Calculus & Graph - Functions Visualizations Python - Mysql Connectivity Project

Authors : **Vaibhav Bakshi** & **Rajveer Vora**

12th September 2023

PREFACE

Calculus - a remarkable study as well as an important field present in Mathematics. The mere mention of calculus evokes a certain mystique. It is basically a branch of mathematics that explores the concepts of change and motion. It provides a powerful set of tools and techniques for understanding and analyzing how things vary and evolve. Developed independently by Sir Isaac Newton and Gottfried Wilhelm Leibniz in the late 17th century, calculus has since become a fundamental part of mathematics and science and that's why we have tried to represent quite a small part of it in a high - level , general purpose, multi - paradigm programming language known as Python developed by the Dutch Programmer Guido van Rossum in early 1990s.

$$\int_a^b f'(x)\,dx = F(a) - F(b)$$

The Fundamental Theorem of Calculus.

So — what exactly are we doing here ? Well , we are here to show some basic operation of calculus like - Differentiation, Integration including the graph of vairous functions. The primary objectives of this project is to implement algorithms for calculating derivatives and integrals of user - input functions. It also uses mathematical libraries (numpy and matplotlib) for graphing functions.

Some of the basic features are:-

- Differentiation: Users can input functions, and the application will calculate and display their derivatives.
- Indefinite Integration: Users can input functions, and the program will calculate and display their indefinite integrals.
- Definite Integration: Users can input functions, and the program will calculate and display their indefinite integrals.
- Graph Visualizations: Graphical representations of functions to aid in visualization.

Also it is to be noted that for simplicity we have used basic linear expressions of function like for tangent functions we have $\sin(ax+c), \cos(ax+c)$ and so on. Similarly for exponential functions we have $e^{(ax+c)}$, etc. This is done for simplicity of the code where the user will input the values of **a** and **c** (sometimes **b** for inverse functions in Integration). We can give an example right here :

$$\int \sin(ax+c)\,dx = -\cos(ax+c) + C$$

where, **a** and **c** are input by the user.

Contents

User Defined Functions

This section explains all the algorithms of user defined functions and their uses in the program.

1 Creating the MySql Database

```
def create_SQL_database(host_name, user_name, password):
```

The first function is pretty much easy to understand. It has parameters of hostname, username and password which will be obviously entered by the user. Next it establishes a connection to that particular MySql Username and Host using a connection object. Following that, it displays an appropiate message and then creates the database and the table. The use of the table is to track the input history of the user. It is in format of 7 columns. Those are listed below with the possible combinations written on right side:

- **Operation Type**: Differentiation / Indefinite Integration / Definite Integration / Graphs
- **Operation Type** Choice: 1 / 2 / 3 / 4
- **Function Type**: Trigonometric / Inverse Trigonometric / Polynomial / Logarithmic / Exponential
- **Function Type Choice** : 1 / 2 / 3 / 4 / 5
- **Function Expression**: The function chosen by the user along with the values of variables input by the user
- **Function Expression Choice**: Depends on the number of functions in a particular choice of Function Type
- **Answer**: Contains the answer of the expression.

Clearly, **Answer** column has the **UNIQUE** constraint so it will have distinct and different results. Finally, in the end we have the except block for catching errors and stopping the code altogether and finally block for closing the connections to save data loss.

2 Insertion of Data into MySql

```
def insert_into_SQL_database(host_name, user_name, password, L):
```

This is another easy function to understand. It contains an additional parameters **L** which is a list containing all the parameters arranged in to the order of the column of the table in the database. It basically executes the query of inserting the values via the cursor object from a list which is in the same order as the order of columns in the table. In the Except block we ignore the errors due to duplication which would otherwise hinder the process of running the program.

3 Checking the Duplicates

```
def check_duplicates(x, host_name, user_name, password):
```

This function checks whether there are any duplicate results by fetching the result set from the table in database into a variable **table** in a tuple form. Apparently for skipping calculations we need to compare it to the **Function Expression** column where the values in table can be compared with what the user entered in the program which is stored in the parameter **x**. If the result is found then a counter (initialized before) is incremented to 1 and the loop is broken. Then the answer and the counter are returned in form of a list and printed instead of performing calculation again. If the condition is not satisfied the counter remains zero then the answer is calculated and then inserted into the database.

4 Superscript Function

```
def get_sup(x):
```

A tricky yet simple function. It converts a base number to its superscript equivalent. The normal characters are stored in the variable **normal**, while the superscript forms are stored in **super_s**. For example, **2** gets converted to ^{2} and so on. It uses two functions **maketrans()** and **translate()** which behave as mapping-table functions. Now **maketrans()** replaces all instances of **normal** with the ones in **super_s**. Then **translate()** returns the modified string variable.

5 Conversion to LaTeX format for Differentiation Function

```
def convert_to_Latex_for_differentials(x, mappings):
```

This one is a bit complicated and we will go line by line. So we will divide into specific sub - sections as follows:

5.1 Parameters

Apparently this function has 2 parameters , i.e., the **x** which is for the function to be converted and the **mappings**. The mappings consists of all the functions with their LaTeX equivalent. Some of the list of mappings (not all) are as follows:

```
mappings = {
    "cos": r"\cos",
    "sin": r"\sin",
    "tan": r"\tan",
    "csc": r"\csc",
    "sec": r"\sec",
    "cot": r"\cot",
    "log": r"\log",
    "(": r"{(",
```

```
    ")": r")}",
    "(e": r"(\mathrm{e}^",
}
```

5.2 Square Root($\sqrt{}$) Checking along with Division Symbol(/)

This checking is done majorly for those functions in **Inverse Trigonometric** ones like $\sin^{-1}$, $\cos^{-1}$, $\tan^{-1}$, $\cot^{-1}$, etc. This is majorly because the $\frac{d}{dx}(\sin^{-1}(ax+c)) = \frac{a}{\sqrt{1-(ax+c)^2}}$ and so on and so forth for other functions and clearly in normal mathmatical format we both have square roots and division symbols at same place. Now we find the index of the square , i.e., ^{2} in the expression using the **index()** function. After that we check whether there is a **bracket "()"** or **minus sign "-"** in the next position or not as follows:

```
if x[n + 1] == ")" or x[n + 1] == "-":
            x = x.replace('^2', r'^{2}')
```

Note: It wasn't possible to display ^{2} due to encoding issues so we replaced with "^2".

Now finding whether there is bracket or not after the index of square is obvious because of the above result of inverse sine function. However we also used to check the **minus "-"** because of the fact that the $\frac{d}{dx}(\csc^{-1}(x)) is \frac{-1}{|x|\sqrt{x^2-1}}$ and we can clearly notice the **minus "-"** in this. It's same the same thing for $\sec^{-1}(x)$ function. Overall we replace square here with its LaTeXequivalent.

5.2.1 Division Symbol Splitting , Replacements & Final Expression Formation

This part is pretty easy to understand. First we split the expression on the basis of /. This is to seperate the contents of numerator and denominator in the returned list. Next we have:

```
for item in expression_list:
            updated_item = item
            for key, value in mappings.items():
                updated_item = updated_item.replace(key, value)
            updated_expression_list.append(updated_item)
```

Basically it performs the replacement things like changing the normal mathematical forms into their LaTeX equivalent respectively in the **expression_list** and then appends it to the new list **updated_expression_list** which contains the modified expressions. Lastly we have this:

```
st = r"$\dfrac{" + updated_expression_list[0] + "}{" +
   updated_expression_list[1] + "}$"
```

This is also the basic format of a fraction we have in LaTeX. So the basic format is that if $\dfrac{numerator}{denominator}$ is entered it renders in form of $\frac{numerator}{denominator}$. Hence we have accordingly formed the string by contatenating the **$** symbol along with **dfrac** and also the closing part at end which is again a **$**. The middle part **updated_expression_list[0]** and the

updated_expression_list[1] contains the numerator and the denominator respectively as the numerator always comes first in normal linear mathematical expression.

5.3 Other Conditions & Final Results

The other condition contains a basic thing of splitting on basis of division (/) character and seperating the numerator and denominator in the list and then writing it directly in the form of LaTeX fractional expression. This was done for normal fractional functions like $\frac{\sin(5x+4)}{5}$.
In the next condition, we check whether a square is present in the expression. This is for polynomial and exponential functions and it checks whether there are any parentheses after the square symbol for verification purposes. We then replace it with the LaTeX equivalent of square, i.e., ^{2}.

Finally we print the modified string and return it in the function.

5.4 Working Example

Let's a take a function which contains all the above instances. Suppose the user chooses $\csc(ax+c)$ choice under **Inverse Trigonometric** in **Differentiation**. We know that :

$$\frac{d}{dx}(\csc(ax+c)) = \frac{-1}{|x|\sqrt{x^2-1}}$$

If the user enters **a** as **5** and **c** as **4** then the expression to be evaluated is $\csc(\mathbf{5x+4})$. Its differentiation is obviously clear, i.e., $\frac{-5}{|(5x+4)|\sqrt{(5x+4)^2-1}}$. In normal linear form it is $-\mathbf{5}/|\mathbf{x}|\sqrt{((\mathbf{5x+4})^{\mathbf{2}}-\mathbf{1})}$.
To convert it to LaTeX form it will check whether there are square roots and the square symbol. Since the condition is satisfied it finds out the index of **^** in the expression and its clearly in 16th Index position. Clearly after the 16th Index it checks whether there is a minus symbol or not and since it is there so the square symbol is replaced by its LaTeX equivalent. So the linear expression becomes $-\mathbf{5}/|\mathbf{x}|\sqrt{((\mathbf{5x+4})\text{\^{}}\{\mathbf{2}\}-\mathbf{1})}$.

Next it splits the expression into two parts , i.e. , the numerator and the denominator in the list. The list would look like **[-5 ,** $|\mathbf{x}|\sqrt{((\mathbf{5x+4})\text{\^{}}\{\mathbf{2}\}-\mathbf{1})}$ **]**. The numerator is on the left side while the denominator on the right side. Next we iterate the list and perform the replacements with the values of keys defined in the **mappings** dictionary which contains their LaTeX equivalents. That brings the final expression list to **[-5 ,** $|\mathbf{x}|\backslash\mathbf{sqrt}\{((\mathbf{5x+4})\text{\^{}}\{\mathbf{2}\}-\mathbf{1})\}$ **]**

So finally we can now concatenate the contents of the list to by joining the numerator with '**$\dfrac{**' with the numerator which is '-1' and then '**}{**' and then the denominator and finally the '**}$**'. Hence the final experssion is **'$\dfrac{-5}{|x|\sqrt{((5x + 4)^{2}-1)}$'**.

6 Conversion to LaTeX format for Integral Function

```
def convert_to_Latex_for_integrals(x, mappings):
```

Another complicated and comprehensive function we have here and we will be also going through this in details. Parameters are same as the above function discussed earlier.

6.1 Square Root and Power 1 (^{1}) Checking

The general Integrals have '**+C**' with the answer of their integrals which is the constant of Integration. It is due to the fact that for a number of functions on differentiation we get a common answer. However on integrating (which is the reverse process of Differentiation) it does not always give that particular answer and hence a constant of Integration is defined. Clearly, the length of '**+ C**' is 4. Hence we are taking the expression by slicing it from 1st index to the 5th last index so as to omit the constant of Integration as here:

```
expression_list = x[1:-5].split("/")
```

The reason why we chose from 1st to -5th Index is that in 0th Index we want to omit the enclosing extra parentheses () and also omit the constant of Integration. **Also it is to be noted that this condition is especially for complicated functions like the inverse tangent functions (e.g.:** $\frac{\tan^{-1}(\frac{(5x+4)}{\sqrt{(3)}})}{(5\sqrt{(3)})} + C$**)**. So the above '**split()**' function will actually give a 3 member list due to the presence of 2 fractions. Next we have :-

```
func_end_idx = (expression_list[0].index("^{1}"))
a = expression_list[0][func_end_idx + 1:]
updated_expression_list = [a[1:], expression_list[1][:-1]]
```

After splitting into the list we find the position of power 1 in the numerator , i.e., the 0th index of the **expression_list**. The reason why we chose the numerator for checking is that in General Integration if any coefficient is multiplied with the variable it automatically gets divided by the final answer (partly due to the way how **Integral Substitution** works). For example $\int \frac{1}{\sqrt{1-4x^2}}\,dx = \frac{\sin^{-1}(2x)}{2} + C$ and clearly the main function is in numerator.

Now after that we store the function parameter in a variable **a** including the parentheses '**()**' with one extra opening one. So for e.g. if $\sin^{-1}((5x+4)$ is the 0th element in list, then '**((5x + 4)**' will be stored. Next we store 2 elements in a new list , i.e., **updated_expression_list**. We first store a part of **a** excluding the first open parentheses. The 2nd element in the **updated_expression_list** is the **modified 2nd element of the original expression_list which just contains the expression excluding the closing parentheses**. Next we have is:

```
for i in range(len(updated_expression_list)):
    updated_item = updated_expression_list[i]
```

```
    for key, value in mappings.items():
        updated_item = updated_item.replace(key, value)
    updated_expression_list[i] = updated_item
```

Its clearly the general replacement algorithm with their LaTeX equivalents which was discussed earlier before with the **mappings**. dictionary. The **updated_expression_list** contains the LaTeX equivalents which can be concatenated as follows:-

```
st = r"\dfrac" + updated_expression_list[0] + "{" +
   updated_expression_list[1] + "}"
```

Here as usual the 0th Index contains the numerator and the denominator is at the 1st Index. Now we have :-

```
func = expression_list[0][:func_end_idx + 1]
```

This basically stores the function name along with the inverse symbol (containing the minus and 1) into a variable '**func**'. Next we perform replacements and also additionally for the **backslashes** to escape them from **Python's Escape Sequence**. Next we have:

```
num_r = func + "{(" + st + ")}"
den_r = expression_list[2]
```

The numerator part is easy to understand as we are contatenating the **function name** '**func**' with the **function parameter** '**st**' along with curly braces. However in second part we are taking the denominator from the 3rd element of the original **expression_list** and then perform replacements as follows:

```
for key, value in mappings.items():
      den_r = den_r.replace(key, value)
```

Finally, we form the final expression as in the general LaTeX expression for fractions as follows:

```
final_exp = r"\dfrac" + "{" + num_r + "}" + den_r
st = '$' + final_exp + x[-4:] + '$'
```

The reason why we added **x[-4:]** is to add the constant of integration '+ **C**' which we truncated at first.

6.2 Division Symbol '/' Checking

Works clearly as the one discussed in Differential LaTeX converter. Basically splits the expression into 2 parts in list due to presence of 1 division symbol only as follows:

```
expression_list = x[1:-5].split("/")
updated_expression_list = []
```

Now we perform the replacements including for the backslash to avoid escape sequences as follows:

```
for item in expression_list:
    updated_item = item

```

```
    for key, value in mappings.items():
          updated_item = updated_item.replace(key, value)

    updated_item = updated_item.replace('\\\\', '\\')
    updated_expression_list.append(updated_item)
```

The final expression we thus get is:

```
st = '$' + r'\dfrac{' + updated_expression_list[0] + "}" + "{" +
   updated_expression_list[1] + "}" + x[-4:] + '$'
```

6.3 Square Checking (^{2})

This condition is mainly for exponential, logarithmic and polynomial expressions. Clearly we first find the index of the square by:

```
n = x.index('^{2}')
```

We then check whether there is a opening parentheses sign or a plus sign as follows:

```
if x[n + 1] == "(" or x[n + 2] == "+":
      x = x.replace('^{2}', r'^{2}')
```

In the end , we finally perform replacements and return the expression:

```
for key, value in mappings.items():
    x = x.replace(key, value)

    st = "$" + x + "$"
```

6.4 Working Example

Lets take a complicated function as we took before that is the $\int \frac{1}{3+(5x+4)^2}$. Clearly its answer was already given above which is $\frac{\tan^{-1}(\frac{(5x+4)}{\sqrt{(3)}})}{(5\sqrt{(3)})} + C$. However in linear form it will be displayed as:-

$$\underline{(\mathbf{tan^{-1}((5x+4)/\sqrt{(3)})/(5\sqrt{(3)}))+C}}$$

We need to convert this into LaTeX format. Let's follow step by step.

Firstly, we check whether this expression has power (^{1}) and $\sqrt{}$ symbols. Clearly its present so it executes the first condition. Now going line by line it first executes the following:

```
expression_list = x[1:-5].split("/")
```

It gives a list **['tan^{-1}((5x + 4)' , '$\sqrt{(3))}$' , '(5$\sqrt{(3))}$']**. Now we find the position of [1] in the first element of **'expression_list'**. It clearly **4**. After that from the following:

```
a = expression_list[0][func_end_idx + 1:]
updated_expression_list = [a[1:], expression_list[1][:-1]]
```

We can clearly see that the function parameter including 3 parentheses (2 open, 1 close) is stored in **a**. In other words **((5x + 4)** is stored in **a**. Now in the **updated_expression_list** variable , the first element which we store in the list is the function parameter except for the first parentheses, i.e., **(5x + 4)** is stored. The next element is the denominator '$\sqrt{(3))}$' which is stored as '$\sqrt{(3)}$' , i.e., except for the last parentheses. So the '**updated_expression_list**' is basically the list as :- '**[(5x + 4) , $\sqrt{(3)}$]**'.

Now since we got the list already we can now easily convert it into LaTeX format for fractional expression. But first we replace the expressions by their LaTeX equivalents using the following:

```
for i in range(len(updated_expression_list)):
     updated_item = updated_expression_list[i]
     for key, value in mappings.items():
          updated_item = updated_item.replace(key, value)
     updated_expression_list[i] = updated_item
```

And after this we concatenate the strings accordingly as:

```
st = r"\dfrac" + updated_expression_list[0] + "{" +
   updated_expression_list[1] + "}"
```

So we have the expression as '**\dfrac{(5x + 4)}{\sqrt{(3)}}**'. After this we store the function name along with the inverse sign, i.e., 'tan^{-1}' in a variable **func** using this:

```
func = expression_list[0][:func_end_idx + 1]
```

After this we perform replacements (as discussed above) but this also includes replacing the backslashes properly due to Python's escape sequence. Now after this we form the **numerator** as :-

```
num_r = func + "{(" + st + ")}"
```

Here we concantenated the modified expressions like the **func** which is the **function name** along with the **function parameter** stored in the **st** variable. So we finally have the numerator ready as '**\tan^{-1}{\dfrac{(5x + 4)}{\sqrt{(3)}}}**'. In the denominator we store the last element of the original **expression_list** in a variable **den_r** as:-

```
den_r = expression_list[2]
```

Lastly we perform replacements again in **den_r** again including the backslashes and then finally we concatenate the two properly to make a LaTeX expression as follows:-

```
final_exp = r"\dfrac" + "{" + num_r + "}" + den_r
st = '$' + final_exp + x[-4:] + '$'
```

So we finally the expression which will be returned is: -

$$\underline{\dfrac{\tan^{-1}{\dfrac{(5x + 4)}{\sqrt{(3)}}}}{5\sqrt{(3)}}}$$

Note: The Division Symbol Checking behaves the same way as discussed in the Differential LaTeX conversion. Same for the Square Checking Part.

7 LaTeX Display in Matplotlib

```
def display_Latex(expression):
```

Apparently Matplotlib allows LaTeX input to be rendered in a beautiful format. This is due to the fact that Matplotlib is integrated with the TeX Engine which basically recognises LaTeX input and hence is able to render it in textual format. The function is pretty easy to understand as follows :-

```
plt.text(0.5, 0.5, expression, fontsize=15, ha='center', va='center')
plt.axis('off')
plt.show()
```

If you have noticed in the first line of our whole code we have called the **Matplotlib Library** as '**plt**'. Hence we used the **plt.text()** to display the text. The expression is the parameter here and we straight away called it in the **text()** parameters. The first two are the **x** and **y** coordinates which are both **0.5**. The **fontsize** is defined as **15**. The '**ha**' and '**va**' are the **horizontal** and **vertical** alignments which are both set to **center**.
Lastly we disable the **axes** so as to produce a clean text and then show the plot using **plt.show()**.

8 Evaluation of Linear Mathmatical Expression to Numpy Expression

```
def evaluate_math_expression_to_numpy(x):
```

Quoting from **Wikipedia.org**, **NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays**. However we will only use Numpy here for displaying Graphs which are discussed later on. The parameter is obviously the Linear Mathematical Expression as discussed earlier.
Now the first thing we do in this function is to make a copy of the parameter in a variable **st**.

```
st = x
```

Then we defined a dictionary **mappings** which contains all the **linear expressions** as **keys** and their **Numpy equivalents** as **values**. The mappings was not displayed here due to UTF - 8 encoding issues. However it contained all the function symbols which were to be replaced by their Numpy equivalents. The order in the **mappings** is important due to the order of replacement. However the one exception is the **Absolute Value Function** whose both closing and opening

symbols are same due to which we chose '**[x]**' instead of '$|x|$' where the third brackets are for the symbols.
We perform the usual replacement algorithm and then return the original expression and the modified expression as list to be used for later purposes. The code is as follows:

```
for key, value in mappings.items():
     st = st.replace(key, value)
return [st, x]
```

9 Range Deciding Function

```
def decide_range_of_function(a):
```

Before we proceed we need to know what **linspace** and **arange** are.
Linspace basically defined as below is a function used to create an **array**. An **array** is basically a well defined ordered sequence of homogenous items.

```
numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=
   None, axis=0)
```

Although in above case we have not use all the parameters except for the first three but its good to know what these parameters do. Quoting from '**GeeksforGeeks.org**' we have :-

- **Start**: The starting value of the sequence.
- **Stop**: The ending value of the sequence.
- **Num**: The number of evenly spaced samples to generate. Default is 50.
- **Endpoint**: If **True**, stop is the last value in the range. If **False**, the range ends before stop. **Default** is **True**.
- **Retstop**: If **True**, return **(samples, step)**, where **step** is the spacing between samples.
- **Dtype**: The data type of the output array. If not specified, the data type is inferred from the input values.
- **Axis**: The axis in the result along which the linspace samples are stored. The default is 0.

Apparently only the first three parameters are needed. Now for us to understand how its works lets take an example:-

```
arr1 = np.linspace(0, 1, num=5)
print("Example:", arr1)
```

So if we actually calculate 5 numbers in the interval **[0, 1]** which are evenly seperated by **their values** then we have **0, 0.25, 0.5, 0.75, 1**. So the output will be '**[0, 0.25, 0.5, 0.75, 1]**'.

Lets move on to **arange()** function. It is basically the in - built **range()** function but it returns a **Numpy array**. It is defined as:-

```
numpy.arange(start, stop, step, dtype=None)
```

Clearly we have:

- **Start**: The starting value of the sequence.If not provided, the default is 0.
- **Stop**: The ending value of the sequence.
- **Step**: The step or spacing between the values. If not provided, the default is 1.
- **Dtype**: The data type of the output array. If not specified, the data type is inferred from the input values.

For example we have the following :-

```
array1 = np.arange(5)
print("Example:", array1)
```

Using the above two functions helps us to make the graph more appealing and visualising because we can select suitable ranges for specific intervals.
The above code prints an array containing the numbers from **0** to **5** at a step value of **1**. Therefore the output will be:- '**[0, 1, 2, 3, 4]**'. The parameter '**a**' is obviously the **modified numpy expression** of the **original expression** entered by the user. The modification is done using the function **def evaluate_math_expression_to_numpy(x)** which is discussed earlier. The code is written in a basic if and else condition that on the basis of what function is there it decides the **linspace** and **arange** accordingly as:

```
elif "log" in a:
        n = np.linspace(0.1, 5, 100)
        m = np.arange(0.1, 5, 0.1)
```

We finally store both these values in form of a list to be used later :-

```
return [m, n]
```

9.1 Working Example

Suppose the user enters the function 'sin $(\mathbf{5x+4})$' then the corresponding **linspace** and **arange** is chosen, i.e. :-

```
n = np.linspace(-10, 10, 1000)
m = np.arange(-10, 10, 1)
```

Clearly the **linspace** displays 1000 points between the intervals **[-10 , 10]** evenly spaced and so does the **arange** which displlays the points at an interval of **1** value.

10 Graphing Function for Universal Mathematical Functions

```
def graph_function_for_universal_functions(I):
```

A general graphing function which is used for displaying graphs of various functions in a general format using Matplotlib. Quoting from **Wikipedia.org**, **Matplotlib is a plotting library for the Python programming language and its numerical mathematics extension NumPy. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits like Tkinter, wxPython, Qt, or GTK**. Proceeding with the code we first have:

```
s = evaluate_math_expression_to_numpy(I)[0]
m = evaluate_math_expression_to_numpy(I)[1]
L = decide_range_of_function(I)
```

This is pretty easy to understand we perform the **Numpy Conversion** as discussed earlier by using the function **def evaluate_math_expression_to_numpy(x)** on the parameter **I** which is the **original expression** entered by the user. Since it returns a list containing two elements in the form **'['modified_numpy_expression', 'original_expression']'**. So **'s'** is the **modified_numpy_expression** while **'m'** is the **original_expression** which are stored in their respective variables. We also store the **linspace** and **arange** in the a list **'L'**. Next we have:-

```
if "[" in I:
    I = I.replace("[", "|")

if "]" in I:
    I = I.replace("]", "|")
```

If you remember earlier due to closing and opening sign similarity of **Absolute Value Function** we used **[ ]** instead of '| |'. Since we want to display it correctly in the **Graph Labels** we therefore performed replacements in the parameter '**I**' which was entered by the user. Next we display the **Numpy Modified Expession** in text. Now we store the **linspace** which is the 2nd element in the list returned by the function **decide_range_of_function(I)** in a variable **'x'**. Also we store the **Numpy Modified Expression** in an evaluated form in a variable **y** instead of usual string as follows :-

```
x = L[1]
y = eval(s)
```

The next part is the plotting of the whole function using **Matplotlib**. We first use **plt.plot()** function with the x and y axis where **x** was the **linspace** and **y** was the **Numpy Modified Expression** in evaluated form. We also added the labels for the function by calling the parameter **I** in a formatted string :-

```
plt.plot(x, y, label = f'{I}')
```

Next we have **axhline()** and **axvline()** which basically define the x and y axes in a properly manner such they are easily visible and differentiable :-

```
plt.axhline(0, color='black', linewidth=1.5)
plt.axvline(0, color='black', linewidth=1.5)
```

We gave the color **black** since the background of the cartesian plane is white and also the **linewidth** is **1.5** so as to make a little bit thicker than the other lines. Next we have are the labels where the **x - axis label** is basically the string '**x**' and the **y - axis label** is the formatted string of the parameter which is entered by the user. The code is as follows :-

```
plt.xlabel('x')
plt.ylabel(f'{I}')
```

Finally we have :-

```
plt.title(f'Graph␣of␣{I}')
plt.xticks(L[0])
plt.grid()
plt.legend()
plt.show()
```

This is easy to understand. We first define the label where we again call the parameter in the formatted string. Next we call the **arange function** the first element of the list **L** which we got from the **decide_range_of_function(L)** inside the **xticks()** function. The textbfxticks() function is used to mark specific points on the x - axis and it accepts those points in form of a list which is given by the **arange()** function. The **labels** though optional and not defined in this case are the second parameter. Next we set the **grid**, **legend** to **True**. Then we finally show the graph using **plt.show()**.

11 Graph Function For Polynomials

```
def graph_function_for_polynomials(coefficients, final_exp):
```

This function is especially for polynomials p(x) of n - degree or of form where :-

$$\mathbf{p(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \ldots + a_1 x^1 + a^0}$$

11.1 Parameters

The parameters are pretty easy to understand which are the **coefficients** and the **final expression**. The **final expression** is called here to display it as a label.

11.2 Finding Roots and Evaluating the Range

In **Numpy** we can use **np.roots()** to find the roots of a particular polynomial. The **np.roots** takes an **array containing the coefficients of all the powers of variables along with the constant at the last** and finds all possible **real roots** only. For example if we have p(x) = $x^2 - 3x + 2$ then its array of coefficients will be **'[2, -3, 2]'**. Hence on **np.roots()** it would give

1 and 2 in form of an array which is **'[1, 2]'**. Hence we used that and thereafter we found the maximum and minimum roots using **np.max()** and **np.min()** respectively as follows:-

```
roots = np.roots(coefficients)
min_root = np.min(roots)
max_root = np.max(roots)
```

Next we decide the **linspace** accordingly from the minimum and maximum roots with a 1000 point gap between them for more detailing. This is so as to display which part of the function is intersecting the x - axis. Also we used **np.polyval(coeff, x)** which is used to evaluate the value of p(x) at a particular value given in the parameter **'x'**. This enables us to draw the curve over a set of values which is given by **linspace** as follows:-

```
x = np.linspace(min_root - 1, max_root + 1, 1000)
y = np.polyval(coefficients, x)
```

Next we define the label in the formatted in the string as the **final_exp**. The other lines are same as discussed earlier. The only difference is that we used **plt.ylim()** function. It is a function used to set or get the y-axis limits of the current axes. It is often used in conjunction with other plotting functions to control the range of values displayed along the y-axis. The last part is as follows :-

```
plt.plot(x, y, label=f'f(x) = {final_exp}')

plt.axhline(0, color='black', linewidth=1.5)
plt.axvline(0, color='black', linewidth=1.5)

plt.xlabel('x')
plt.ylabel(f'{final_exp}')

plt.title(f'Graph of {final_exp}')
plt.ylim(bottom=min(plt.ylim()[0], -1))
plt.grid()
plt.legend()
plt.show()
```

Therefore we come to the end of the discussion of algorithms of the user defined functions. Let's move on to the Main Code.

Preliminaries for Main Code

The main code structure is quite basic. The basic structure of the code is as follows :-

```
print("-" * 80)
print("CALCULUS␣AND␣GRAPH␣-␣FUNCTIONS␣VIZUALIZATION␣PYTHON␣-␣MYSQL␣
   CONNECTIVITY␣PROJECT")
print("-" * 80)
print("1.␣While␣taking␣input␣use␣parentheses␣to␣avoid␣operator␣precedence␣
   confusion.")
.    #<RULES STARTING>
.
.
.    #<RULES DEFINED HERE>
.
.    #<RULES ENDING>
#<input to proceed>

#<while loop to proceed>
    #<try block for handling database errors>
        #<inner while loop for main operations execution>
            #<try block for handling arithmatic and other errors>
                #<main code>
            #<closing except block>
    #<closing except block>
#<else condition for displaying msg when no proceeding>
```

Lets go through the code once. Clearly we have defined a couple of rules beforehand which need to be **strictly** followed. After that we asked an input from user regarding his / her choice to proceed with the program :-

```
n = input("Make␣sure␣you␣have␣read␣the␣above␣mentioned␣rules.␣Press␣Y␣to␣
   proceed␣(Y/N):␣")
```

Now if the **'n'** is **'Y'** or **'y'** then the code obviously proceeds with the **while loop** but if not then it displays the message:

```
else:
    print("Code␣Exited␣due␣to␣non␣-␣agreement␣of␣user.")
```

[style=python] Now in the **while** loop we have the **try** block where the user needs to enter **host_name**, **user_name** & **passwd**. If a database error occurs then it executes the **except** block where **os._exit(0)** is performed which stops the program completely.

```
try:
        host_name = input("Enter␣host␣name:␣")
        user_name = input("Enter␣user␣name:␣")
        passwd = input("Enter␣password:␣")

        create_SQL_database(host_name, user_name, passwd)
```

```
except:
      except (UnboundLocalError, ValueError, KeyboardInterrupt, NameError)
          as e:
       print("Unexpected Error occured. Please try again.")
       print("Error encountered :", str(e))
       os._exit(0)
```

That's why we defined two try and except blocks in this program. The inner while loop contains the main code and its also in a try and except block in order to avoid Arithmetic and other possible errors.

```
while True:
    print("SELECT OPERATION TYPE")
    print("MAIN MENU:\n1.Differentiation\n2.Indefinite Integration\n3.
       Definite Integration\n4.Graphs\n5.Exit")

    try:
       #<MAIN PROGRAM HERE>
    except (ValueError, KeyboardInterrupt, ZeroDivisionError,
       UnboundLocalError, NameError):
       print("Invaild Input. Please try again")
```

Next we have are bunch of variables defined for later purposes as follows :-

- **Mappings** here is basically the dictionary containing the keys which are in linear mathematical form and the values which are in LaTeX form.
- **Ch1, ch2, ch3** all these are choices and the functions respectively initialized for later uses.
- **Inv** is the $^{-1}$ symbol.
- **func, func_1, func_type** are the function variable defined for LHS uses.
- **I** is for storing the answer.
- **LHS** is for storing the Left Hand Side, i.e., the question according the user inputs.

It is also important to note that basic structure of the code,i.e., '#<**MAIN PROGRAM HERE**>' in the second try block:-

```
print("SELECT OPERATION TYPE")
print("MAIN MENU:\n1.Differentiation\n2.Indefinite Integration\n3.Definite
   Integration\n4.Graphs\n5.Exit")

try:
    #<Variables Defined Here>
    #<Choice for OPERATION TYPE accepted from user>
    if ch1 == 1:
        #<Differentiation Code>
    elif ch1 == 2:
```

```
        #<Indefinite Integration Code>
    elif ch1 == 3:
        #<Definite Integration Code>
    elif ch1 == 4;
        #<Graphing Code>
    elif ch1 == 5:
        #<Exit Code>
    else:
        #<Exit Code>
    #<Display and Result>
except:
    #<Except Block>
```

Clearly we are executing functions on the basis of choices input by user. The basic layout of all the first 3 operations except for Graphing Function is as follows (for all operation types including function types) :-

```
print(``#<List of functions' operation here which needs to be performed. E
   .g: Trigonometric, Logarithmic>'')
ch = #<Choice input by the user>
a = #<input by user>
c = #<input by user>
b = #<input by user only for Inverse Trigonometric Functions type>
choice_type = #<List of function operation type>
if ch == #<Choice entered by the user>:
    print(``#<List of functions' type here which needs to be calculated. E
       .g.: sin(5x+4), cos(5x+4)>'')
    ch = #<Choice input by user>
    expression_list_2 = #<List of choices>
    exp = f"{a}x+{c}"

    if ch == #<Choice entered by the user>:
          func = new_expressions_list_2[<choice input by user> - 1]
          if check_duplicates(func, host_name, user_name, passwd)[0] == 1:
                print("Record fetched from database")
                I = check_duplicates(func, host_name, user_name, passwd)
                   [1]
           else:
                print("Record added to database.")
                I = #<Answer Evaluation>
```

A print statement first displays the operation type which contains 5 types. The user enters a choice out of any 5 and accordingly the user is prompted with the choices of functions' type line **sine, cosine**,etc. Before that the user inputs **a, c and b**. Next an **'expression_list_2'** is defined which contains all functions' type. This is for tracking the choice as in the next line in variable **func**. Using check_duplicates we can check whether it exists in database and we know it returns in form of a list with **counter C** in 0th Index and **Answer** in 1st Index. Else it is added to database later with the answer defined in a variable **I** declared before. Let's move on to the different sections.

Sections of Main Code

12 Differentiation Section

Now we first have a variable **'func'** which basically will be used later for LaTeX format storing. Next as we saw before we defined the 5 types of operation possible. Then the choice is defined. After that the user inputs the values of **a** and **c** (**b** in case of Inverse Trigonometric Functions). The later part is already discussed as above. If a user enters any one choice out of the 5 then a new prompt comes which tells to enter the function types like of sine , cosine functions, etc. Next we have is :

```
expressions_list = [r"\sin{(ax+c)}", r"\cos{(ax+c)}", r"\tan{(ax+c)}", r"\
    cot{(ax+c)}", r"\csc{(ax+c)}", r"\sec{(ax+c)}"]
expression_list_2 = ["sin(ax+c)", "cos(ax+c)", "tan(ax+c)", "cot(ax+c)", "
    csc(ax+c)", "sec(ax+c)"]
new_expressions_list = [i.replace("ax+c", exp) for i in expressions_list]
new_expressions_list_2 = [i.replace("ax+c", exp) for i in
    expression_list_2]
```

The first expression list is for LaTeX replacement with the **exp** variable which is **ax** + **c** so as to form the list of **LHS** choices and then select the choice according to the user. The next one is for forming the normal **LHS** for addition to database. Both these are stored in **function** and **func** variables respectively. This is true for all the choices in this part. All this is done using **List Comprehension**. **List Comprehension** is a shorter way of appending elements to a list while in a **for loop**. For example if we have :-

```
for i in <list>:
    <new list>.append(i or <any element>)
```

The above code can we written in one line as :-

```
<newlist> = [i for i in <list>]
```

Now also at the end we defined the **LHS** variable as :-

```
LHS = r"\dfrac{d}{dx}␣\left(" + "␣" + function + "␣" + r"\right)␣" + "␣" +
    "="
func_type = choice_type[ch2 - 1]
```

This is obviously clear that we want to display '$\frac{\mathbf{d}}{\mathbf{dx}}(\mathbf{func}) =$'. The next line stores the function type choice in a variable **func_type** to be later added to database.

13 Indefinite Integration Section

This has the same structure as the previous one. The only difference is the formation of **LHS** defined as follows :-

```
LHS = r"\int␣\left(" + "␣" + integrand + "␣" + r"\right)␣\,␣dx" + "␣" + "=
    "
func_type = choice_type[ch2 - 1]
```

Clearly the LHS is a representation of the equation as '$\int$ **(integrand) dx** ='. The next line is obviously storing the choice to be later used for database.

14 Definite Integration Section

This has the same structure as the previous one. Although this is more of a mathematical formulae being used and subtracting the **lower limit** from **upper limit** according to the **Fundamental Theorem of Calculus**.At the end we have :-

```
LHS_int = r"\int_{lwr_bound}^{upr_bound} \left(" + " " + integrand + " " +
    r"\right) \, dx" + " " + "= $"
LHS_mod = LHS_int.replace("lwr_bound", f"{lwr_bound}")
LHS = LHS_mod.replace("upr_bound", f"{upr_bound}")
func = func_1 + f" from {lwr_bound} to {upr_bound}"
func_type = choice_type[ch2 - 1]
```

The first variable '**LHS_int**' is used for storing the equation '$\int_a^b$ **(integrand) dx** =' where **a** is the **upper limit** and **b** is the lower limit. Next we have '**LHS_mod**' which stores the replaced versions of the limits and finally stores them in the variable '**LHS**'. The later part contains two variables **func** and **func_type** which stores the choices and description to be later used in the database.

15 Graphing Section

The graphing section is differently written if compared to others. Clearly we first have the menu and then a list '**choice_type**' which is '**["Polynomial Function" , "Basic Mathematical Function"]** '. There are two choices given :- One for **Polynomial Functions Visualization** and Other for **Universal Functions Visualization**.

15.1 Polynomial Function Visualization

This function only accepts polynomials and displays them in a better way. It shows the roots , i.e. , the points of the intersection with the x - axis. However it does not work well with polynomial functions with **Complex Roots** as it would display a wrong graph. We first have is the '**deg**' which stores the degree of the polynomial. The **'coefficients'** are initialized in a list. We also have a string '**st**' initialized as follows :-

```
deg = int(input("Enter degree (highest power of polynomial): "))
coefficients = []
st = ""
```

Next we have is a for loop which appends all the coefficients on the basis of degree (the highest power of polynomial), i.e., if the degree is 4 it will accept four coefficients in a downward manner from 4 to 0th degree. After that we take a seperate variable **const** for appending the constant to the **coeff** list.

```
for i in range(deg, 0, -1):
    coeff = int(input(f"Enter the coefficient of x^{i} : "))
      st += str(coeff) + f"x^{i} + "
      coefficients.append(coeff)

      const = int(input("Enter constant (last number) of polynomial : "))
      coefficients.append(const)
```

Meanwhile, in between we also update the **st** variable for displaying it as a label in **Matplotlib**. Next we have finally :-

```
I = st + str(const)
graph_function_for_polynomials(coefficients, I)
```

Here we concatenate the string with the constant and store it in the variable '**I**' which will be used both in the database and as a label as we see in the next line where we called the graphing function.

15.2 Universal Function

This choice doesn't have much things as we only accept the function from the user and then call the function as follows :-

```
I = input("Enter the function: ")
graph_function_for_universal_functions(I)
```

16 Exit Option

The '**os._exit(0)**' method exits / stops the program altogether. Although we used sys.exit() but it doesn't seem to work properly hence we switched to this method. This function needs the **os** module to be imported.

17 Database Appending and Final Calculation and Display

We come to the final section of the main code and we will be going through this sections one by one.

17.1 Database Appending

Since we have tracked the choices of the user, we can easily define a if - else algorithm to append specific data to the database. First we display the normal linear expression of the answer stored in **'I'** and then we define a list **'L'**. The list will contain the information in the form of :-

[<Operation Type>, <Choice - 1>, <Function Type>, <Choice - 2>, <Function (Question)>, <Choice - 3>, <Answer>]

Now we have the following code which is self explanatory:-

```
if ch1 == 1:
    L = ["Differentiation", ch1 , func_type, ch2 , func, ch3, I]
    insert_into_SQL_database(host_name, user_name, passwd, L)

elif ch1 == 2:
    L = ["Indefinite Integration", ch1 , func_type, ch2 ,func, ch3,  I]
    insert_into_SQL_database(host_name, user_name, passwd, L)

elif ch1 == 3:
   L = ["Definite Integration", ch1 , func_type, ch2,  func, ch3, I]
   insert_into_SQL_database(host_name, user_name, passwd, L)

elif ch1 == 4:
   L = ["Graph", func_type, ch1,  I, ch2 , "-", I]
   insert_into_SQL_database(host_name, user_name, passwd, L)
```

On the basis of first choice we are giving the **Operation Type**.
After that we have the LaTeX notation display part. If the **choice** is 1 then its obviously **Differentiation** and hence we perform the LaTeX conversion as :-

```
if ch1 == 1:
     print("Displaying Latex Notation ... ")
     RHS = convert_to_Latex_for_differentials(I, mappings)[1:]
     final_exp = "$" + LHS + " " + RHS
     display_Latex(final_exp)
```

The RHS contains the LaTeX converted expression except the last '**$**' sign since we have it in the **final_exp** already. Then we concatenate it and finally display it. The principle is same for the Integration Part also except for the number conversion in the Definite Integration we convert it to string before hand as :-

```
if type(I) == int or type(I) == float:
      RHS = str(I)
```

Hence we come to the end of the entire **Calculus** code here.

Conclusion & Bibliography

We hope this document was informative for you to understand the mechanism and algorithm of the code. We hope you did not found any discrepencies in the code. If any you can just mail it at our emails provided in the **readme.md** or the Repository Description in Github.

Following were the sources we referred to while writing this documentation :-

- geeksforgeeks.org
- w3schools.com
- python.org
- wikipedia.org
