactive Environments: In interactive environments (like Jupyter Notebooks), SymPy can display expressions as LaTeX-rendered output. Use `init_printing()` to enable this.

Example 4: Enabling LaTeX Printing

Code
```
from sympy import init_printing
init_printing()  # Enables LaTeX rendering in Jupyter
expr = x**2 + 2*x + 1
expr  # Automatically pretty-prints in LaTeX style
```

Output: $x^2 + 2x + 1$

Example 5: (Pretty Printing Matrices) Pretty printing works for matrices as well.

Code:
```
from sympy import Matrix
matrix = Matrix([[1, 2], [3, 4]])
pprint(matrix)
```
Output:
```
⎡1  2⎤
⎢    ⎥
⎣3  4⎦
```

1.13. Solving Equations using SymPy

Solving equations with SymPy is straightforward. SymPy provides the `solve()` function, which can handle algebraic, transcendental, and systems of equations.It supports both linear and nonlinear equations and can solve systems of equations as well. When provided with an expression containing a symbolic variable, such as x, `solve()` computes the value of that variable. By default, the function assumes the given expression is set equal to zero, meaning it finds the value that satisfies this condition.

For example, consider the simple equation $2x - 11 = 9$. Before using `solve()`, we must rewrite it so that one side equals zero: $2x - 11 - 9 = 0$. Once in this form, we can apply the `solve()` function to determine the value of x. Below are examples of solving equations using SymPy.

Example-1: Solve the Linear Equation Solve $2x - 11 = 9$

Code:
```
from sympy import symbols, Eq, solve
x = symbols('x')
eq = 2x-11-9  # Represents x + 2 = 5
solve(eq)
```
Output:
```
Solution: [10]
```

When we use `solve()`, it determines that $x = 10$ because this value satisfies the equation $10x - 11 = 9$. The result is returned as a list, since an equation may have multiple solutions, for example, a quadratic equation typically has two solutions, in which case the list will contain all possible values.

Additionally, `solve()` can return the result in the form of dictionaries, where each dictionary contains the variable as a key and its corresponding solution as the value. This format is particularly useful for solving simultaneous equations with multiple variables, as it clearly associates each solution with its respective variable.

In SymPy, `Eq` stands for Equation. It is used to explicitly represent an equation where the left-hand side (LHS) is equal to the right-hand side (RHS). Syntax for `Eq` is `Eq(lhs, rhs)`. Where lhs: Left-hand side of the equation.

rhs: Right-hand side of the equation.

Eq(lhs, rhs) represents lhs = rhs.

For example:

Code:
```
from sympy import Eq, symbols
x = symbols('x')
equation = Eq(x**2 - 4, 0)
print(equation)
```

This creates the equation $x^2 - 4 = 0$.

1.13.1. Solving a Single Equation

Example: Solve the Linear Equation Solve $x + 12 = 15$.

Code:
```
from sympy import symbols, Eq, solve
x = symbols('x')
eq = Eq(x + 12, 15)  # Represents x + 2 = 5
solution = solve(eq, x)
print(f"Solution: {solution}")
```

Output:
```
Solution: [3]
```

Example: Solve the symbolic equation $ax + b = 0$.

Code
```
a, b = symbols('a b')
eq = Eq(a*x + b, 0)
solution = solve(eq, x)
print(f"Symbolic Solution: {solution}")
```

Output:
```
Symbolic Solution: [-b/a]
```

1.13.2 Solving a Quadratic Equation

In core Python, we determined the roots of the quadratic equation $ax^2 + bx + c = 0$ by using the root formulas and substituting the values of the constants a, b, and c. Now, we'll explore how the SymPy `solve()` function can find the roots without using manual formula substitution. Let's look at an example.

Example: Solving a Quadratic Equation $x^2 - 13x + 40 = 0$

Code
```
from sympy import solve
x = Symbol('x')
eq = Eq(x**2 - 13*x + 40, 0) #Define the Equations Using Eq
solution = solve(eq, x)
print(f"Solution: {solution}")
```

Output:
```
Solution: [5, 8]
```

Here, SymPy finds that the roots of the equation $x^2 - 13x + 40 = 0$ are 5 and 8.

Example: Solve the Quadratic Equation $x^2 + x + 1 = 0$

We know that the roots of the given equations are complex.

Let's attempt to find the complex roots of the equation using solve().

Code

```
from sympy import solve
x = Symbol('x')
eq = Eq(x**2 +x + 1, 0) #Define the Equations Using Eq
solution = solve(eq,  dict=True)
print(f"Solution: {solution}")
```

Output:

```
Solution: [{x: -1/2 - sqrt(3)*I/2}, {x: -1/2 + sqrt(3)*I/2}]
```

Here, 'I' indicates a symbol to denote an imaginary component**.**

Example: Solve the Quadratic Equation $x^2 + 6x + 5 = 0$.

Here we solve the equation without creating an equation using Eq.

Code

```
from sympy import solve
x = Symbol('x')
expr = x**2 +6*x + 5 #Define the expression
solution = solve(expr, dict=True)
print(f"Solution: {solution}")
```

Output:

```
Solution:  [{x: -5}, {x: -1}]
```

First, we define the symbol x and create an expression representing the quadratic equation $x^2 + 6x + 5 = 0$. Then, we call the solve() function with this expression. The second argument, dict=True, ensures that the solutions are returned as a list of Python dictionaries. Each dictionary contains the symbol as a key and its corresponding solution as the value. If no solutions exist, an empty list is returned. In this case, the roots of the given equation are -5 and -1.

1.13.2 Solving for Several Variables

To solve a system of equations using SymPy, you can utilize the `solve()` function in conjunction with `Eq`. Let's look at a step-by-step guide:

1. **Import Necessary Functions:**
   ```
   from sympy import symbols, Eq, solve
   ```
2. **Define the Variables:**
   ```
   x, y = symbols('x y')
   ```
3. **Define the Equations Using Eq:**
   ```
   eq1 = Eq(2*x + y, 10)
   eq2 = Eq(x - y, 2)
   ```
4. **Solve the System:**
   ```
   solution = solve((eq1, eq2), (x, y))
   ```
5. **Display the Solution:**
   ```
   print(solution)
   ```

Complete Example:

```
from sympy import symbols, Eq, solve
x, y = symbols('x y')             # Define the variable
eq1 = Eq(2*x + y, 10)             # Define the equations
eq2 = Eq(x - y, 2)
# Solve the system of equations
solution = solve((eq1, eq2), (x, y))
# Display the solution
print(solution)
```

Output:

```
{x: 4, y: 2}
```

Example: Solve a system of linear equations: $x + y = 15; x - y = 11$

Code

```
y = symbols('y')
# Defining the system of equations
eq1 = Eq(x + y, 15)
eq2 = Eq(x - y, 11)
solution = solve([eq1, eq2], (x, y))
print(f"Solution: {solution}")
```

Output:

```
Solution: {x: 13, y: 2}
```

1.13.4. Solving Polynomial Nonlinear Equations

A nonlinear equation is an equation where the variable appears with exponents other than 1, or within transcendental functions like exponentials, logarithms, or trigonometric functions. Unlike linear equations, nonlinear equations may have multiple or no solutions and often require numerical or symbolic techniques to solve. SymPy provides powerful tools to solve nonlinear equations, including `solve()` for symbolic solutions and `nsolve()` for numerical approximations. A polynomial nonlinear equation is an equation where the variable appears with exponents greater than 1 (quadratic, cubic, quartic, etc.). These equations can often be solved exactly using algebraic methods. SymPy provides the `solve()` function to find symbolic solutions for polynomial equations. A simultaneous polynomial nonlinear system consists of multiple polynomial equations that need to be solved together. SymPy `solve()` or `nonlinsolve()` can be used to find exact solutions.

Example: Solve a system of nonlinear equations: $x^2 + y^2 = 25;\ x + y = 7$

Code(using solve()):

```
from sympy import symbols, Eq, solve
x, y = symbols('x y')
# Defining the system of equations
```

```
eq1 = Eq(x**2 + y**2, 25)
eq2 = Eq(x + y, 7)
solution = solve([eq1, eq2], (x, y))
print(f"Solution: {solution}")

Output: Solution: [(3, 4), (4, 3)]
```

Example: Solve a system of nonlinear equations: $x^2 + y^2 = 25;\ x^2 - y^2 = 7$

Code (using nonlinsolve()):

```
from sympy import symbols, Eq, nonlinsolve
x, y = symbols('x y')
# Defining the system of equations
eq1 = Eq(x**2 + y**2, 25)
eq2 = Eq(x**2 - y**2, 7)
solutions_set = nonlinsolve([eq1, eq2], [x, y])
print("Solutions (nonlinsolve):", solutions_set)
```

Output:

```
Solutions (nonlinsolve): FiniteSet((-4, -3), (-4, 3), (4, -3), (4, 3))
```

1.13.5. Solving Transcendental Equations

A **transcendental equation** is an equation that involves transcendental functions, such as exponential, logarithmic, or trigonometric functions, which cannot be expressed using only algebraic operations. Unlike polynomial equations, transcendental equations often do not have closed-form solutions and require numerical or symbolic methods to solve.

SymPy provides powerful tools to solve transcendental equations, including solve() for symbolic solutions and nsolve() for numerical approximations. Many cases, transcendental equations do not have an exact solution, in this case solve() function is not applicable.

Example: Solve the transcendental equation $e^x = 3x$.

Code:

```
from sympy import symbols, Eq, exp,  nsolve
x = symbols('x')
equation = Eq(exp(x) - 3*x, 0)  # e^x = 3x
# Numerical solution
numerical_sol = nsolve(equation, x, 1)  # Initial guess x=1
print("Numerical Solution:", numerical_sol)
```

#Output:

```
Numerical Solution: 1.51213455165784
```

Example: Solve the transcendental equation $sin(x) = 0.5$ using solveset().

SymPy's solveset() function is an advanced solver that returns solutions as sets. It works well for equations with symbolic solutions and can handle many features.

Code:
```
from sympy import sin, pi, solveset
solution = solveset(Eq(sin(x), 0.5), x)
print(f"Solution: {solution}")
```

Output:
```
Solution: {π/6, 5π/6}
```

Example (Solving Inequalities): Solve the inequation $x^2 - 4 > 0$

Code:
```
from sympy import solve_univariate_inequality
inequality = x**2 - 4 > 0
solution = solve_univariate_inequality(inequality, x)
print(f"Solution: {solution}")
```

Output:
```
Solution: (−∞, −2) ∪ (2, ∞)
```

Example: Solve $sin(x) - 0.5 = 0$ using the sympy package.

Code:
```
from sympy import *
x = symbols('x')
r=nsolve(sin(x)-0.5, x, 1)
print(f"Solution x: {r}")
```

Output:
```
Solution x: 0.523598775598299
```

1.14. Calculus: Limit, Differentiation and Integration

SymPy is extensively used in calculus for differentiation, integration, and limits. Its differentiation feature is beneficial for every applied field like physics and engineering. Actually, derivatives are used to compute rates of change and other properties.

1.14.1. Limits and Continuity

SymPy can calculate limits, which are essential in calculus, especially when analyzing the behavior of functions near a point.

Example (Evaluation of limit from positive side of a function):

Code:
```
from sympy import limit
from sympy import *
x=Symbol('x')
f = 1 / (x - 1)
lim_x1 = limit(f, x, 1, '+')
print(lim_x1)
```

# Output:
```
oo (∞)
```

SymPy calculates the limit of f as x approaches 1 from the positive side. This is helpful in calculus for studying continuity and behavior of functions at critical points.

Example (Evaluation of limit of a function):

Code:
```
from sympy import limit
expr = (x**2 - 1) / (x - 1)
lim = limit(expr, x, 1)
print(lim)
```

#Output: `2`

SymPy finds that the limit of $\frac{x^2-1}{x-1}$ as x approaches 1 is 2.

1.14.2. Differentiation of a function

SymPy supports differentiation and integration, core to calculus. These features are extensively used in physics, machine learning, and economic modeling.

Example (Differentiation):

Code:
```
from sympy import *
f = x**3 + 2*x**2 + x
f_prime = diff(f, x)
print(f_prime)
```

#Output: 3*x**2 + 4*x + 1

In this example, SymPy computes the derivative of f with respect to x, a task useful in physics (e.g., calculating velocity from displacement) and in optimizing cost functions in economics.

Example (Differentiation):

Code:
```
from sympy import diff
expr = x**3 + 3*x**2 + 5
```

```
derivative = diff(expr, x)
print(derivative)

#Output:
3*x**2 + 6*x
```

SymPy computes the derivative $3x^2 + 6x$ of the function $x^3 + 3x^2 + 5$.

1.14.3. Integration with SymPy

Example (Integration):

```
Code:
from sympy import integrate
area_under_curve = integrate(f, (x, 0, 2))
print(area_under_curve)

# Output: 14/3
```

SymPy calculates the definite integral of f from 0 to 2. This result is vital in fields like probability (finding areas under curves) and physics (calculating the work done by a force over a distance).

Example (Integration):

```
Code:
from sympy import integrate
expr = x**2 + x
integral = integrate(expr, x)
print(integral)

# Output:
x**3/3 + x**2/2
```

SymPy provides the indefinite integral $\frac{x^3}{3} + \frac{x^2}{2}$ of $x^2 + x$.

The following table shows a list of **calculus commands** in SymPy:

Table: List of calculus commands in SymPy

Operation	Description	SymPy Function
Differentiation	Computes the derivative of an expression	diff(f, x)
Partial Derivative	Computes the partial derivative	diff(f, x, y)

Higher-Order Derivative	Computes the nth derivative	diff(f, x, n)
Gradient	Computes the gradient vector	gradient(f, (x, y, z))
Jacobian	Computes the Jacobian matrix	Jacobian(f, (x, y, z))
Hessian Matrix	Computes the Hessian matrix (second-order partial derivatives)	hessian(f, (x, y, z))
Divergence	Computes the divergence of a vector field	divergence(F, (x, y, z))
Curl	Computes the curl of a vector field	curl(F, (x, y, z))
Integration	Computes the integral of an expression	integrate(f, x)
Definite Integral	Computes a definite integral	integrate(f, (x, a, b))
Multiple Integrals	Computes double and triple integrals	integrate(f, (x, a, b), (y, c, d))
Limit	Computes the limit of a function	limit(f, x, a)
Series Expansion	Computes the Taylor/Maclaurin series	series(f, x, n)
Laplace Transform	Computes the Laplace transform	laplace_transform(f, t, s)
Inverse Laplace Transform	Computes the inverse Laplace transform	inverse_laplace_transform(F, s, t)
Fourier Transform	Computes the Fourier transform	fourier_transform(f, x, k)
Inverse Fourier Transform	Computes the inverse Fourier transform	inverse_fourier_transform(F, k, x)
Residue	Finds the residue of a function at a singularity	residue(f, x, a)

1.14.4. Series Expansion with SymPy

Series expansions are widely used in calculus and mathematical analysis for approximating functions. **SymPy** can compute the Taylor or Maclaurin series of functions.

Example (Series):

Code:
```
from sympy import sin, series
taylor_series = series(sin(x), x, 0, 6)
print(taylor_series)
```
# Output:
```
x - x**3/6 + x**5/120 + O(x**6)
```

This is the Maclaurin series expansion of $sin(x)$ up to x^5.

Example (Series):

Code:
```
from sympy import series, sin
taylor_series = series(sin(x), x, 0, 5)
print(taylor_series)
```
# Output:
```
x - x**3/6 + x**5/120 + O(x**6)
```

SymPy computes a Taylor series expansion of sin(x) up to the fifth term, useful in approximating functions in physics (e.g., small angle approximations).

1.14.5. Matrix algebra with SymPy

SymPy includes linear algebra tools for operations on matrices, like finding determinants, eigenvalues, and inverses, which are important in areas like computer graphics, quantum mechanics, and systems of equations.

Example (Matrix):

Code:
```
from sympy import Matrix
matrix = Matrix([[1, 2], [3, 4]])
determinant = matrix.det()
inverse_matrix = matrix.inv()
print(determinant)
```
# Output: -2   print(inverse_matrix)

# Output: Matrix([[-2, 1], [3/2, -1/2]])

SymPy calculates the determinant and inverse of the given matrix.

The following table shows a list of common **matrix operations** in SymPy :

Table: List of common matrix operations

Operation	Description	SymPy Function
Matrix Creation	Creates a symbolic or numerical matrix	Matrix([[a, b], [c, d]])
Identity Matrix	Creates an Identity matrix I_n	eye(n)
Zero Matrix	Creates a matrix filled with zeros	zeros(n, m)
Ones Matrix	Creates a matrix filled with ones	ones(n, m)
Transpose	Transposes the matrix AAA	A.T
Inverse	Computes the inverse of a matrix	A.inv()
Determinant	Computes the determinant of a matrix	A.det()
Rank	Computes the rank of a matrix	A.rank()
Eigenvalues	Finds the eigenvalues of a matrix	A.eigenvals()
Eigenvectors	Computes the eigenvectors	A.eigenvects()
Characteristic Polynomial	Finds the characteristic polynomial	A.charpoly()
Trace	Computes the trace (sum of diagonal elements of a matrix)	A.trace()
Row Echelon Form	Converts matrix to row echelon form	A.rref()
LU Decomposition	Performs LU factorization	A.LUdecomposition()
QR Decomposition	Performs QR decomposition	A.QRdecomposition()
Cholesky Decomposition	Computes the Cholesky factorization	A.cholesky()
Singular Value Decomposition (SVD)	Computes SVD	A.singular_value_decomposition()
Matrix Addition	Adds two matrices	A + B
Matrix Subtraction	Subtracts two matrices	A - B
Matrix Multiplication	Multiplies two matrices	A * B
Scalar Multiplication	Multiplies a matrix by a scalar	k * A
Element-wise Power	Raises each element to a power	A**n

Solving Linear Systems	Solves AX=B for X	A.solve(B)
Adjugate (Adjoint) Matrix	Computes the adjugate matrix	A.adjugate()
Cofactor Matrix	Computes the cofactor matrix	A.cofactor_matrix()
Jordan Form	Computes the Jordan normal form	A.jordan_form()

1.14.6 Special Matrices in SymPy

SymPy provides built-in functions to create various **special matrices**, which have unique structures and properties. These matrices are commonly used in linear algebra, numerical analysis, and symbolic computations.

The following table shows a list of some important special matrices in SymPy:

Table: List of some important special matrices in SymPy

Matrix Type	Description	SymPy Function
Identity Matrix	Square matrix with ones on the diagonal and zeros elsewhere	eye(n)
Zero Matrix	A matrix with all elements as zero	zeros(n, m)
Ones Matrix	A matrix with all elements as one	ones(n, m)
Diagonal Matrix	A square matrix with specified values on the diagonal and zeros elsewhere	diag(*values)
Jordan Block Matrix	Block diagonal matrix used in Jordan form computations	jordan_block(size, value)
Hilbert Matrix	A matrix where each element is given by Hi,j=1i+j−1H_{i,j} = \frac{1}{i+j-1}Hi,j=i+j−11	Hilbert(n)
Hadamard Matrix	A square matrix whose entries are +1 or -1, used in signal processing	hadamard(n)
Vandermonde Matrix	A matrix where each row follows a geometric progression	Matrix.vandermonde(x, n)
Companion Matrix	A special square matrix associated with polynomials	companion(polynomial)
Toeplitz Matrix	A matrix where each descending diagonal has constant values	Toeplitz(row, col)

Circulant Matrix	A special Toeplitz matrix where each row is a cyclic shift of the previous one	circulant(vector)
Random Matrix	A matrix with random elements	randMatrix(n, m)
Wronskian Matrix	A determinant useful in differential equations	wronskian(functions, var)

1.14.7. Differential Equations

SymPy provides functions to solve ordinary differential equations (ODEs), which are frequently encountered in engineering, physics, and economics.

Example (Differential Equation):

Code
```
from sympy import Function, Eq, dsolve
y = Function('y')
ode = Eq(y(x).diff(x) - y(x), 0)
solution = dsolve(ode)
print(solution)
```
# Output: Eq(y(x), C1*exp(x))

Here, SymPy finds the solution $y(x) = C1e^{x}$ to the differential equation $\frac{dy}{dx} - y = 0$.

Table: List of commands for differential equations in SymPy

Operation	Description	SymPy Function
Define a Differential Equation	Represents a differential equation	Eq(y.diff(x) - y, 0)
First-Order ODE Solution	Solves a first-order ODE	dsolve(Eq(y.diff(x) - y, 0), y(x))
Higher-Order ODE Solution	Solves higher-order ODEs	dsolve(Eq(y.diff(x, 2) + y, 0), y(x))
System of ODEs	Solves a system of coupled ODEs	dsolve([Eq(y1.diff(x), y2), Eq(y2.diff(x), -y1)], [y1, y2])
General Solution	Finds the general solution of an ODE	dsolve(ODE, func)

Particular Solution	Finds a particular solution with initial conditions	dsolve(ODE, func, ics={y(0): 1, y.diff(x).subs(x, 0): 0})
Verify Solution	Checks if a given function is a solution of the ODE	checkodesol(Eq(y.diff(x) - y, 0), y(x) - exp(x))
Classify ODE	Determines the type of an ODE	classify_ode(Eq(y.diff(x) - y, 0))
Series Solution	Finds a series expansion solution	dsolve(ODE, hint='series')
Numerical Solution (ODE Solver)	Solves an ODE numerically	odeint(f, y0, t) (from SciPy)

1.15. Worked Out Examples

Example-1: What is a Computer Algebra System (CAS)? Give some common examples. Write Some key features of CAS.

Answer: A **Computer Algebra System (CAS)** is a software tool designed to perform symbolic mathematical computations. Unlike traditional numerical calculators, a CAS can manipulate algebraic expressions, solve equations symbolically, perform differentiation and integration, and simplify complex mathematical expressions.

Examples of CAS include SymPy (Python-based), Mathematica, Maple, Maxima, MATLAB (Symbolic Math Toolbox)

Key Features of CAS include

- Symbolic Computation: Works with algebraic expressions instead of just numbers.
- Equation Solving: Solves equations and systems symbolically.
- Differentiation & Integration: Performs symbolic calculus operations.
- Matrix Algebra: Supports operations on matrices and vectors.
- Visualization: Generates plots of mathematical functions.

Example - 2: What is SymPy, and how does it differ from NumPy?

Answer:

SymPy: SymPy (Symbolic Python) is a symbolic mathematics library for Python. It allows for exact algebraic computations, including differentiation, integration, equation solving, matrix operations, and symbolic plotting. SymPy is particularly useful for symbolic computation in mathematics, physics, engineering, and computer science.

NumPy: NumPy (Numerical Python) is a numerical computing library for Python. It Provides support for fast array operations, linear algebra, Fourier transforms, random number generation, and numerical integration. NumPy is widely used for scientific computing and machine learning due to its efficient handling of large datasets.

Table: Comparison of SymPy and NumPy

Characteristics	SymPy	NumPy
Feature	Symbolic computation (exact math)	Numerical computation (approximate math)
Type of Data	Works with symbolic expressions	Works with arrays and numbers
Accuracy	Exact results (e.g., fractions, square roots)	Approximate floating-point results
Computation Type	Algebraic manipulation	Fast numerical calculations
Different use	Solving equations, calculus, algebra	Data analysis, statistics, machine learning
Computation	diff(x**2, x) → 2*x	np.diff([1, 2, 4]) → [1, 2]

Use of SymPy vs Use of NumPy

- **Use SymPy** when you need **exact symbolic solutions** (e.g., solving algebraic equations, symbolic differentiation).
- **Use NumPy** when you need **fast numerical calculations** (e.g., working with large datasets, performing matrix operations).

Coding Example

Code :
```
from sympy import symbols, diff
x = symbols('x')
expr = x**2
derivative = diff(expr, x)
print(derivative)
```
# Output: 2*x

Code :

```
import numpy as np
arr = np.array([1, 2, 4])
diff_arr = np.diff(arr)
print(diff_arr)

# Output:  [1 2]
```

Example- 3: Use SymPy to compute the expression $x^2y^3 + x^2e^y$ for $x = 2.3,\ y = 1.75$.

Solution:

Code:

```
from sympy import symbols
x, y=symbols('x y')
y=1.75;
f=x**2*y**3+x**2*exp(y)
f1 = f.subs(x,2.3)
print(f"Substitution (x=2.3,y=1,75): {f1}")
```

Output:

```
Substitution (x=2.3,y=1,75): 58.7929419060703
```

Example - 4: How is SymPy different from other Computer Algebra Systems such as Mathematica and Maple?

Answer: SymPy is a lightweight, Python-based symbolic mathematics library that stands out among other Computer Algebra Systems (CAS) due to its flexibility, ease of integration, and pure Python implementation. Below is a comparison of SymPy vs. other major CAS like Mathematica and Maple.

Table: Key Differences SymPy vs. Mathematica and Maple

Criteria	SymPy	Mathematica	Maple
Programming Language	Pure Python	Proprietary language	Proprietary language
Cost to buy	Free & Open-source	Paid	Paid
Ease of Use	Easy for Python users	GUI-based, steep learning curve	GUI-based, better for research
Numerical Computation	Limited (via evalf())	High precision	High precision
Graphing & Visualization	Uses Matplotlib	Built-in	Built-in
Nature	**Native Python library**	Needs external APIs	Needs external APIs

Example - 5: What do you mean by Symbolic Computation? How is it different from numerical computation? Discuss with an example.

Solution: Symbolic computation (also called computer algebra) refers to the manipulation and evaluation of mathematical expressions in exact form, without numerical approximation. It allows for algebraic transformations, differentiation, integration, simplification, and equation solving in terms of symbols rather than fixed numerical values.

Key Features of Symbolic Computation are listed below:

- Works with **mathematical symbols** rather than fixed numbers.
- Provides **exact results** (e.g., fractions, square roots, and algebraic expressions).
- Allows for **differentiation, integration, and equation solving in symbolic form**.
- Used in **calculus, algebra, and symbolic logic applications**.

Example of Symbolic computation: Finding the derivative of $f(x) = x^2 + 3x + 2$ using symbolic computation, we can differentiate the function **exactly** as:

Code:
```
from sympy import symbols, diff
x = symbols('x')
f = x**2 + 3*x + 2
derivative = diff(f, x)
print(derivative)
```
# Output: 2*x + 3

Numerical computation deals with approximate solutions using floating-point arithmetic. It is widely used in scientific computing, engineering, and real-time applications where exact solutions are impractical.

Key Features of Numerical Computation are listed below:

- Works with **approximate floating-point numbers**.
- Uses **numerical methods** to compute derivatives, integrals, and solutions.
- Can handle **large datasets and high-speed computations efficiently**.
- Used in **machine learning, simulations, and data science**.

Example of Numerical Computation: Approximating the derivative of

$$f(x) = x^2 + 3x + 2$$

Instead of finding an exact formula, we compute an **approximate** derivative at $x = 2$ using numerical methods.

Python Code Using NumPy (Numerical Computation):
```
import numpy as np
def f(x):
    return x**2 + 3*x + 2
x_val = 2.0
dx = 1e-5  # Small step size
```

```
numerical_derivative = (f(x_val + dx) - f(x_val)) / dx
print(numerical_derivative)
```
# Output: ~6.99999 (close to 7)

Example - 6: How do you define a symbolic variable in SymPy?

Solution: In SymPy, a symbolic variable is defined using the symbols() or Symbol() function from the sympy module. These symbolic variables are used for algebraic manipulations, differentiation, integration, and equation solving.

Example of defining symbol using symbols():

Code:
```
from sympy import symbols
x = symbols('x')
y, z = symbols('y z')  # Multiple variables
print(x, y, z)
```
# Output: x y z

Example of defining symbol using Symbol():

Code:
```
from sympy import Symbol
a = Symbol('a')
print(a)
```
# Output: a

Example - 7: Solve the quadratic equation $ax^2 + bx + c = 0\ (a \neq 0)$ for x in SymPy considering $a,\ b,\ c$ are constants and return the solutions as a dictionary.
Solution:

Code:
```
from sympy import *
x = Symbol('x')
a = Symbol('a')
b = Symbol('b')
c = Symbol('c')
expr = a*x*x + b*x + c
solve(expr, x, dict=True)
```
Output:

$$\left[\left\{x: \frac{-b+\sqrt{-4ac+b^2}}{2a}\right\}, \left\{x: \frac{-b-\sqrt{-4ac+b^2}}{2a}\right\}\right]$$

Example - 8: Solve the system of two equations 2x+3y−6=0 and 3x+2y−12=0 in SymPy, verify the solution by substituting it back into the original equations, and ensure both equations evaluate to zero.
Solution:

Code:
```
from sympy import *
x = Symbol('x')
y = Symbol('y')
expr1 = 2*x + 3*y - 1
expr2 = 3*x + 2*y - 4
solve((expr1, expr2), dict=True)
soln = solve((expr1, expr2),dict=True)
soln = soln[0]
print(f"Solution={soln}")
expr1.subs({x:soln[x], y:soln[y]})
expr2.subs({x:soln[x], y:soln[y]})
```

Output:
```
Solution={x: 2, y: -1}
0
```

Example - 9: Expand the expression $(x + y)^3(y + z)^2 + (x + y)^2(y + z)^3$ and present it collecting the co-efficients of x and collecting the co-efficients of y.

Solution:

Code:
```
from sympy import *
x= Symbol('x')
y= Symbol('y')
z= Symbol('z')
f=x**2*(x+z)+y**2*(y+z)+x*(y**2+z**2)+x**3*z+y**3*x
print("Expanded form of the function:")
print(f.expand())
print("Expanded form of the function after collecting x:")
print(f.expand().collect(x))
print("Expanded form of the function after collecting y:")
print(f.expand().collect(y))
```

Output:
```
Expanded form of the function:
x**3*z + x**3 + x**2*z + x*y**3 + x*y**2 + x*z**2 + y**3 + y**2*z
Expanded form of the function after collecting x:
x**3*(z + 1) + x**2*z + x*(y**3 + y**2 + z**2) + y**3 + y**2*z
Expanded form of the function after collecting y:
x**3*z + x**3 + x**2*z + x*z**2 + y**3*(x + 1) + y**2*(x + z)
```

Example - 10: Write symbolic function $f(x) = xsin(x) - log(2 + x^2)$ in Python. Then find the functions after substituting x by $3x$ and $y/2$. Also find the value of the function at $x = \pi$.

Solution:

Code :
```
x= Symbol('x')
y= Symbol('y')
f=x*sin(x)-log(2+x**2)
print("The original function is: ")
print(f)
print("The substitute x with 3x in the function and find new
function: ")
print(f.subs(x,3*x))  # here x is replaced by 3x
print("The substitute x with y/2 in the function and find new
function: ")
print(f.subs(x,y/2))  # here x is replaced by y/2 (variable
different)
print("Value of the function after substitution of a numerical
value of x:")
f.subs(x,pi)   # Give functional value at x=pi
```

#Output:
```
The original function is:
x*sin(x) - log(x**2 + 2)
The substitute x with 3x in the function and find new function:
3*x*sin(3*x) - log(9*x**2 + 2)
The substitute x with y/2 in the function and find new function:
y*sin(y/2)/2 - log(y**2/4 + 2)
Value of the function after substitution of a numerical value of x:
-log(2 + pi**2)
```

Example - 11: Write $f(x) = log(sin(x) + 2tan(x) + 3) + sin(2x) - sin^3(x)$ in Python. Then find the function after substituting sin by cos.
Solution:

Code :
```
from sympy import *
x= Symbol('x')
h=log(sin(x)+2*tan(x)+3)+sin(2*x)-sin(x)**3
print("The original function is: ")
print(h)
print("Reduced function after substitution of the function sin by cos")
print(h.subs(sin,cos))
#In this case sin(x) is replaced by cos(x) in the expression
```

#Output:
```
The original function is:
log(sin(x) + 2*tan(x) + 3) - sin(x)**3 + sin(2*x)
Reduced function after substitution of the function sin by cos
```

```
log(cos(x) + 2*tan(x) + 3) - cos(x)**3 + cos(2*x)
```

Example - 12: Solve $s = ut + \frac{1}{2}at^2$ for t, and display the solutions in a well-formatted manner using SymPy.

Solution:

```
from sympy import Symbol, solve
# Define symbolic variables
s = Symbol('s')
u = Symbol('u')
t = Symbol('t')
a = Symbol('a')
# Define the equation s = ut + (1/2) * a * t^2
expr = u*t + (1/2)*a*t**2 - s
# Solve for t
solutions = solve(expr, t)
# Display solutions in a well-formatted manner
print("Solutions for t:")
pprint(solutions)
```

Example - 13: Simplify after substituting $1 - y$ for x into the expression $x^2 + xy + y^2$ using SymPy.

Solution:

Code:

```
from sympy import symbols,simplify
x, y=symbols('x y')
f=x**2+x*y+y**2
f1 = f.subs(x,1-y)
print(f"Substitution (x=1-y): {f1}")
f2=simplify(f1)
print(f"Simplified form: {f2}")
```

Output:

```
Substitution (x=1-y): y**2 + y*(1 - y) + (1 - y)**2
Simplified form: y**2 - y + 1
```

Example - 14: Substitute $x = 1 + y$ into the expression $x^3 + x^2y + xy^2 + y^3$, and simplify the resulting expression using SymPy.

Solution:

Code:

```
from sympy import symbols,simplify
x, y=symbols('x y')
f=x**3+x**2*y+x*y**2+x**3
f1 = f.subs(x,1+y)
print(f"Substitution (x=1+y): {f1}")
f2=simplify(f1)
```

```
print(f"Simplified form: {f2}")
```

Output:

```
Substitution (x=1+y): y**2*(y + 1)+y*(y + 1)**2+2*(y + 1)**3
Simplified form: 4*y**3+9*y**2+7*y+2
```

Example - 15: How can you create the expression x−5−7, and solve x−5=7 for x using SymPy?

Solution:

Code:

```
from sympy import Symbol
# Define the variable x
x = Symbol('x')
# Define the expression x - 5 - 7
expr = x - 5 - 7
# Display the expression
print("Expression:")
print(expr)
# Solve for x
 solution = solve(expr, x)
# Display the solution
print("Solution for x:")
print(solution)
```

Output:

```
Expression:  x - 12
Solution for x:   [12]
```

Example - 16: How can you take a mathematical expression as input from the user in SymPy, convert it into a symbolic expression, and store it for further operations?

Solution:

Code:

```
from sympy import sympify
expr = input('Enter a mathematical expression: ')
expr = sympify(expr)
print(f"Expression={expr}")
```

Output:

```
Enter a mathematical expression: 2*x+5*y
Expression=2*x + 5*y
```

Example - 17: Use SymPy to

(a) split $3x(x - 1)(x + 2)(x - 5)$ **into partial fractions.**

(b) simplify $sin^4(x) - 2cos^2(x)sin^2(x) + cos^4(x)$.

(c) expand $(y + x - 3)x^2 - y + 4$.

Solution:

(a)

Code:
```
import sympy as sp
# Define symbolic variable
x = sp.symbols('x')
# Define the fraction
expr = 3*x / ((x - 1) * (x + 2) * (x - 5))
# Perform partial fraction decomposition
partial_fractions = sp.apart(expr)
print(partial_fractions)
```

Output:
```
-2/(7*(x + 2)) - 1/(4*(x - 1)) + 15/(28*(x - 5))
```

(b)

```
# Define trigonometric expression
expr = sp.sin(x)**4 - 2*sp.cos(x)**2 * sp.sin(x)**2 +
sp.cos(x)**4
# Simplify the expression
simplified_expr = sp.simplify(expr)
print(simplified_expr)
```

Output: `cos(4*x)/2 + 1/2`

(c)

```
# Define symbolic variables
x, y = sp.symbols('x y')
# Define the expression
expr = (y + x - 3) * x**2 - y + 4
# Expand the expression
expanded_expr = sp.expand(expr)
print(expanded_expr)
```

Output: x**3 + x**2*y - 3*x**2 - y + 4

Example - 18: Solve the system of linear equations $2x + y = 5$; $3x - 7y = 1$.
Solution:

Code:
```
from sympy import *
x = Symbol('x')
y = Symbol('y')
eq1 = Eq(2*x+y,5)
eq2 = Eq(3*x -7*y, 1)
solution = solve([eq1, eq2], (x, y))
print(f"Solution: {solution}")
```

#Output:
```
Solution: {x: 36/17, y: 13/17}
```

Example - 19: Solve $xlog(x) = 5cos(x)$ using package sympy in Python.

Code:
```
from sympy import *
x = symbols('x')
r=nsolve(x*log(x)-5*cos(x), x, 1)
print(f"Solution x= {r}")
```

Output:
```
Solution x= 1.46005225470169
```

Exercise - I

1. What is *SymPy*, and what are its key features?
2. How do you define and declare symbolic variables in *SymPy*?
3. What is the purpose of the *sympify()* function, and how does it handle user input?
4. How can you expand, factor, and simplify algebraic expressions using *SymPy*?
5. What functions are used in *SymPy* to differentiate and integrate expressions?
6. How do you solve algebraic equations symbolically in *SymPy*?
7. What is the purpose of the *subs()* function in SymPy?
8. How can you work with matrices and perform matrix operations in *SymPy*?
9. How do you convert a *SymPy* expression into a numerical value using *evalf()*?
10. How can you solve a system of equations using *SymPy*?
11. What is the *expand()* function in *SymPy*? Demonstrate its use with polynomial expressions.
12. How can SymPy be used to factorize polynomials? Provide examples.
13. Explain the use of the *subs()* function in *SymPy*. How does it help in expression evaluation?
14. Write python code to get expanded form of
    (i) $p(x) = x^2 + (x + y)^3$ and (ii) $f(x, y, z) = x^2 z + (x + y)^3 + z(x + y)^2$.
15. Write *SymPy* command to get the factors of the functions
    i) $f(x) = (x^3 - 1)(x - 1)$ ii) $f(x, y) = \left(x^4 - y^4\right)\left(x^2 - y^2\right)$ iii) $f(x) = x^6 - 1$
16. Define following symbolic functions in *SymPy* and find the value at points $x = 0.2,\ 0.4,\ 0.6,$ 0.8 using $subs()$.
    (a) $f(x) = \dfrac{5^{\frac{3}{2}}+\sqrt{x^3}}{\sqrt{5+\sqrt{x+\sqrt{x}}}}$ (b) $f(x) = \sqrt{1 - 0.162x} + \sqrt{\dfrac{3x+2}{5}}$
    (c) $f(x) = \dfrac{x^5-3x^2+7}{x^3+x-2}$ (d) $f(x) = \dfrac{x^3\sqrt{sin\,(x+\sqrt{x})}}{1+\pi x^3 log\, x}$
17. Factorise using *SymPy* Command
    (a) $(x^{64} - 1)$ (b) $\left(x^8 - 1\right) + \left(x^4 - 1\right)\left(x^2 - 1\right)$

(c) $p^2 - 2ap + \left(a^2 - b^2\right)$ (d) $x^2 - (m + n)x + (a + m)(a + n)$

18. Define symbolic function $f(x) = x^2 + 2$ in Sympy. Then
   a. add an expression xe^x to $f(x)$ and show the resulting function. Further replace x by log $log\, x$.
   b. multiply $f(x)$ by $x^5 + 2x + 3$ and show fully simplified form and its degree.

19. Let $f(x) = x^5 + 2x^3 - 3x + 7$, then find another function in y such that x and y are related by the relation $2x - 7y = 1$ *using Sympy* commands.

20. Use Sympy find roots of the following algebraic equations:
   (a) $x^n - 1 = 0$ for $2 \le n \le 8$ (b) $x^n + 1 = 0$ for $2 \le n \le 8$
   (c) $x^3 - 1.1x^2 + 4x - 4.4 = 0$ (d) $x^3 - x - 1 = 0$
   (e) $x^3 + 2x - 6 = 0$ (f) $x^3 + 7x^2 + 9 = 0$

21. Use Sympy find a root of the following Transcendental equations:
   (a) $3x - \cos x - 1 = 0$ (b) $\log x - \cos x = 0$
   (c) $x^x + x - 4 = 0$ (d) $x\tan x + 2x^2 - 2.5 = 0$
   (e) $\log\left(1 + x^2\right) + e^{k\sin\sin x} - 1.6 = 0$ for k=2.5(0.1)2.9
   (f) $x^2 - 5\log\left(ax^2 + bx + c\right) = 0$ for $a = 5,\ b = 2,\ c = 3$
   (g) $\tan ax - \tanh ax - a = 0$, where $a = 1.25$

22. Use Sympy to the system of linear equations:

(i) $x - 2y + 2z = 2$ $2x - y - 2z = 1$ $2x + 2y + z = 7$	(ii) $3x + y + 2x = 3$ $2x - 3y - z =- 3$ $x + 2y + z = 4$
iii) $2x + y + z = 10$ $3x + 2y + 3z = 18$ $x + 4y + 9z = 16$	iv) $2x + 3y + z = 9$ $x + 2y + 3z = 6$ $3x + y + 2z = 8$

23. Use *SymPy* code to evaluate the following limits:

i) $\frac{x^2+3x}{x^2+x+1}$ ii) $\frac{\sin x}{x+\cos x}$ iii) $\left(1 + \frac{1}{x}\right)^x$ iv) $\frac{\sin(1-x)}{1-x^2}$

24. Test the continuity of the following functions at indicated points using Sympy

i) $f(x) = x - [x]$, at $x = 1$.

ii) $f(x) = x^2$ *when* $0 < x < 1$; x, *when* $1 \le x < 2$; $\frac{x^3}{4}$, *when* $2 \le x < 3$

at $x = 1, 2$

iii) $f(x) = x, \quad 0 \le x \le 1; \quad x - 2, \quad 1 < x \le 2$ at $x = 1$

iv) $f(x) = 1 + x, \quad x \le 0; \quad x, \quad 0 < x < 1; \quad 2 - x, \quad 1 \le x \le 2; \quad 3x - x^2, \quad x > 2$
at $x = 0, 1, 2$

25. Use Sympy to find $\frac{dy}{dx}$ for the following:

i) $y = log\ cosh\, x$ ii) $y = \sqrt{sin\sqrt{x}}$ iii) $y = x^x + x^{\frac{1}{x}}$ iv) $y = x^x$

v) $y = \frac{\left(a+bx^{\frac{3}{2}}\right)}{cx^{\frac{5}{4}}}$, where a, b, c are constants. vi) $y = x^{x^x}$ vii) $y = \sqrt{\frac{1-x}{1+x}}$

viii) $y = (e^x \sin x)$ ix) $y = 10^{log\ sin\ x}$ x) $y = x^x$

26. Find $\frac{dy}{dx}$ from the following parametric curves using Sympy commands

i) $x = at^2,\ y = 2at$ ii) $x = a(t - \sin sin\, t),\ y = a(1 - \cos cos\, t)$

iii) $x = a(\cos cos\, t + t \sin sin\, t\ ),\ y = a(\sin sin\, t - t \cot cot\, t\ )$

iv) $x = \frac{t}{\sqrt{\cos cos\ 2t}}\quad y = \frac{t}{\sqrt{\cos cos\ 2t}}$

27. Use Sympy to

i) differentiate $\left(x^2 + ax + a^2\right) \log log\ \cot cot\ \frac{x}{2}$ with respect to $a \cos cos\ bx$.

ii) differentiate $x^n \log log\ x$ with respect to $\frac{\sin sin\sqrt{x}}{x^{\frac{3}{2}}}$.

iii) differentiate $\frac{\sqrt{1+x^2}+\sqrt{1-x^2}}{\sqrt{1+x^2}-\sqrt{1-x^2}}$ with respect to $\sqrt{1 - x^4}$.

28. Use Sympy to find 2nd , 3rd and 5th order differentiation of the following functions:

i) $y = x^5e^{3x}$ ii) $y = \frac{1}{x^2-x-2}$ iii) $y = tan\ e^{5x}$

iv) $y = x^2 log\ \frac{x}{a}$ v) $y = \frac{log\ x}{x}$ vi) $y = e^{2x}3x$

29. Use Sympy code to find partial derivatives up to 2nd order of the following functions

i) $f(x, y) = log\left(\frac{x^2-y^2}{x^3+y^3}\right)$ ii) $f(x, y) = \frac{x-y}{\sqrt{\left(x^2+y^2\right)^5}}$

then find i) $x\frac{\partial f}{\partial x} + y\frac{\partial f}{\partial y}$ and ii) $\frac{\partial^2 f}{dx^2} + \frac{\partial^2 f}{\partial y^2}$

Chapter - 2
Plotting using SymPy

2.1 Introduction

SymPy (abbreviation for Symbolic Python) is a powerful Python library for symbolic mathematics that includes built-in plotting tools for plotting mathematical functions and symbolic expressions. Using its plot module, users can effortlessly visualize equations, expressions, and functions without needing external plotting tools. This feature is particularly valuable for examining the behavior of functions, studying equation solutions, and exploring mathematical concepts. SymPy's plotting capabilities work seamlessly with symbolic expressions, enabling dynamic and interactive visualizations that enhance symbolic computations.

SymPy's plotting functionality is particularly useful for:

- **Understanding the behavior of functions**: Visualizing how a function changes over a range of values.
- **Analyzing equation and its solutions**: Exploring solutions graphically by plotting the expressions or equations.
- **Exploring mathematical concepts**: Visualizing symbolic computations to gain deeper insights into their properties.

2.2 SymPy's Plotting Module

Import Required Modules: To use the plotting features, we need to import SymPy and the necessary plotting tools:

Code:
```
from sympy import symbols, plot, sin, cos, exp
               Or
 from sympy import *
```

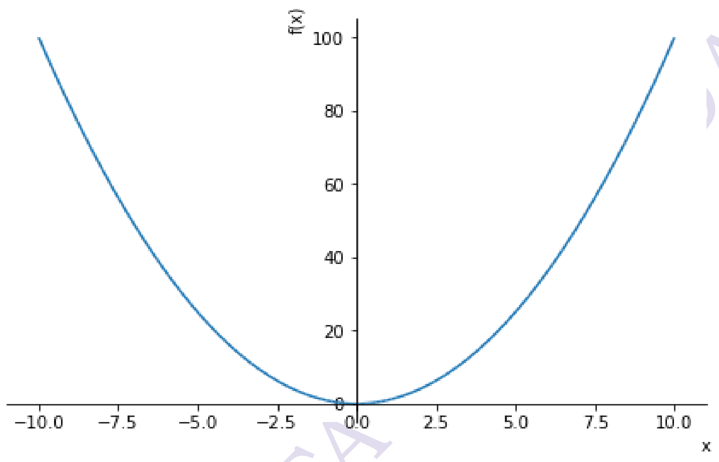
2.2.1. Basic Plotting: Use the plot function to visualize a mathematical expression or function.

Example(Default range of basic plot):

SymPy has a plot module for visualizing mathematical functions. This feature can help scientists and mathematicians gain insights into function behaviors.

Code:
```
from sympy import *
x=Symbol('x')
plot(x**2)
```

Output:

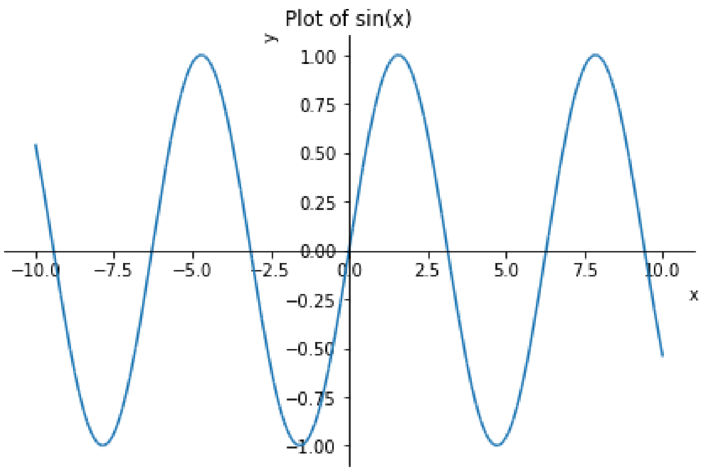


This code generates a plot of the function $f(x) = x^2$ from -10 to 10. Visualizations like this are essential in fields like physics, where plotting potential energy functions reveals equilibrium points, or in statistics to view distribution curves.

Example (Basic Plotting):

Code:
```
from sympy import symbols, plot, sin
x = symbols('x')
plot(sin(x), (x, -10, 10),
title="Plot of sin(x)", xlabel="x",  ylabel="y")
```

Output: The following plot is obtained of the sine function over the range [−10,10].

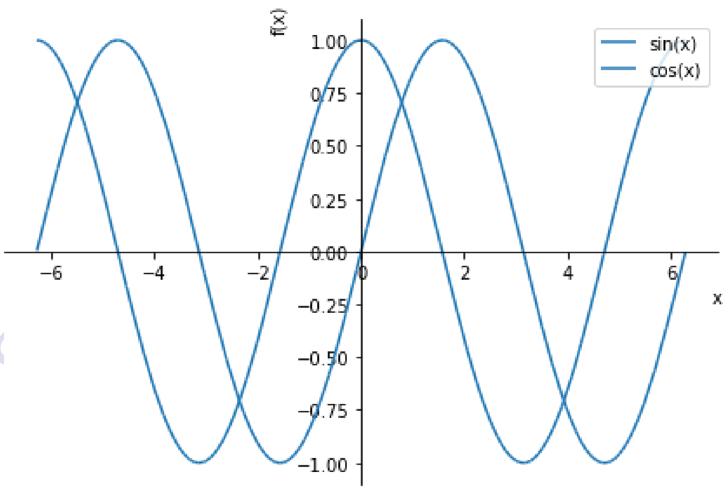


2.2 Plotting of Several Functions: We can plot multiple functions together for comparison.

Example (Plotting of Several Functions):

Code:
```
from sympy import cos
plot(sin(x), cos(x), (x, -2*3.14, 2*3.14), legend=True)
```

Output: A single graph showing both sin(x) and cos(x) over the range [−2π,2π].

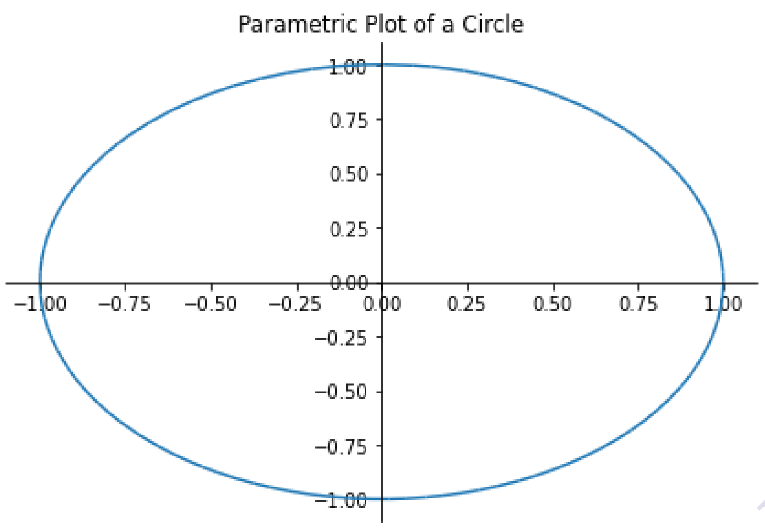


Parametric Plotting: For parametric equations, SymPy provides plot_parametric.

Example (Parametric Plotting):

Code:
```
from sympy import cos, pi
t = symbols('t')
plot_parametric(cos(t), sin(t), (t, 0, 2*pi),
title="Parametric Plot of a Circle")
```

Output: A circular parametric plot.

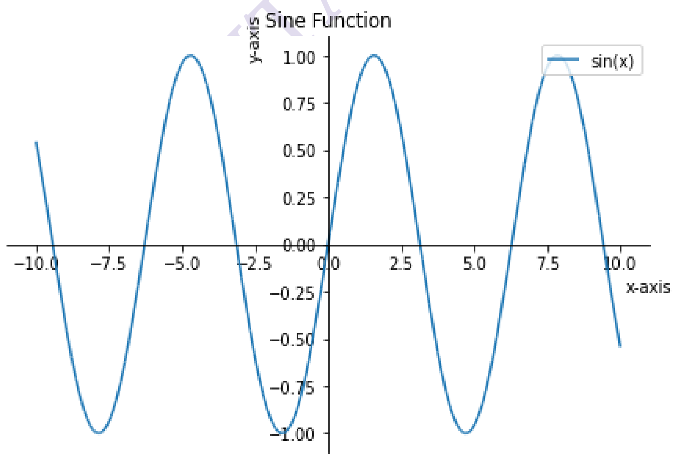


2.3. Features and Customization

(a) Customizing the Plot: We can add titles, labels, and legends to enhance your plot:

Code:
```
plot(sin(x), (x, -10, 10), title="Sine Function",
xlabel="x-axis", ylabel="y-axis", legend=True)
```

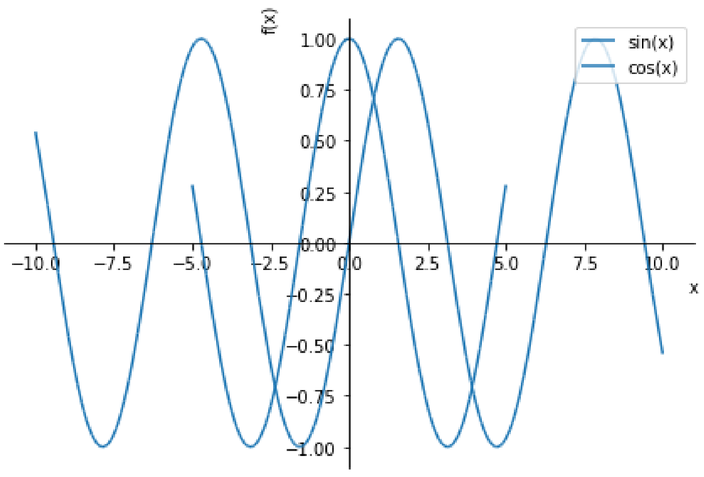
Output:



(b) Multiple Ranges: Plotting functions with different ranges:

Code: `plot((sin(x),(x,-10,10)),(cos(x),(x,-5,5)),legend=True)`

Output:



2.4. Advantages of SymPy Plotting

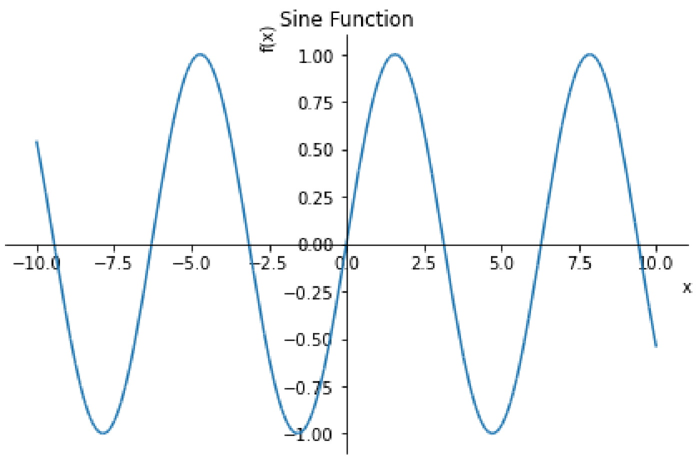
1. **Integration with Symbolic Expressions**: Since SymPy is a symbolic mathematics library, its plotting tools are tightly integrated with symbolic computations.
2. **No Need for External Libraries**: Basic visualization tasks can be accomplished without relying on external plotting libraries like Matplotlib.
3. **Dynamic Visualizations**: Functions and expressions can be visualized dynamically by simply modifying the symbolic expressions or ranges.

Example-1:

Code:

```
from sympy import symbols, plot, sin
# Define the variable
x = symbols('x')
# Plot a single function
p1=plot(sin(x),(x,-10,10),show=False,title="Sine Function")
p1.show()
```

Output:

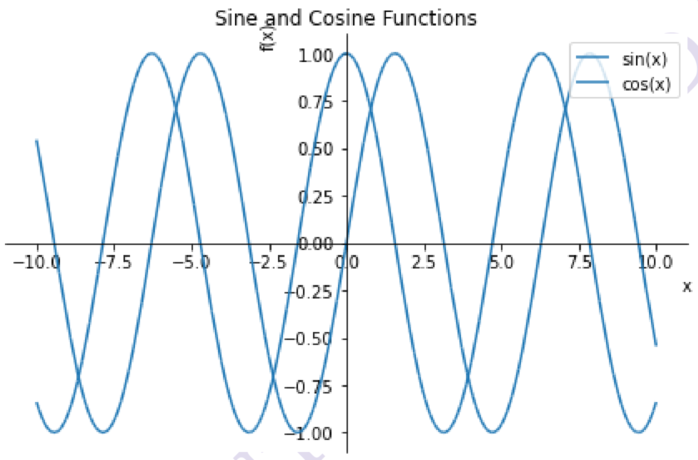


Example-2

Code:

```
from sympy import symbols, plot, sin, cos,
# Define the variable
x = symbols('x')
# Plot multiple functions
p2 = plot(sin(x), cos(x), (x, -10, 10), legend=True,
show=False)
p2.title = "Sine and Cosine Functions"
p2.show()
```

Output:



2.5. Plotting Expressions Input by the User

Using SymPy, we can dynamically accept user input for mathematical expressions, convert them into symbolic expressions, and then plot them. This method utilizes SymPy's `sympify` function to safely transform user-provided strings into symbolic expressions. This following script demonstrates how to use SymPy to create dynamic plots of user-defined functions, offering an interactive and flexible way to visualize mathematical concepts programmatically.

Code:

```
from sympy import symbols, sympify, plot
# Step 1: Define the variable
x=symbols('x')
# Step 2: Take user input for the mathematical expression
my_input=input("Enter a mathematical expression in terms of x
(e.g., sin(x), x**2 + 3*x - 5): ")
try:
# Step 3: Convert the string input into a symbolic expression
expr=sympify(my_input)
```

```
# Step 4: Plot the expression
print("Plotting the expression...")
plot(expr, (x, -10, 10), title=f"Plot of {my_input}",
xlabel="x-axis", ylabel="y-axis")
except Exception as e:
# Handle invalid input
print(f"Error:{e}.Please enter a valid mathematical
expression.")
```

Explanation of the above code:

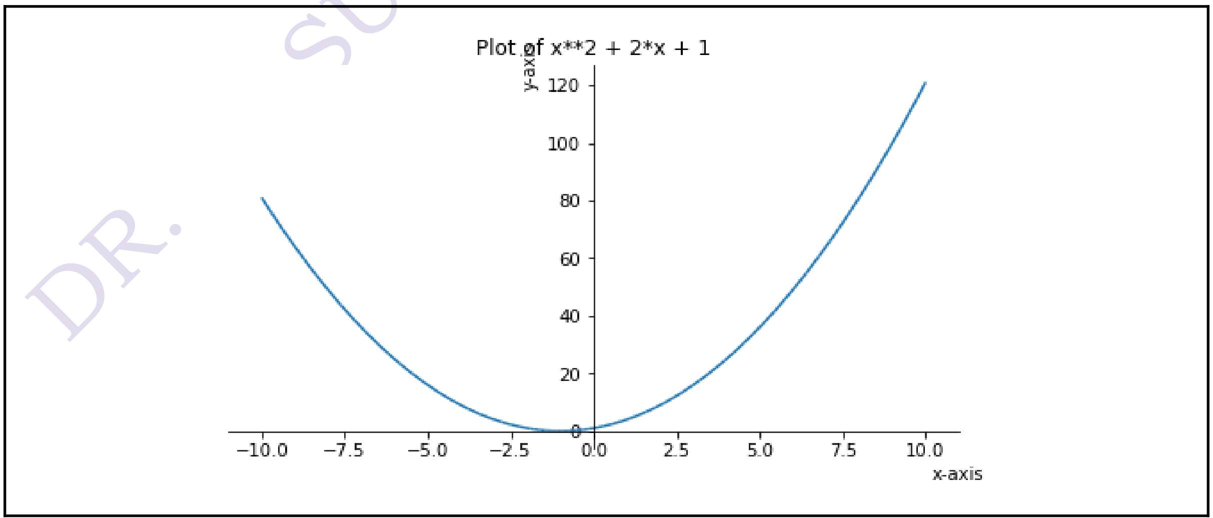
- A symbolic variable x is defined using `symbols('x')`. This variable will represent the independent variable in the user-defined expression.
- The input function is used to take the mathematical expression as a string (e.g., `x**2 + 3*x - 5`).
- The sympify function converts the string input into a SymPy symbolic expression. This function handles mathematical expressions while ensuring they are safely parsed.
- The plot function from SymPy is used to visualize the parsed expression over a default range or a custom range (in this case, −10 to 10).
- Error Handling: If the user inputs an invalid expression, an exception is raised and caught to provide meaningful feedback.

Hands -On Examples with user inputs:

Input:

Enter a mathematical expression in terms of x: `x**2 + 2*x + 1`

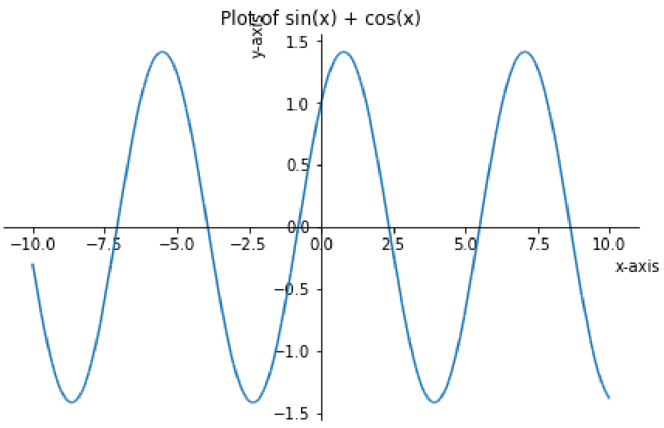
Output: A plot (shown in the following figure) of the quadratic function $x^2 + 2x + 1$ over the range [−10,10].



Input:

Enter a mathematical expression in terms of x: `sin(x) + cos(x)`

Output: A plot (shown in the following figure) showing sin(x)+cos(x) over the range [−10,10].



Input (Invalid): Enter a mathematical expression in terms of x: `x + *`

Output:

```
Error: Sympify of expression 'could not parse 'x + *'' failed, because of exception being raised:
SyntaxError: unexpected EOF while parsing (<string>, line 1).Please enter a valid
mathematical expression.
```

Customization of the Script

1. **Range Input**: Allow the user to specify a custom range for plotting.

```
 x_min = float(input("Enter the minimum value of x: "))
x_max = float(input("Enter the maximum value of x: "))
plot(expr, (x, x_min, x_max), title=f"Plot of {my_input}")
```

2. **Several Expressions**: Allow the user to input and plot multiple expressions simultaneously.

```
expressions = input("Enter mathematical expressions separated
               by commas: ").split(',')
plots = [sympify(expr.strip()) for expr in expressions]

plot(*plots, (x, -10, 10))
```

Plotting with Customization Options

Here we demonstrate how SymPy's plotting module provides powerful customization options, making it easy to create clear and visually appealing plots directly from symbolic expressions.

Here is an example of plotting with **SymPy** using a similar approach, where we customize the colors of multiple plots and include a legend:

Code Example

```
from sympy import Symbol
from sympy.plotting import plot
# Define the symbolic variable
x = Symbol('x')
# Create the plot with multiple expressions
p = plot(2*x + 3,  3*x + 1, legend=True, show=False)
# Customize the colors of each plot
p[0].line_color = 'b'  # Blue for the first line
p[1].line_color = 'r'  # Red for the second line
# Add legend labels for clarity
p[0].label = 'y = 2x + 3'
p[1].label = 'y = 3x + 1'
# Show the plot
p.show()
```

Brief explanation of the above lines of code:

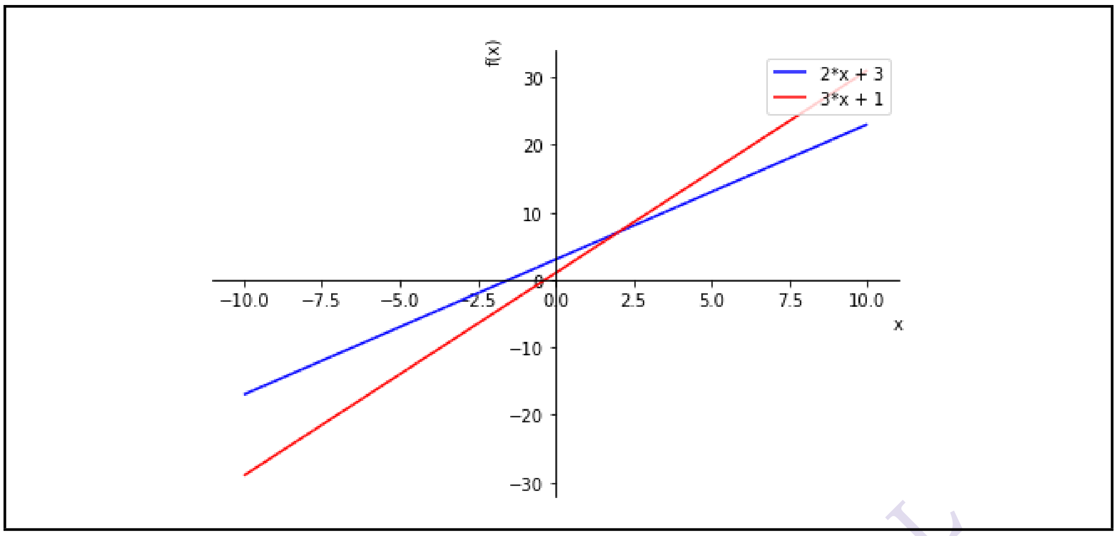
- A symbolic variable x is created using `Symbol('x')`.
- The plot function is used to plot the two linear expressions `2x+3` and `3x+1`.
- The `legend=True parameter is used to enable legends for the plots.`
- `show=False` allows customization of the plot before it is displayed.
- The `line_color` property of each plot object is set to '`b`' (blue) and `'r'` (red), respectively.
- `Labels` are assigned to each plot using the label attribute to make the legend meaningful.
- `p.show()` shows the plot with the customized settings.

Output:

The output will be a graph showing:

A blue line representing y=2x+3.

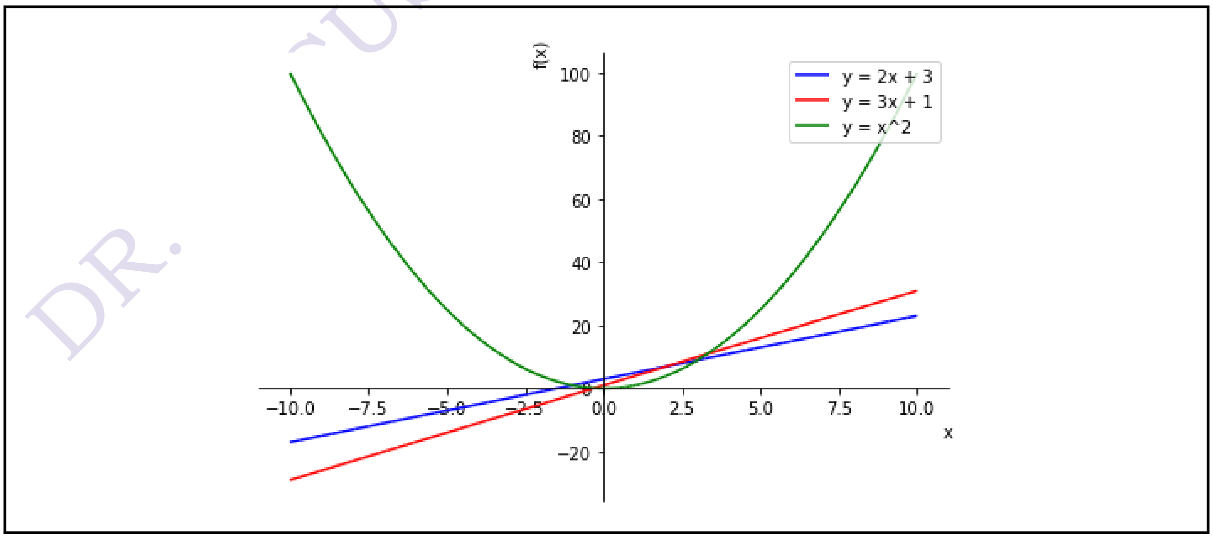
1. A red line representing y=3x+1.
2. A legend identifying each line by its equation.



Now we will extending the above example by adding more functions:

code
```
p = plot(2*x + 3, 3*x + 1, x**2, legend=True, show=False)
p[0].line_color = 'b' # Blue
p[1].line_color = 'r' # Red
p[2].line_color = 'g' # Green for the quadratic function
p[0].label = 'y = 2x + 3'
p[1].label = 'y = 3x + 1'
p[2].label = 'y = x^2'
p.show()
```

Output:



The following table shows a list of **plotting commands** in SymPy:

Table: A list of plotting commands in SymPy

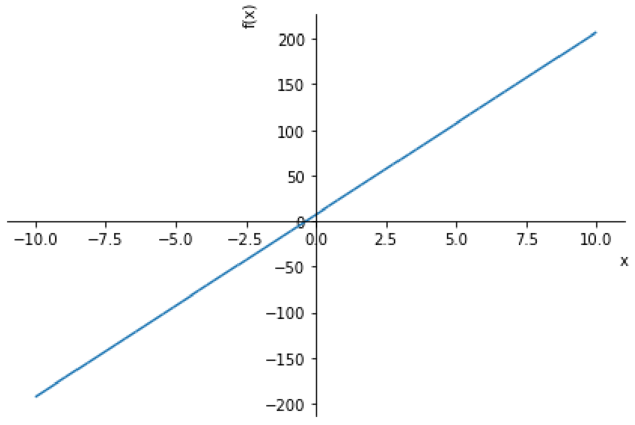
Operation	Description	SymPy Function
2D Plot	Plots a single-variable function	plot(f, (x, a, b))
Multiple Functions Plot	Plots multiple functions on the same graph	plot(f1, f2, (x, a, b))
Parametric Plot	Plots a parametric curve	plot_parametric((x_expr, y_expr), (t, a, b))
3D Surface Plot	Plots a 3D surface	plot3d(f,(x,a,b),(y,c,d))
3D Parametric Surface	Plots a parametric 3D surface	plot3d_parametric_surface(x_expr, y_expr, z_expr, (u, a, b), (v, c, d))
3D Parametric Line	Plots a 3D parametric curve	plot3d_parametric_line((x_expr, y_expr, z_expr), (t, a, b))
Contour Plot	Plots contour lines of a function	plot_contour(f, (x, a, b), (y, c, d))
Implicit Plot	Plots an implicit function F(x,y)=0	plot_implicit (Eq(f(x, y), 0), (x, a, b), (y, c, d))
Vector Field Plot	Plots a vector field	plot_vector_field ((fx,fy),(x, a, b), (y, c, d))
Customizing Plots	Modify labels, titles, and colors	plot(f, xlabel="X-axis", ylabel="Y-axis", line_color='red')

2.6 Worked Out Examples

Example - 1: Plot the linear equation $y = 20x + 7$ using SymPy with minimal setup and default settings.

Solution:

Code:
```
from sympy import Symbol
x = Symbol('x')
plot(20*x+7)
```
Output:



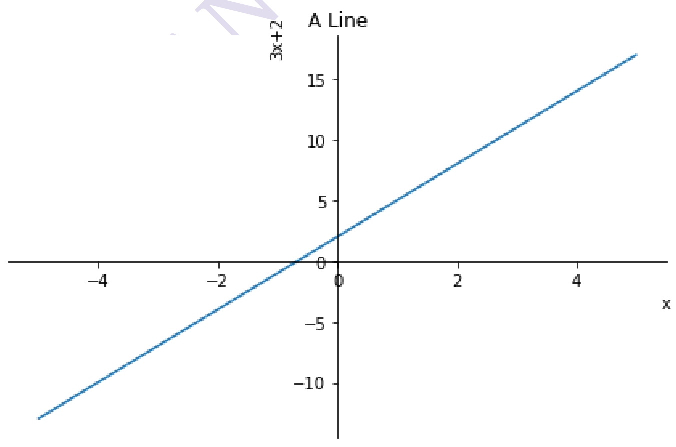
Example - 2: Plot of the linear equation $y = 3x + 2$ over the range $-5 \leq x \leq 5$ with the title "A Line", label the x-axis as "x" and the y-axis as "3x+2", without displaying the plot immediately in SymPy.

Solution:

Code:

```
from sympy import *
x = symbols('x')
r = plot(3*x + 2, (x, -5, 5), title="A Line", xlabel='x',
ylabel='3x+2', show=False)
r.show()
```

Output:



Example - 3: Plot the following functions using SymPy on the same graph over the interval (-1, 1):

(a) $y = x^2 + 5x - 8$; **(b)** $y = cos(3x)$;

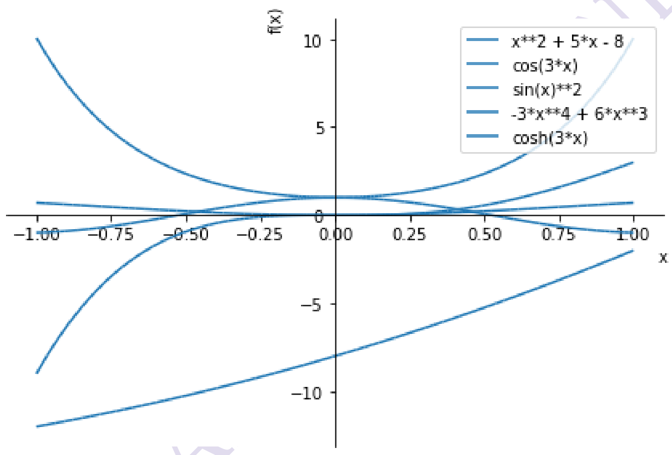
(c) $y = sin^2(x)$; **(d)** $y = 6x^3 - 3x^4$;

(e) $y = cosh(3x)$.

Solution:

```
from sympy import *
# Define the symbolic variable
x = symbols('x')
# Define the functions
f1 = x**2 + 5*x - 8       # y = x^2 + 5x - 8
f2 = cos(3*x)             # y = cos(3x)
f3 = sin(x)**2            # y = sin^2(x)
f4 = 6*x**3 - 3*x**4      # y = 6x^3 - 3x^4
f5 = cosh(3*x)            # y = cosh(3x)
# Plot all functions in a single figure
p = plot(f1,f2, f3,f4,f5,
      (x, -1, 1),  # Define x range
      show=True,
     legend=True)
```

Output:



Example - 4: How can you create a plot of the linear equation $y = 2x + 3$ over the range $-5 \leq x \leq 5$, set a title and axis labels without displaying the plot immediately in SymPy. Save the plot as an image file named 'line.png' and display the plot using SymPy.

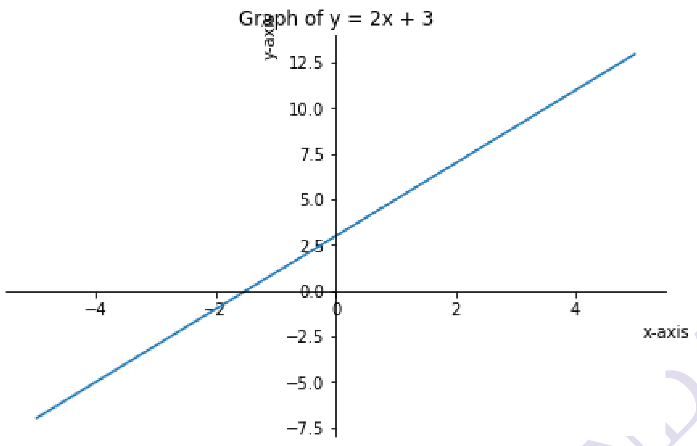
Solution:

Code:

```
from sympy import Symbol
from sympy.plotting import plot
# Define the symbolic variable
x = Symbol('x')
# Define the function y = 2x + 3
y = 2*x + 3
# Create the plot
p = plot(y, (x, -5, 5), show=False, title="Graph of y = 2x
+ 3", xlabel="x-axis",ylabel="y-axis")
# Save the plot as an image file
```

```
p.save('line.png')
# Show the plot
p.show()
```

Output:

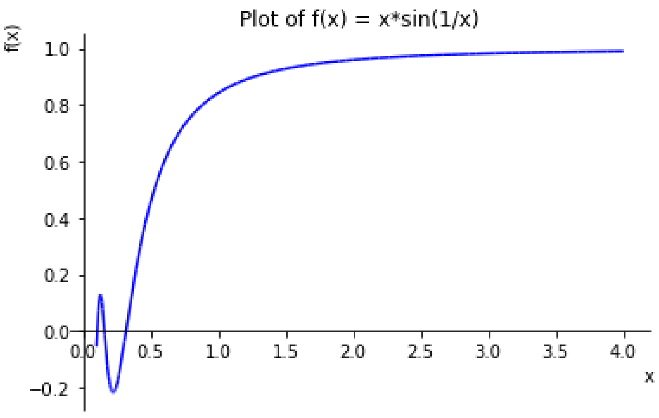


Example - 5: Plot the function $f\ (x) = xsin\,\frac{1}{x}$ using SymPy in the interval [0, 4]**. Save this image file named 'xsin.pdf' using SymPy.**

Solution:

```
import sympy as sp
# Define the symbolic variable and function
x = sp.symbols('x')
f = x * sp.sin(1/x)
# Plot the function in the interval [0, 4] and save as PDF
p = sp.plot(f, (x, 0.1, 4), show=False, title="Plot of f(x) = x*sin(1/x)")
p[0].line_color = 'b' # Set line color to blue
 p.save("xsin.pdf") # Save the plot as 'xsin.pdf'
```

Output: The following output is saved in the current directory as xsin.pdf.

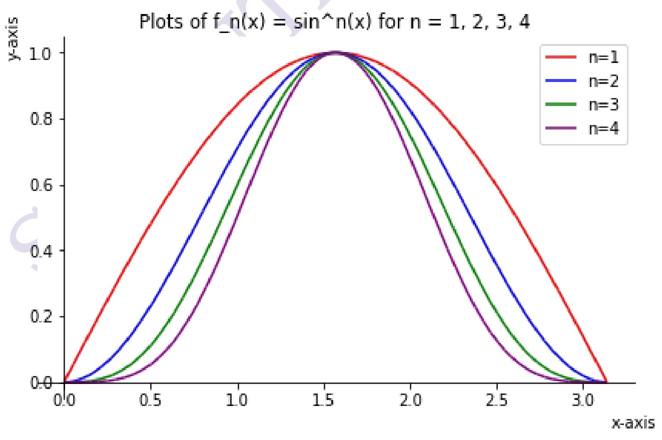


Example - 6: Plot the functions $f_n(x) = sin^n x$ using SymPy in the interval [0, π] for n= 1, 2, 3, 4. Use different colors for different plots.

Solution:

```
from sympy import *
# Define the symbolic variable
x = Symbol('x')
# Define the functions f_n(x) = sin^n(x) for n = 1, 2, 3, 4
functions = [sin(x)**n for n in range(1, 5)]
# Define colors for different plots
colors = ['red', 'blue', 'green', 'purple']
# Create the plot
p = plot(*functions, (x, 0, pi), show=False, title="Plots of
f_n(x) = sin^n(x) for n = 1, 2, 3, 4", xlabel="x-axis",
ylabel="y-axis", legend=True)
# Assign colors and labels to each function
for i, func in enumerate(functions):
    p[i].line_color = colors[i]
    p[i].label = f"n={i+1}"
# Show the plot
p.show()
```

Output:



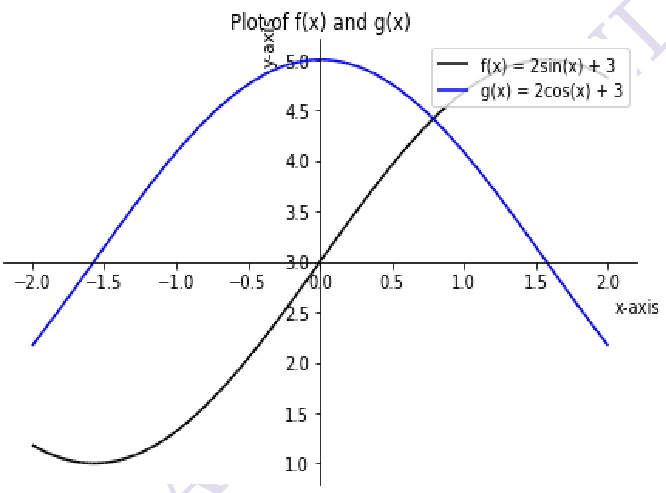
Example - 7: Plot the functions $f(x) = 2sin\, x + 3$ and $g(x) = 2cos\, x + 3$ on the same graph over the interval (-2, 2) in black color for $f(x)$ and in blue color for $g(x)$. Label both the graphs for more clarity.

Solution:

```
from sympy import *
# Define the symbolic variable
x = Symbol('x')
```

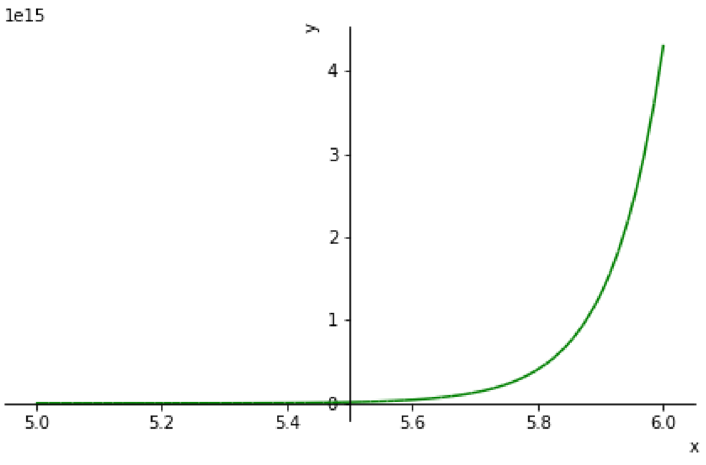
```
# Define the functions
f = 2*sin(x) + 3
g = 2*cos(x) + 3
# Plot with custom colors
p = plot(f, g,  (x, -2, 2), show=False,  title="Plot of f(x) and g(x)",  xlabel="x-axis",
        ylabel="y-axis",  legend=True)
# Set colors for the functions
p[0].line_color = 'black'  # f(x) in black
p[0].label = "f(x) = 2sin(x) + 3"
p[1].line_color = 'blue'   # g(x) in blue
p[1].label = "g(x) = 2cos(x) + 3"
p.show()
```

Output:



Example - 8: Plot the functions $f(x) = 2\,log\,x + e^{x^2} + sin^{-1}x$ in the range (1, 6) with green color and with proper x-label and y-label.
Solution:

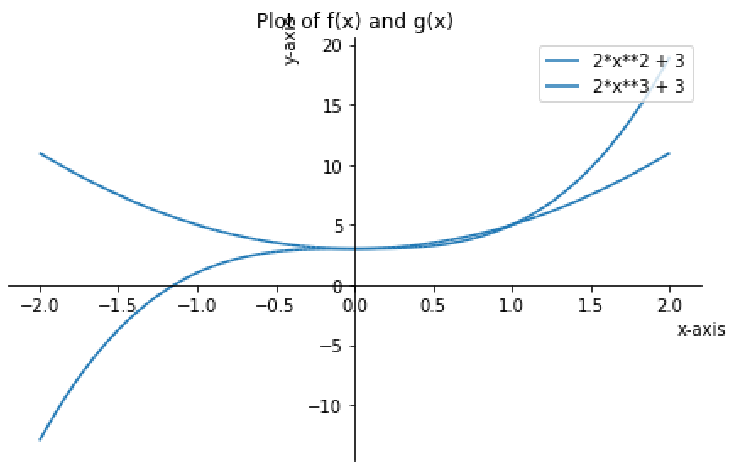
Code:
```
from sympy import *
x = symbols('x')
f=2*log(x)+exp(x**2)+asin(x)
r = plot(f,(x,1,6),line_color='g', xlabel='x',
ylabel='y',show=False)
r.show()
```
Output:



Example - 10: Write code using SymPy to plot the functions $f(x) = 2x^2 + 3$ and $g(x) = 2x^3 + 3$ on the same graph over the interval [-2, 2]. Also set a title and axis labels.
Solution:

```
from sympy import *
# Define the symbolic variable
x = Symbol('x')
# Define the functions
f = 2*x**2 + 3
g = 2*x**3 + 3
# Plot the functions with custom line styles
p = plot(f, g,  (x, -2, 2),
title="Plot of f(x) and g(x)",
xlabel="x-axis",
ylabel="y-axis",
show=True, legend=True)
# Set labels for the functions
p[0].label = "f(x) = 2x² + 3"
p[1].label = "g(x) = 2x³ + 3"
```

Output:



Exercise - II

1. How do you import the plot function from SymPy?
2. What is the basic syntax to plot a single-variable function using SymPy?
3. How can you plot multiple functions on the same graph in SymPy?
4. How do you add a title and labels to a plot in SymPy?
5. What is the difference between plot and plot_parametric in SymPy?
6. How can you specify a custom range for the variable while plotting in SymPy?
7. How do you change the color and style of a plotted function in SymPy?
8. How can you plot a piecewise function using SymPy?
9. What is the purpose of plot_implicit in SymPy, and how is it different from plot?
10. How can you use legend=True in a SymPy plot?
11. Write a Python program to plot the function $f(x) = x^2$ over the range $x \in [-5, 5]$ using *SymPy*. Modify the plot to include labels for the x-axis and y-axis.
12. Plot the function $f(x) = e^{-x}sin(2x)$ over the interval $x \in [-2, 2]$ and set a title for the plot. Save the plot as a PNG file and display the saved image.
13. Plot the parametric equations $x = cos(t)$, $y = sin(t)$ to visualize a unit circle.
14. Use plot_implicit to graph the equation $x^2 + y^2 = 4$ and show the output.
15. Write *SymPy* command to plot the graph of x^3 in $[-4, 4]$ with the following options

    a) give a title "This is a graph of $y = x^3$"

    b) line color is purple

    c) fixed the size of the graph
16. Write *SymPy* command to plot the graph of $f(x) = e^{x^2} + ln\left(x^2 + 2\right) - 0.6(x + 1)$ in $[-1, 5]$ with color and thickness features.
17. Write *SymPy* command to plot the graph of $f(x) = \frac{sinh\left(1-x^2\right) + cosh\left(1-x^2\right)}{1-x+x^2}$ in $[-1, 1]$ with color, frame and axes label.
18. Write *SymPy* command to plot the ellipse $\frac{x^2}{2} + \frac{y^2}{4} = 10$.
19. Write *SymPy* code to plot the graph of $r = 1 + 2cos\, 3\theta$ in $[0, 2\pi]$.
20. Write *SymPy* commands which plots the graph of the parametric curve $x = sin\, t$, $y = cos\, (t + sin\, (3t))$ in $[0, 2\pi]$ and display the image.
21. Write *SymPy* command to plot the graph of $y = xsin\, \frac{1}{x}$ in $[-0.1, 0.1]$ with specified size, color, aspect ratio, labelling axes, thickness and grid.
22. Write *SymPy* command to plot the graph of following function in one plot with different color and appropriate legend

a) $f(x) = e^{-\frac{x^2}{2}} sin\,(2x)$ in $[-\ 3,\ 3]$

b) $g(x) = e^{-\frac{x^2}{2}} sin\left(\frac{x}{2}\right)$

c) $h(x) = e^{-\frac{x^2}{2}} sin\left(\frac{x^2}{2}\right)$

d) one big dot at $(1,\ 2)$ with size 50.

23. Write *SymPy* command to plot the graph of $f(x) = x^{-sin\,x} + log\,e^{-x^2}$ in $[-\ 1,\ 1]$ with title " A plot" with position $(-\ 0.5,\ 0.1)$.

24. Write *SymPy* command to draw and shade the region enclosed by the curves $y = \frac{x+1}{x^2+3}$ & $y = x^4 - x$.