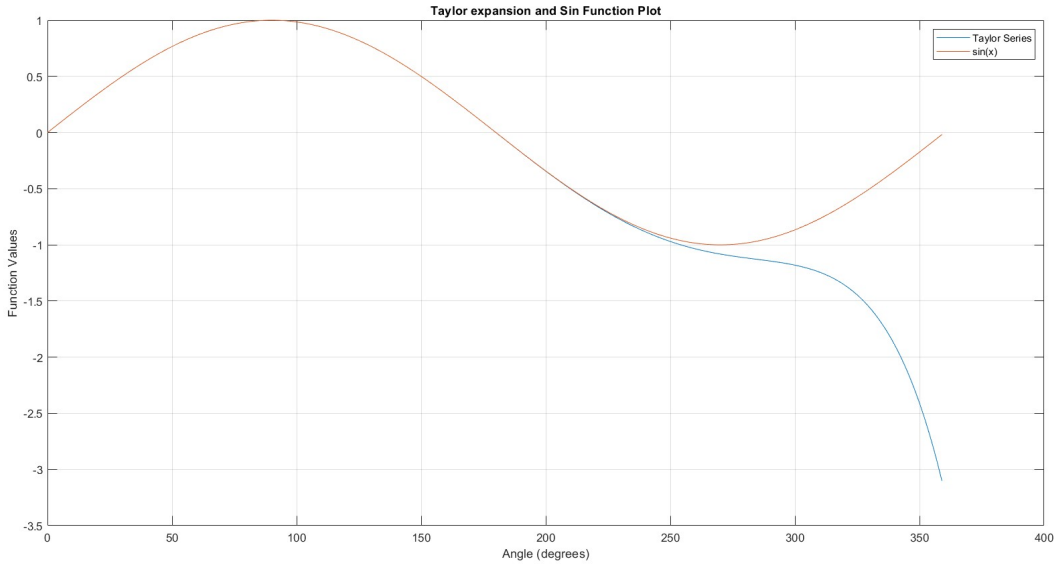

Figure 1: plot showing how the Taylor series of 6 terms compares to the sin function over the interval 0-360°C

Homework Assigntment 2 report: Implementation of the Taylor Series in C and a comparison with the Sin function

[153071]
PHY2027, University of Exeter, U.K.

This program made use of functions in order to carry out the repeated processes of finding factorials of certain terms, as well as combining terms of the Taylor series. This task was completed with the usage of the concept of divide and conquer, with this principle the task was broken down into several key steps in order to create a full solution.

Firstly, a function was developed to find the factorial of a given integer. This was done by taking the passed value **(n terms)** and using backwards iteration and multiplying the total by the current value of the iterator **i** which worked from **n-1** to 0. The product of these terms would then be returned. The checks in place within this function were to ensure that the passed number of terms was greater than 0, if not then the function would return an error. In addition, there was another branch to check if the passed value was 0 or 1, which would return 1, this improves the speed of the function. The function was texted by passing in value and comparing them to the known values.

The second step was to generate a function that computed the Taylor series for the number of terms. This was done by declaring a new function **sum sin series** which took inputs of **n terms** and the point of the approximation. The function first tested to see if the inputed value of 'n' was positive, returning an error message if not. Then the conversion between degrees and radians was written as a variable utilising the modulus function and the pi value given by the **math.h** package. After initialisation the function calculated terms, this was done by multiplying the value of sign by the point that had be converted to radians and raising it to the appropriate index and dividing it by the same powered factorial, which was accomplished by calling the factorial function. After this the sign was multiplied by -1, meaning that the sign would be alternating on each iteration. This was repeated for n terms. These terms would then be added to the result which at the end of the iterations would be returned. This function was tested by separating the two parts of the calculation and working out the result manually. These were then combined and the values were compared to a value given by a handwritten Taylor series.

After this the main function was changed to accept user inputs of 5 point values and take a value

for **n terms**, this was tested to ensure that it was a positive integer, this was done using a do while loop, this ensures an inputed integer is positive, this does however break down if another data type is passed into the statement. The length of the array points was set as a variable **array len** which was set to 5, this avoids setting this value as a magic number which improves the robustness of this program. The main function then called **sum sin series** which calculated the given terms and these were returned to the user along with the true sin value so that the values could be easily compared. After this the number of terms was set to 6 and a 1D array of 360 items in length was created. This array was iterated through and the value was set to the index position degrees' and these were returned to screen for all values along with the true sin value, this sin value was returned in degrees, which required a conversion. After this the array was exported into **MATLAB** where a graph of the interval was plotted Please see Figure 1

When optimising code for the compilers use there is always the danger of hindering the readability and length of code at the cost of performance, however there are several key improvements that could be made to this code. Firstly: By rewriting the factorial function to use a recursive approach where the function would call itself, this would reduce the length of the code and increase speed by improving the memory management. Another improvement would be the complete removal of magic numbers, and the conversion into variables in the code, notably the number 360 with context to the array wave and the for loop that populates it. This not only helps with issues of scalability, error avoidance and readability but also allows the compiler to further optimise the code, as the significance of a given value can be understood and applied. A feature that significantly slows the speed of this program is the nested calculation located in the function sum sin series for loop $sign * pow(radians, 2 * i + 1)$ could be written as a variable that is then divided by the factorial of the correct order, in a similar way to how the radians variable worked. In addition when handling the error statements the switch operator could be used which creates a jump table, which is very fast and does not rely on passing through irrelevant if statements. Further optimisations could be made with the use of performance profiling tools such as GNU profiler which helps to analyse the CPU usage, memory allocation and additional optimisation suggestions.

In reference to figure 1 the approximation for sin(x) is very accurate for the first half period. Then as the approximation continues the negative terms begin to dominate and diverge the two results. Given that this approximation is very accurate for this first half period it could be applied to extend the validity of solutions so that a scalar multiplication could be used outside of the range of 0-180 degrees. The solution would be more accurate with more terms used which would make the approximation hold longer, given the higher degree of the polynomial sum. In order for this approximation to remain accurate longer the number of terms must be increased. As the Taylor series is a converging series it can hold for all points/ values of x but this requires more and more points which would take more processing time and power, meaning a more heuristic approach would be useful for very high values of x.