GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN IN PUBLICA COMMODA
SEIT 1737

Project Report

Expression Parsing

Humaira Azam

Matriculation No: 26210266

Supervisor: Dr. Igor Voulis

Georg-August-Universität Göttingen
Institute of Computer Science

May 18, 2024

Abstract

In this project we have built an expression parser that evaluate and process arithematic and polynomial expressions. The parser take the arithematic expression, matries and polynomial as string and solve them accordingly to create the valuable output.

The Expression parser [5] is built on python. The python library Numpy is used to leverage mathematical operations. NumPy's efficient mathematical operations, including arithmetic functions like addition, subtraction, multiplication, division, and exponentiation, are leveraged within the parser. Additionally, NumPy provides essential trigonometric functions such as ‘sin‘, ‘cos‘, ‘tan‘, and ‘sqrt‘. It is also used in matrix evaluation.

The Shunting Yard algorithm [4]used to convert infix expressions into postfix notation, ensuring accurate evaluation of arithmetic operations while respecting the order of operations. The regular expression method helps in identification of operands, operators, and functions also helps in validating the characters in expression.

The integration of arithmetic operations[5] enables seamless handling of basic mathematical expressions, ensuring correct evaluation and computation of results. For example, expressions like "2 + 3 * sin(0)" are parsed and evaluated accurately, maintaining precedence and yielding correct outcomes.

More over the expression parser also supports the polynomial operations like finding the roots of polynomial, computing derivative of polynomial or solving the expressions with the polynomial.

Furthermore, the code architecture prioritizes clarity and maintainability, with distinct functionalities encapsulated within a dedicated Python library[8]. Extensive numerical tests conducted within a Jupyter Notebook, accompanied by detailed explanations, validate the robustness and reliability of the implemented functionalities.

In summary, this Python-based mathematical expression parser offers a versatile and efficient solution for parsing, evaluating, and computing mathematical expressions, enhancing productivity and facilitating numerical computations in various applications.

Introduction

In computational mathematics, accurately reading and evaluating mathematical expressions is essential for various tasks. From simple calculations to complex polynomial operations, a dependable parser is crucial for computational analysis. Whether solving equations, simulating systems, or analyzing data, mathematical expressions play a vital role in scientific and engineering fields.[2][6] However, the complexity of mathematical expressions introduces challenges in programming. Unlike natural language, which allows for ambiguity resolution through context, mathematical expressions require strict adherence to specific syntax and semantics. A single misplaced operator or missing parenthesis can alter the interpretation and outcome of an expression.[9][11]

Challenges in Mathematical Expression Parsing

Analysis of mathematical terms presents many challenges arising from the complexity and diversity of mathematical notation. Unlike natural language analysis, where ambiguity is common and context contributes to its clarification, mathematical terminology requires unambiguous interpretation based on defined syntactic and semantic rules. One of the primary challenges in mathematical expression parsing is the presence of infix notation, where operators appear between operands (e.g., "3 + 4"). Converting infix expressions to a more computationally efficient form, such as postfix or prefix notation, is a fundamental task in parsing. Techniques like the Shunting Yard algorithm [4]provide a systematic approach to this conversion, enabling parsers to handle complex expressions with ease.

Another challenge arises from the diverse range of mathematical operations and functions that expressions may encompass. From elementary arithmetic operations like addition and multiplication to transcendental functions like sine and logarithm, parsers must be equipped to handle a broad spectrum of mathematical constructs. Moreover, the introduction of variables, constants, and trigonometric functions further complicates the parsing process, necessitating robust mechanisms for symbol resolution and evaluation.

Objectives of the Project

In light of the aforementioned challenges and the pivotal role of mathematical expression parsing, this project aims to develop a mathematical expression parser in Python. The primary objectives of the project are as follows:

1. Implement Basic Mathematical Operations: Build a parser that can handle basic mathematical operations such as addition, subtraction, multiplication, and division. The primary objective is to setup the foundation to ensures that the parser can accurately handle simple mathematical expressions.

2. Advanced Mathematical Operations and Error Handling: Enhance the parser to support more complex mathematical operations, including exponentiation, trigonometric functions, and logarithms. Additionally, integrate robust error handling mechanisms to detect and manage syntactical inaccuracies in input expressions, ensuring the reliability and stability of the parser.

3. Incorporate Support for Matrices and Matrix Operations: Extend the parser's capabilities to include support for matrix expressions and operations. This expansion

broadens the parser's applicability, enabling it to handle diverse computational tasks involving matrices, such as matrix multiplication, determinant calculation, and matrix inversion.

4. Enhance to Handle Polynomial Expressions: Further enrich the parser to handle polynomial expressions, including functionalities such as evaluating polynomial degrees, computing derivatives, and solving polynomial equations. These additions empower the parser to tackle a wide range of mathematical problems, from curve fitting to root finding, thereby enhancing its utility in diverse scientific and engineering applications.

Technical Approach and Tools

The successful realization of the project objectives relies on leveraging Python's versatility and powerful libraries, along with employing sophisticated parsing algorithms and mathematical techniques. The following technical tools and approaches will be utilized in the implementation of the mathematical expression parser:

- Shunting Yard Algorithm: Utilize the Shunting Yard algorithm to convert infix mathematical expressions into postfix notation, facilitating efficient parsing and evaluation.[4]
- Regular Expressions (Regex): Employ regular expressions for input validation and syntactical correctness, ensuring that input expressions adhere to predefined patterns and constraints.[10]
- NumPy for Mathematical Computations: Harness the capabilities of the NumPy library for efficient handling of basic mathematical operations and matrix computations, enabling optimized performance and scalability.[7]
- Newton-Raphson Method for Polynomial Solving: Implement the Newton-Raphson method for solving polynomial equations, enabling the parser to find the roots of polynomial expressions with precision and efficiency.[1]

Programming Tools

Python, renowned for its simplicity, versatility, and rich ecosystem of libraries, serves as the primary programming language for implementing the mathematical expression parser[5]. Its intuitive syntax and extensive library support make it an ideal choice for developing complex mathematical algorithms and applications.

Utilizing the NumPy Library

NumPy, a fundamental library for numerical computing in Python, plays a pivotal role in the implementation of the mathematical expression parser. By providing high-performance multidimensional array objects and a vast array of mathematical functions, NumPy enables efficient handling of numerical data and computation. Leveraging NumPy's array-based approach, the parser can perform vectorized operations, leading to improved performance and scalability when dealing with large datasets or complex computations. Additionally, NumPy's integration with other scientific computing libraries, such as SciPy and Matplotlib, further enhances the capabilities of the parser, enabling seamless integration with other tools and workflows in the scientific computing ecosystem.[7]

1 Shunting Yard Algorithm

Generally in mathematics, The expression is written in infix notation

For example: "A + (B * C)"

where the BODMAS rule is applied to solving the equation however parsing an infix expression is more hectic and requires resources for computer to solve.
For example, in the expression "A + B * C," multiplication has a higher precedence than addition. So, when evaluating this expression, we would first multiply the values of B and C and then add the result to A. This order of operations is something humans easily understand, but for a computer or machine to perform the calculation correctly, it needs clear instructions about the precedence of each operator.

In 1961, Edsger Dijkstra introduced the "Shunting Yard" algorithm. The Shunting Yard algorithm is a technique that transforms mathematical expressions written in infix notation ("3 + 4 * 2") into Reverse Polish Notation (RPN) or post fix notation ("3 4 2 * +"). RPN is more straightforward and easier to compute because it eliminates the need for parentheses to indicate the order of operations.[4]

The algorithms primarily uses stack structure for storing operators during the operations. The reason for using the stack is to switch the order of the operators in the expression. Also, the stack acts like a storage place. It holds the operators until both numbers that the operator works on are seen. Only then can the operator be shown.[12]

1.1 Infix to Postfix Conversion

In simpler terms, infix notation is the way we typically write mathematical expressions on paper, and it's intuitive for humans. However, when translating these expressions for a computer, we use algorithms like the Shunting Yard algorithm to ensure that the order of operations is correctly understood and followed during the calculation.

When dealing with expressions like $(A+B) \times C$, where A is first added to B and then the sum is multiplied by C, writing an algorithm to parse and evaluate such expressions in infix notation can be quite tedious. This is because you'd need to go through the expression multiple times to figure out which operation to perform first.

Imagine reading the expression like a computer: you'd have to check for parentheses, determine what's inside them, perform addition, and finally, multiply the result by C. As the number of operations increases, the complexity of figuring out the correct order of operations also grows. It becomes a step-by-step process that requires careful tracking of each operation and its operands.

To simplify this for a computer, algorithms like the Shunting Yard algorithm are used. These algorithms help convert infix expressions into a format (postfix notation) that is easier for the computer to evaluate in a linear manner. This way, the computer can follow a straightforward sequence of operations without repeatedly revisiting the expression, making the parsing and evaluation process more efficient.[4]

In postfix notation, also called Reverse Polish Notation (RPN), operators follow their corresponding operands. For example, the infix expression "$A+B \times C$" becomes "$ABC \times +$" in postfix notation.

What's great about postfix is that there are no parentheses, and you don't need to worry about operator precedence. This simplicity makes it easy for a computer to evaluate expressions because the order in which the operators should be applied is fixed.

1.2 Working Steps of Shunting Yard Algorithm

When solving the mathematical expression, Order of operation is very important for example in $A + B \times C$, where multiplication has higher precedence than addition. The Shunting Yard algorithm ensures that the correct order of operations is maintained. For a computer or machine to perform the calculation accurately, it needs explicit instructions about the precedence of each operator.[4]

The algorithm employs a *stack*, a container that stores operators rather than numbers. The stack is crucial for changing the order of operators in the expression and acts as a storage mechanism. Operators are held in the stack until both operands that the operator works on are encountered, allowing the operator to be processed. [13]Following are the detailed working steps of shunting yard algorithm:

1. Initialize an empty stack for operators and an empty queue (or output) for operands and operators in postfix order.
2. Iterate through each token (operands, operators, and parentheses) in the infix expression from left to right.
3. If the token is an operand (a number or variable), add it to the output queue.
4. If the token is an operator:
   - If the stack is empty, push the operator onto the stack.
   - If the stack is not empty, compare the precedence of the current operator with the operator at the top of the stack:
     - If the current operator has higher precedence than the operator at the top of the stack, push the current operator onto the stack.
     - If the current operator has lower precedence than or equal precedence to the operator at the top of the stack:
       - Pop operators off the stack onto the output queue until:
         - The stack is empty.
         - An open parenthesis is encountered.
         - An operator with lower precedence than the current operator is encountered.
         - An operator with equal precedence to the current operator is encountered, and the current operator is left-associative.
       - Push the current operator onto the stack.
     - If the current operator is a right-associative operator with equal precedence to the operator at the top of the stack, push it onto the stack without popping the operator from the stack.

5. If the token is an open parenthesis, push it onto the stack.
6. If the token is a closing parenthesis:
    - Pop operators off the stack onto the output queue until an open parenthesis is encountered. Discard the open parenthesis.
    - If no open parenthesis is found while popping operators, it indicates a mismatched parenthesis error.
7. Repeat steps 3-6 until all tokens have been processed.
8. Pop any remaining operators off the stack onto the output queue.

The resultant postfix expression is further processed by system using stack.

1.3 Evaluating Postfix Expressions Using a Stack

The next step in parsing the mathematical expression is to evaluate the postfix expression. Following are the steps to evaluate the postfix expression using a stack:

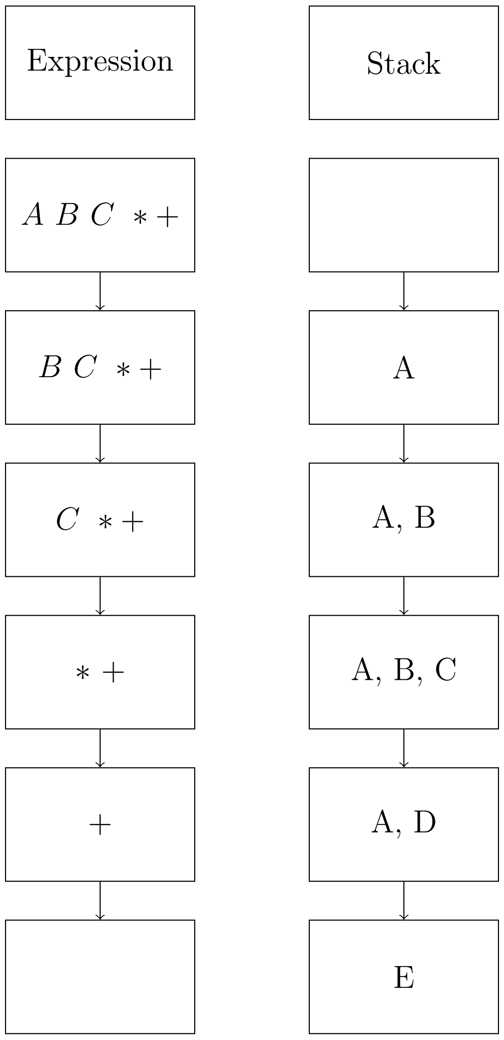
1. Read the postfix expression token by token.
2. If the token is a number (operand), put it on the stack.
3. If the token is an operator, take the top two numbers from the stack, apply the operator, and put the result back on the stack.

For example, let's evaluate "$A\ B\ C\ *+$":

The final result remains on the stack. As you can see, evaluating a postfix expression is straightforward because the order of operations is predetermined.

1.4 Extension for Matrix and Polynomials and Trigonometric Operations

Primarly the algorithm widely applied to parse the expression where the operands are constant. However we have further extended the implementation to add the support to use Matrix as an operands. We have also added the computation of trigonometric operation as well as using a polynomial in the expression.



Here, $D = B \times C$ and $E = A + D$. The final result, represented by E, remains on the stack.

2 Source Code Documentation

2.1 Code Structuring

Modularity:

The classes are structured into distinct modules, each dedicated to a specific aspect of expression parsing. This modular organization fosters maintainability and facilitates seamless expansion as new features or operators are introduced.

Encapsulation:

The inner workings of the classes are encapsulated, abstracting away the intricacies of expression parsing from the user. This encapsulation fosters a clear and intuitive interface, shielding users from unnecessary implementation complexities.

Decoupling:

Modules within the class are designed with loose coupling, minimizing interdependencies between components. This decoupling enhances the class's adaptability, simplifying the modification or extension of individual functionalities without impacting the entire system.

Consistency:

The class maintains uniform naming conventions and coding styles throughout, promoting readability and accessibility for developers of varying expertise levels. Consistency plays a pivotal role in upholding code quality and facilitating collaborative development efforts.

2.2 Code Files Overview

1. **ExParser Class:** This class contains methods to evaluate and solve expressions and perform basic operations.
2. **PolySolver Class:** The PolySolver class is responsible for solving complex polynomial problems, such as finding derivatives.
3. **Python Notebook:** This notebook file serves as the implementation and testing environment for the operations described in the ExParser and PolySolver classes.[14]

2.3 ExParser Class

The `ExParser` class is designed to create an object capable of parsing and evaluating mathematical expressions. This type of code is useful in scenarios where you need to process mathematical expressions entered as strings, such as in a calculator or a mathematical software application.

Methods Description: The code defines a class named `ExParser` to create an object capable of parsing and evaluating mathematical expressions. This type of code is useful in scenarios where you need to process mathematical expressions entered as strings, such as in a calculator or a mathematical software application.

The `__init__` method is a special method in Python that is called when an object of the class is created. In this case, the class is initialized with a dictionary of mathematical operators and their priorities. Operators include addition, subtraction, multiplication, division, and exponentiation. This dictionary contains mathematical operators as keys and tuples as values. Each tuple consists of two elements:

- The priority of the operator (an integer)
- A lambda function that defines how to perform the operation represented by the operator.

Here's a breakdown of the operators in the dictionary:

- `+`: Addition with priority 1
- `-`: Subtraction with priority 1
- `*`: Multiplication with priority 2

- `/`: Division with priority 2
- `^`: Exponentiation (power) with priority 3

The lambda functions associated with each operator perform the corresponding mathematical operation on two operands (`x` and `y`).

This class provides a foundation for building an expression parser that can handle basic arithmetic operations with different priorities. It allows for the parsing and evaluation of mathematical expressions based on the defined operators and their priorities.

The class provides methods for tokenizing, parsing infix expressions into postfix expressions, and evaluating the results.

`makeTokens` Method

The `makeTokens` method is responsible for tokenizing a given mathematical expression.

Parameters:

- `expression` (`str`): The input mathematical expression.
- `poly_value` (`float`): The value assigned to the polynomial variable 'x' (default is 0).

Tokenization Process:

- The method iterates through each character in the expression.
- It groups characters to form numbers, polynomial variables, or matrix identifiers.
- Tokens are appended to the list based on their type.
- Special cases, such as recognizing matrix identifiers and handling polynomial variables, are addressed.

Returns:

- An array of tokens representing the input expression.

`parseExpression` Method

The `parseExpression` method serves the purpose of converting an infix mathematical expression into postfix notation using the Shunting Yard Algorithm. Additionally, it evaluates the result of the expression.

Parameters:

- `expression` (`str`): The input mathematical expression in infix notation.
- `matrix_values` (`dict`): A dictionary containing matrices, where keys are matrix identifiers, and values are the actual matrices.
- `poly_value` (`float`): The value assigned to the polynomial variable 'x' (default is 0).

Process:

1. **Tokenization:**
   - The method calls the `makeTokens` method to obtain an array of tokens from the input expression.
2. **Shunting Yard Algorithm:**
   - The Shunting Yard Algorithm is applied to convert the infix expression into postfix notation.
   - It processes each token, handling operators, parentheses, and associativity rules.
3. **Postfix Expression Evaluation:**
   - The postfix expression is then evaluated using the `evaluatePostfixExp` method.
   - Numeric operands and matrices are pushed onto the operand stack.
   - Operators are applied to operands retrieved from the stack.
   - The final result is obtained from the top of the stack.
4. **Returns:**
   - The method returns the calculated result of the mathematical expression.

Usage:

Let's consider an example expression: `3 * (4 + x) / 2`, and we want to evaluate it with $x = 5$ and a matrix $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$.

```
# Instantiate ExParser
parser = ExParser()

# Define matrix values
matrix_values = {'A': np.array([[1, 2], [3, 4]])}

# Define expression and variable value
expression = "3 * (4 + x) / 2"
x_value = 5

# Parse and evaluate the expression
result = parser.parseExpression(expression, matrix_values, poly_value=x_val

print("Result of the expression:", result)
```

Execution Steps:

1. The `makeTokens` method is internally called to tokenize the expression and handle the variable 'x' and matrix identifiers.
2. The Shunting Yard Algorithm converts the infix expression into postfix notation: `3 4 x + * 2 /`.
3. The `evaluatePostfixExp` method is used to calculate the result of the postfix expression.
4. The final result is printed.

Output:

```
Result of the expression: 15.0
```

2.4 PolySolver Class

The PolySolver class is a Python class designed for solving polynomial equations and finding their roots. It provides methods for computing the degree of a polynomial, finding roots of linear and quadratic equations, extracting coefficients from polynomial expressions, finding roots of polynomial expressions, calculating derivatives of polynomial expressions, defining Jacobian vectors, and calculating Jacobian matrices. The class utilizes regular expressions, numerical methods, and symbolic computation libraries like SymPy and NumPy to perform these operations.

2.4.1 Methods Description

1. `polynomial_degree(poly_str):`

   The purpose of the `polynomial_degree` method is to determine the degree of a polynomial based on its string representation. Here we have some explainations about code.

   (a) **Regular Expression (Regex) Pattern**:
   - The code uses a regular expression (`re`) pattern to extract individual terms from the polynomial string.
   - The pattern `r'([+-]?\s*\d*\s*x(\^\d+)?)'` matches terms that consist of an optional sign (`+` or `-`), followed by an optional integer coefficient (`\d*`), followed by the variable `x`, and an optional exponent (`\^\d+`).
   - The `re.findall(term_pattern, poly_str)` function is used to extract all terms from the input polynomial string `poly_str`.
   - The extracted terms are stored in the `terms` list.

   (b) **Calculate the Degree**:
   - If no terms are extracted (i.e., `terms` is empty), the method returns the message "Not a valid polynomial."
   - If terms are found, the method proceeds to determine the degree by iterating through each term and extracting the exponent of `x` (if present).

- It updates the `degree` variable to be the maximum of its current value and the extracted exponent, ensuring that `degree` holds the highest exponent encountered.

(c) **Return the Degree**:

- Finally, the method returns the calculated `degree`, which represents the degree of the polynomial extracted from the input string.

Example

For `polynomial_degree`: If `poly_str = "3x^2 - 5x + 7"`:

- The terms extracted would be `[('3x^2', '^2'), ('-5x', ''), ('+7', '')]`.
- The degrees extracted would be `[2, 1, 0]` (from the exponents).
- The maximum degree encountered is `2`, indicating that the polynomial is of degree `2` (quadratic).

2. `compute_derivative(coefficients)`: The purpose of the `compute_derivative` method is to compute the coefficients of the derivative polynomial of a given polynomial represented by its coefficients.Here we have some explainations about code.

(a) **Input**:

- The method takes a list `coefficients` as input, where `coefficients` represents the coefficients of the polynomial in the form `[a0, a1, a2, ..., an]`, where `a0` is the coefficient of the highest-degree term (x^n), `a1` is the coefficient of x^{n-1}, and so on.

(b) **Reverse the Coefficients**:

- The code first reverses the order of coefficients using `coefficients[::-1]` to handle coefficients in the order `[an, ..., a2, a1, a0]`.

(c) **Check for Empty Coefficients**:

- If the `coefficients` list is empty (`not coefficients`), the method returns `[0]`, representing the coefficients of the derivative polynomial (which is 0).

(d) **Calculate the Derivative Coefficients**:

- To compute the derivative of the polynomial, the method iterates over the reversed coefficients list (excluding the constant term) using a list comprehension `[i * coefficients[i] for i in range(1, len(coefficients))]`.
- For each term coefficient `coefficients[i]`, it computes the derivative coefficient as $i \times$ `coefficients[i]`, where i corresponds to the degree of the term.

(e) **Return the Result**:

- Finally, the method returns `derivative_coefficients`, which contains the coefficients of the derivative polynomial.

Example

For `compute_derivative:` If `coefficients = [3, 0, -5, 0, 7]`, representing the polynomial $3x^4 - 5x^2 + 7$:

- Reverse the coefficients: `[7, 0, -5, 0, 3]`.
- Compute derivative coefficients: `[0, -10, 0, 12]` (derived from `[1*0, 2*(-5), 3*0, 4*3]`).
- Reverse back to original order: `[12, 0, -10, 0]`.

What We Are Calculating

- We are calculating the degree of a polynomial based on its string representation and the coefficients of the derivative polynomial of a given polynomial.
- The degree of a polynomial is the highest power of the variable (x) that appears in the polynomial expression.
- The derivative of a polynomial $P(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ with respect to x is $P'(x) = n a_n x^{n-1} + (n-1)a_{n-1}x^{n-2} + \cdots + a_1$.

3. `find_roots(expression_str)`: The `find_roots` method computes the roots of a polynomial expression provided as a string representation. Here we have some explainations about code.

   (a) **Input**:
   - The method takes `expression_str` as input, which is a string representation of the polynomial expression.

   (b) **Extract Coefficients and Determine Degree**:
   - The method first calls `extract_coefficients(expression_str)` to extract the coefficients of the polynomial from the string representation.
   - It then calls `polynomial_degree(expression_str)` to determine the degree of the polynomial based on its string representation.

   (c) **Select Root-finding Method Based on Degree**:
   - Based on the determined degree (`degree`):
     - If `degree == 1`, indicating a linear polynomial, the method calls `find_linear_roots` to compute the roots of the linear equation.
     - If `degree == 2`, indicating a quadratic polynomial, the method calls `find_quadratic_roots(coefficients)` to compute the roots of the quadratic equation.
     - If `degree >= 3`, indicating a polynomial of degree 3 or higher, the method uses `np.roots(coefficients)` from NumPy to compute all the roots of the polynomial (which could be complex).

   (d) **Return the Roots**:
   - Finally, the method returns the computed roots of the polynomial as a list.

Auxiliary Methods

`extract_coefficients`

- This method extracts the coefficients of the polynomial from its string representation.
- It handles various edge cases such as zero coefficients, invalid polynomial strings, etc.

`polynomial_degree`

- This method determines the degree of the polynomial based on its string representation.
- It uses regex patterns to extract terms and identify the highest degree.

`find_linear_roots` and `find_quadratic_roots`

- These methods compute the roots of linear and quadratic polynomials, respectively.
- They handle cases like real or complex roots using appropriate mathematical techniques.

Considerations

- Ensure that each auxiliary method (`extract_coefficients`, `polynomial_degree`, `find_linear_roots`, `find_quadratic_roots`) is correctly implemented and handles various edge cases appropriately.
- Utilize NumPy's `np.roots` function for higher-degree polynomials to efficiently compute complex roots if necessary.

Example

Suppose `expression_str = "`$3x^2 - 6x + 2$`"`

- The coefficients extracted would be `[3, -6, 2]` (using `extract_coefficients`).
- The degree of the polynomial would be `2` (using `polynomial_degree`).
- Since the degree is `2`, the method would call `find_quadratic_roots([3, -6, 2])` to compute the roots of the quadratic equation.

What We Are Calculating

- We are calculating the roots of a polynomial expression provided as a string by leveraging appropriate root-finding methods based on the polynomial's degree.
- The method efficiently handles different polynomial degrees and utilizes modular design to compute roots effectively.

,

4. `coffExp(coefficients, variable='x')`: Constructs a polynomial expression from an array of coefficients.
5. `calcExpDerivative(exp)`: Calculates the derivative of a polynomial expression and returns the resultant expression.
6. `compute_jacobian_matrix` method calculates the Jacobian matrix for a system of functions $f(x, y)$ and $g(x, y)$ with respect to the variables x and y.
   (a) **Input Parameters**:
      - `x`, `y`: Variables with respect to which the partial derivatives are computed.
      - `f`, `g`: Functions of x and y for which the Jacobian matrix is computed.

   (b) **Compute Partial Derivatives**:
      - The code uses `sp.diff(f, x)` and `sp.diff(f, y)` to compute the partial derivatives of function f with respect to x and y, respectively.
      - Similarly, it computes the partial derivatives of function g with respect to x and y using `sp.diff(g, x)` and `sp.diff(g, y)`.

   (c) **Construct the Jacobian Matrix**:
      - The Jacobian matrix J is constructed as a 2×2 matrix where:
        - $J_{11} = \frac{\partial f}{\partial x}$
        - $J_{12} = \frac{\partial f}{\partial y}$
        - $J_{21} = \frac{\partial g}{\partial x}$
        - $J_{22} = \frac{\partial g}{\partial y}$
      - This is done using `sp.Matrix([[df_dx, df_dy], [dg_dx, dg_dy]])`.

   (d) **Return the Jacobian Matrix**:
      - Finally, the method returns the computed Jacobian matrix J.

Example Usage

Suppose we have functions $f(x, y) = x^2 + y$ and $g(x, y) = xy - y^2$. To calculate the Jacobian matrix for these functions:

```
import sympy as sp

# Define variables and functions
x, y = sp.symbols('x y')
f = x**2 + y
g = x*y - y**2

# Compute Jacobian matrix
solver = PolySolver()
Jacobian_matrix = solver.compute_jacobian_matrix(x, y, f, g)

print("Jacobian Matrix:")
print(Jacobian_matrix)
```

What We Are Calculating

- We are calculating the Jacobian matrix J for the system of functions $f(x, y)$ and $g(x, y)$, which represents the rate of change of these functions with respect to changes in x and y.
- Each element J_{ij} of the Jacobian matrix represents the partial derivative of the i-th function with respect to the j-th variable.
- The Jacobian matrix is a fundamental tool in calculus and is used in various applications such as optimization, differential equations, and transformations.[3][15]

Usage

To use the PolySolver class, instantiate an object of the class, and then call its methods to perform specific tasks such as finding roots, calculating derivatives, or determining polynomial degrees.
Example usage:

```
solver = PolySolver()

# Find roots of a polynomial expression
roots = solver.find_roots("2*x^2 - 5*x + 2")
print("Roots:", roots)

# Calculate derivative of a polynomial expression
derivative = solver.calcExpDerivative("3*x^2 + 2*x + 1")
print("Derivative:", derivative)

# Calculate the Jacobian matrix
Jacobian_matrix = solver.compute_jacobian_matrix(x, y, f, g)

# Print the Jacobian matrix
print("Jacobian Matrix:")
print(Jacobian_matrix)
```

The PolySolver class provides a comprehensive set of functionalities for polynomial manipulation and equation solving. It combines symbolic computation and numerical methods to handle various polynomial-related tasks efficiently. With its intuitive interface and robust implementation, it serves as a valuable tool for mathematical computations involving polynomials.

3 Future Work

Researchers are working on making a mathematical expression parser even smarter by exploring different areas. One big focus is on improving how it handles complex math problems symbolically. This means developing techniques for things like proving theorems automatically, doing symbolic integration, and solving symbolic differential equations. The goal is to make the parser able to deal with tough math challenges using symbols, which would make it more useful across a wide range of math problems.

Another exciting area is combining the parser with machine learning. By training it with lots of examples of math expressions, we can help it learn patterns and relationships in math. This could make it better at spotting trends and making predictions. Alongside these advancements, there's also work happening to make the parser better at doing numerical analysis, creating tools to visualize math problems graphically, and even teaching it to understand math expressions written in plain language. Plus, researchers are exploring how we can use the parser in artificial intelligence systems to help with tasks like making decisions based on math. These efforts aim to make the parser more useful and user-friendly for all kinds of math and computer science tasks.

4 Conclusion

In summary, the creation of a mathematical expression parser and evaluator in Python has resulted in a powerful and flexible tool for managing mathematical calculations. By utilizing tools like NumPy, infix-to-postfix conversion, regular expressions, and arithmetic operations, the parser provides a comprehensive solution for accurately parsing, evaluating, and computing mathematical expressions.

Throughout its development, we've prioritized clarity, maintainability, and reliability. The code has been carefully organized, with each function neatly encapsulated within a dedicated Python library. Extensive testing, including numerical tests carried out in a Jupyter Notebook, has confirmed the strength and accuracy of the implemented features.

Looking forward, there are numerous exciting opportunities for further development and improvement. These include investigating optimization techniques to boost performance, expanding support for complex numbers and additional mathematical functions, integrating with external libraries and frameworks, and creating user-friendly interfaces for interactive use.

References

[1] Saba Akram and Quarrat Ul Ann. "Newton raphson method". In: *International Journal of Scientific & Engineering Research* 6.7 (2015), pp. 1748–1752.

[2] Christopher M Bishop. "Pattern recognition and machine learning". In: *Springer google schola* 2 (2006), pp. 645–678.

[3] Thomas F Coleman, Burton S Garbow, and Jorge J Moré. "Software for estimating sparse Jacobian matrices". In: *ACM Transactions on Mathematical Software (TOMS)* 10.3 (1984), pp. 329–345.

[4] Edsger Dijkstra. *Shunting-yard algorithm.* 2016.

[5] *Expression Parser.* URL: `https://www2.lawrence.edu/fast/GREGGJ/CMSC270/parser/parser.html`.

[6] George E Forsythe et al. *Computer methods for mathematical computations.* Prentice-hall, 1977.

[7] Charles R Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (2020), pp. 357–362.

[8] Donald Ervin Knuth. *The art of computer programming*. Vol. 3. Pearson Education, 1997.

[9] Rainer Kress. *Numerical analysis*. Vol. 181. Springer Science & Business Media, 2012.

[10] Yunyao Li et al. “Regular expression learning for information extraction”. In: *Proceedings of the 2008 conference on empirical methods in natural language processing*. 2008, pp. 21–30.

[11] William H Press. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[12] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-wesley professional, 2011.

[13] Gilbert Strang. *Introduction to linear algebra*. SIAM, 2022.

[14] Yunus Turayev and Nozima Kamolova. “OLIY TA’LIM TIZIMIDA FIZIKA VA MATEMATIKA DARSLARIDA PYTHON DASTURLASH TILIDAN FOYDALANISH”. In: *Research and Publications* 1.1 (2024), pp. 150–154.

[15] KJ Waldron, Shih-Liang Wang, and SJ Bolin. “A study of the Jacobian matrix of serial manipulators”. In: (1985).