Yolia Simon - Assignment report:

This program calculates the Taylor series approximation of sin(x) for given angles *x* using *n* terms. First, it prompts the user to enter 5 angles in degrees and a number of terms *n* to use in the Taylor series approximation. It then outputs both the true sin(x) and its Taylor approximation. Additionally, it generates data for angles from 0° to 360° into a .csv file which can then be used in Excel for plotting the Taylor approximation with 6 terms alongside the actual sin(x) function.

Constants:

In this program, I used #include <math.h> to allow me to use mathematical functions such as 'sin()' in my calculations later on. I defined π as a constant equal to 3.14159265 using the '#define' header. This improves readability and ensures accuracy when converting angles from degrees to radians later on. I also used the '#define' header to define the array size as 5. This is to prevent 'magic numbers' within my code. Magic numbers are numbers in the code that appear to be arbitrary without explanation. By replacing arbitrary numbers with defined constants, it reduces the chance of minor errors, improves functionality and makes it easier for the program to be adapted later on if needed.

Functions:

Next, I declared 2 prototype functions outside of the main program: 1. 'double factorial(int n)' and 2. 'double sum_sin_series(double x, int n)'. The former function is the factorial function while the latter function is the function which calculates the Taylor series approximation for sin(x). These functions are defined at the bottom of the program outside of the main program. For both functions, I used the 'double' data type because it allows for higher precision, reduces rounding errors and ensures consistent, smooth results. This is important for approximating trigonometric functions and working with factorials, especially for large values and detailed calculations, where precision is needed to maintain accuracy.

1. The factorial function calculates the factorial of an integer *n*. I used a 'for' loop which iterates from 1 to n and multiplies the variable 'result' by each integer *i* in the range. Using the increment operator '++' to increase the variable by 1, the multiplication accumulates in the 'result' variable. Once the loop is complete, the function returns 'result' which would then hold the factorial of *n*. I used the data type 'int' for *n* so it could only take numerical inputs for n. This function takes an integer and returns as a 'double' data type which allows the function to handle larger values that 'int' or 'float' types would not be able to store without overflowing due to how factorial values grow very quickly. I initialised the variable 'result' to 1.0 instead of 0 because the multiplication would have produced 0 for every value of *n* which is incorrect.

2. The Taylor series approximation function takes an angle *x* (in degrees) and an integer *n* (the number of terms in the series) and calculates the Taylor series approximation for sin(x) up to the *n*th term. First, the function converts the input angle from degrees to radians using this equation: $x = x \times \frac{\pi}{180}$
   I initialised the variable 'sum', which stores the running total of the Taylor series approximation. I set it to 0 so each calculated term is added to this variable in the loop. I used a 'double' data type again here to allow the function to handle larger values. I then used a 'for' loop that iterates *n* times to add each of the terms in the Taylor series expansion. I then initialised a variable called 'term'. The 'pow(-1, i)' section alternates the sign of each term which matches the alternating signs found in the Taylor series. The 'pow(x, 2 * i + 1)' section calculates the power of *x* by using 2i + 1 to calculate the correct odd power for each term (since sin(x) only has odd powers). The 'factorial(2 * i + 1)' section calculates the factorial of the power (which is the denominator) by using the previously defined factorial function. Then, by using the 'add and' assignment operator (+=), 'sum += term' adds each of the calculated terms to the running total 'sum'. Once the loop is complete, the function returns 'sum' which holds the Taylor series approximation for sin(x) using *n* terms.

Main program:

To start with, I declared the array for x values using a 'double' data type variable called 'x_values[array_size]' using the previously defined 'array_size'. Then I used a 'scanf' function inside a 'for' loop to ask the user to enter numerical values of x. The loop runs 5 times (based on 'array_size') to collect each value of x and store it in the variable 'x_values'. Inside the 'for' loop, I initialised a variable of 'int' data type called 'valid_input1' where the 'scanf' function is stored. I used a 'do...while' loop to check that each input is a valid positive number. The 'do...while' loop ensures that if the wrong input is entered, the program will prompt them to re-enter until they provide a valid positive integer. The 'if..while' loop also activates which displays an error message and deletes the incorrect input from the system. 'valid_input1 != 1' checks that 'scanf' successfully read a numerical

input and ‘x_values[i] < 0’ ensures that the input is 0 and above (positive). In the ‘scanf’ function, I used the double format specifier ‘%lf’ instead of ‘%d’ because ‘%d’ caused errors in my code.

The next section in the program asks the user how many terms *n* of the Taylor series to use for the approximation. I took an approach similar to the previous section by initialising a variable of ‘int’ data type called ‘valid_input2’ where the ‘scanf’ function is stored. I used the format specifier ‘%d’ to give a decimal integer input. I also used a ‘do…while’ loop which validates that n is a positive integer above 0. Again, the ‘do…while’ loop ensures that if the wrong input is entered, the program will prompt them to re-enter until they provide a valid positive integer. An error message is also displayed and ‘getchar()’ deletes the incorrect input from the system.

The sin(x) and the Taylor series approximation of sin(x) are then calculated and printed to the console by using a ‘for’ loop that iterates through each of the 5 ‘x_values’ that the user previously provided. Within the ‘for’ loop, I initialised 2 ‘double’ data type variables: ‘sin_x1’ and ‘approx_sin_x1’ (again, ‘double’ was used to handle the large values). ‘sin_x1 = sin(x_values[i] * (PI/180))’ converts each angle from degrees to radians while ‘approx_sin_x1 = sum_sin_series(x_values[i], n);’ calls on the previously defined function ‘sum_sin_series’ to calculate the Taylor series approximation. The output is presented in a clear format, making it easy to compare the actual sine values with the Taylor approximations.

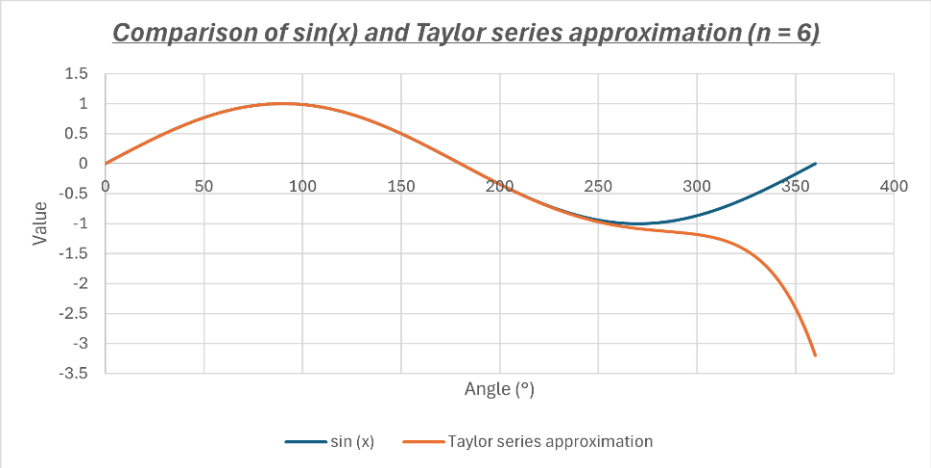
The final section of the program calculates sin(x) and its Taylor approximation for angles from 0° to 360° using 6 terms (n = 6) and creates data in a .csv file which can be used for plotting graphs. ‘FILE * fptr’ and ‘fptr = fopen("sin_taylor_series.csv","w")’ opens a file in write mode called ‘sin_taylor_series.csv’ where the data will be saved. ‘fprintf(fptr,"x, sin(x), Taylor_approx\n")’ prints the heading titles to the .csv file. I used a ‘for’ loop to loop from 0 to 360 and calculate values for each integer angle. I used the format specifier ‘%d’ for the ‘int’ data type variable ‘x’ to print a decimal integer input. I used the format specifier ‘%.5f’ for the ‘double’ data type variables ‘sin_x2’ and ‘appox_sin_x2’ to round them to 5 decimal places. ‘fprintf(fptr,"%d,%.5f,%.5f\n",x, sin_x2, approx_sin_x2)’ writes the angle *x*, sin(x) and the Taylor approximation to the .csv file for each angle. ‘fclose(fptr)’ closes the file once all data has been written and a message is printed to the console that confirms that the values were saved successfully.

Choice of variable names:

‘x_values’ is named that way because it stores the array of the values of *x*. ‘valid_input1’ and ‘valid_input2’ are named that way to avoid confusion between the two input checking methods. ‘sin_x1’ ‘approx_sin_x1’ and ‘sin_x2’ ‘approx_sin_x2’ are named that way to avoid confusion and separate the pairs of variables.

Improvements that could be made: To improve my program, I could enhance the input validation checks by adding checks for out-of range values to prevent overflow issues where the result could potentially exceed the capacity of the ‘double’ data type. To improve the speed, I could have avoided redundant calculations by storing radian values in a separate array after conversion and use that directly. To extend the program further, I could have added an error analysis feature that calculates the difference between sin(x) and the Taylor series approximation of sin(x). This would allow the user to view the approximation accuracy.

Discussion of how well Taylor series approximates sin(x):



With n = 6 terms, the Taylor Series closely approximates sin(x) for small angles around 0° - 250°. However, from 270°, as x approaches 360°, the approximation begins to diverge. This is because the Taylor series is most accurate around the centre of expansion and becomes less accurate as x moves further from this point. The plot visually demonstrates this behaviour, showing that the Taylor series can be an effective approximation in specific ranges but is limited for larger angles when a small number of terms are used.