EECS402 Fall 2025 Project 1

Calculating the sine of an angle is pretty simple when you have a calculator handy, but what if you don't? In this project, you will implement a program to estimate the sine of an angle using a Taylor series. This particular series is known to approximate the sine of an angle quite well, even with a relatively small number of terms in the series. If you are not familiar with Taylor series or even if you're not comfortable with sines of angles, that's ok, as this specification document should provide you with the information you need.

Submission and Due Date

You will submit your program using an email-based submission system by attaching *one C++ source file* only (named exactly "sineApprox402.cpp"), and *one script file* only (named exactly "typescript"). Do not attach any other files with your submission – specifically, do *not* include your compiled executable, etc. The due date for the project is **Tuesday, September 16, 2025 at 4:30pm**. Early submission bonus will be applied as described in the course syllabus.

No submissions will be accepted after the submission final deadline. As discussed in lecture, please double check that you have submitted the correct files (the correct version of your *source code* file named exactly as specified above and the typescript file, correctly generated showing you building your project executable and running valgrind). Submission of the "wrong files" will not be grounds for an "extension" or late submission of the correct files, and will result in a score of 0.

High-Level Description

Calculating the sine of an angle is useful in many mathematical contexts. Typically, we just use a calculator to provide us with an accurate result, or we may use a computer program or library to compute it. While there are numerous ways to obtain the sine of an angle in the real world, for this project, you will be implementing a simple program that do it, without relying on any other programs or libraries (note: NO libraries are allowed to be "#include"d except for <iostream> for this project – be careful not to "#include", or use any functionality from other C++ libraries such as <cmath>, which would be a specification violation resulting in a grade of 0).

The Taylor series most commonly used for approximating the sine of an angle is as follows:

$$\sin(angleRadians) \approx angleRadians + \sum_{n=1}^{\infty} \frac{(-1)^n}{(2*n+1)!} * angleRadians^{2*n+1}$$

Figure 1: Taylor series used to approximate sine of an angle

That complicated-looking formula is really just a sum of terms that involve raising values to a power and computing the factorial of values – both of these are easily implemented without relying on any libraries. While the formula shows the summation going to infinity, we are going to limit the number of terms in this series to be 5 or less for this project (the reason for this is because of that factorial – if we

allowed n to be 6, we would need to compute `13!` which is too large of a value to store in an `int` data type!)

In the above formula, we are using the value of an angle in units of radians, but it is quite common for angles to be expressed in units of degrees. Conversion from degrees to radians is quite simple, and is shown here:

$$angleRadians = angleDegrees * \frac{\pi}{180}$$

Figure 2:Converting an angle in degrees to radians

Note: since we are not allowing any libraries other than <iostream> to be #include'd you'll need to provide your own value for π. For the purposes of this project, you should have this global constant defined *exactly* as shown below, and then utilize that named constant when your program needs to use the value of π:

```
const double PI_VALUE = 3.14159265359;
```

Now, let's say we want to approximate the sine of the angle 58° using 4 terms in the Taylor series described above. First, we convert the angle from degrees to radians to get an angle of about 1.012290 radians. Then, we use the formula in Figure 1 with a total of 4 terms (NOTE: we are counting the first "angleRadians" as a term, so we use that plus 3 additional terms via the summation). Here is what those calculations look like:

First Term | Second Term | Third Term | Fourth Term

$$\sin(1.012290) \approx 1.012290 + \frac{(-1)^1}{(2*1+1)!} * 1.012290^{2*1+1} + \frac{(-1)^2}{(2*2+1)!} * 1.012290^{2*2+1} + \frac{(-1)^3}{(2*3+1)!} * 1.012290^{2*3+1}$$

$$\sin(1.012290) \approx 1.012290 + \frac{-1}{6} * 1.037328 + \frac{1}{120} * 1.0629842 + \frac{-1}{5040} * 1.08927498$$

$$\sin(1.012290) \approx 1.012290 - 0.172888 + 0.0088582 - 0.000216126$$

$$\sin(1.012290) \approx 0.848045$$

Figure 3: Computations performed to approximate the sine of 58 degrees using 4 terms

If I use my calculator to compute sin(58°) I get 0.848048, so it looks like that is a pretty decent approximation.

With the above description, you will write an interactive C++ program that will use loops, branching, and functions to approximate the sine of an angle using the exact approach described. While there may be many ways of accomplishing this, you will only receive credit for the project if you implement it using the required functions (shown below), and doing the computations as shown above.

You may notice there are some ways of combining things in the math shown or computing things a different way. For example, the numerator of the fraction inside the summation is -1 or +1 depending on which term you are computing. We could determine that by asking whether the term number is even or odd and using the appropriate value – but for the purposes of this project, you must compute the value by raising the value -1 to the appropriate power. Similarly, you may notice that the entire fraction inside the summation does not depend on "angleRadians" at all. Since we are only allowing up to 5 terms, we could just store those coefficients as pre-determined constants and avoid doing the math at runtime – but for the purposes of this project, you must perform the computations as shown using the required functions (shown below). Avoid the temptation to be "clever" in the way you implement this project, as we need your computations to be the same as our computations so that we get the exact same floating point results when grading your implementation!

Requirements

The following global functions are required for this project. You may not modify the function prototype as given for ANY reason. This includes changing the order of parameters, the types of parameters or return values, or renaming the function in ANY way. When describing functions, words in "quotation marks" usually refer to the parameters that are passed in.

```
double degreesToRadians(const double angleDeg);
```

This function converts the value of "angleDeg" (which will be an angle, specified in degrees) to radians and return the result.

```
bool toThePower(const double baseVal, const int exponentVal, double&
outResult);
```

This function raises "baseVal" to the power "exponentVal" and stores the result in the reference parameter "outResult". For example, if "baseVal" is 4.0 and "exponentVal" is 3, then "outResult" would be computed as 64.0. No input or output is done in this function! This function ONLY supports non-negative exponents – any negative exponent results in function failure. The function will return true on success, and false on failure. When the function fails and false is returned, the output value of "outResult" is undefined (note: this just means that outResult may be any value, as no specific value is specified on failure). Remember, you may NOT use anything from the math library, so don't call the "pow" function in your implementation, even if you know how.

```
bool computeFactorial(const int inVal, int& outFactorial);
```

This function computes the factorial of "inVal" and assigns the result to the reference parameter "outFactorial". This function only supports input values from 0 to 12 (inclusive) – any input value outside of that range results in function failure. The function will return true on success, and false on failure. When the function fails and false is returned, the output value of "outFactorial" is undefined.

`bool approximateSine(const double angleRad, const int numTerms, double& outSineVal);`

This function will approximate the value of the sine of "angleRad", an angle specified in units of radians, using a Taylor series using the number of terms specified via "numTerms". The approximation result is stored in the output reference parameter "outSineVal". This function supports input angles of any value, but the Taylor series used is only valid for angles in the range -π to +π, so if the input angle is out of that range, then this function will add or subtract 2π as needed such that the angle ends up being in the range -π to +π. The function will return true on success, and false on failure. This function fails when either numTerms is outside the allowed range (1 to 5 inclusive), or if any of the mathematical functions it relies on (toThePower and computeFactorial) fails during the computations. When the function fails and false is returned, the output value of "outSineVal" is undefined.

`int main();`

The main function will start by prompting the user for some inputs. First, the user is prompted for whether their input angle will be specified in degrees or radians – this will allow the user to provide their angle in whichever units are more convenient for them. If the user does not respond properly then this is considered a failure. If the user did respond properly then the user is prompted for the angle they want to approximate the sine value for. There are no restrictions placed on the value of the angle. Next, the user is prompted for the number of Taylor series terms to use when approximating the value. If the user doesn't respond with a valid number of terms, then this is considered a failure. If the user did respond properly, then the sine of the angle specified is approximated, and if successful, the approximated value is output to console. Finally, if everything was successful, this function will then compute the approximate sine using all other allowed number of terms and output those approximations as well. This is done primarily to allow the user to see how different numbers of terms would affect the approximated value. Upon any failure, as described here, the remainder of this function is effectively skipped and the function simply prints "Unable to provide results due to invalid inputs!" and ends.

This list of functions is completely exhaustive. You may not implement *any* additional functions for *any* reason. You also may *not* choose to leave out any number of the required functions for any reason.

Special Testing Requirement

In order to allow the course staff to easily test parts of your implementation using our own main() function in place of yours, some special preprocessor directives will be required. Don't worry about what all this means right now – it should not affect anything that you do or how your program acts, but is useful for course staff during grading. Immediately before the definition of the main() function, (but after ***all*** other prior source code), include the following lines EXACTLY:

```
#ifdef ANDREW_TEST
#include "andrewTest.h"
#else
```

and immediately following the main() function, include the following line EXACTLY:

```
#endif
```

Therefore, your source code should look as follows:

```
library includes
program header
constant declarations and initializations
global function prototypes with comments
#ifdef ANDREW_TEST
#include "andrewTest.h"
#else
int main()
{
  implementation of main function
}
#endif
global function definitions
```

Lines above in red are to be used exactly as shown. Other lines simply represent the location of those items within the source code file.

Additional Information

- The program must exit "gracefully", by reaching the ONE and only return statement that should be the last statement in the main function - do not use the "exit()" function or additional “return”s in main.
- Any floating point values you need should be declared of type "double" - do NOT use any "float" variables in this program.
- You don’t need to worry about number formatting. Don’t use setprecision or anything similar. Just let cout print the values the way it wants to. This is important as it will ensure your program prints out numbers the way our grading expects them to be printed.
- The only library you may #include is <iostream>. No other header files may be included, and you may not make any call to any function in any other library (even if your IDE allows you to call the function without #include'ing the appropriate header file). Be especially careful not to use any function from the <cmath> library.
- All user input (via cin) and all output to console (via cout) must be done in the main function – do not do any input or output in any other function for this project.
- When implementing your solution, use only topics that we’ve discussed in lecture by the date that the project was posted, or that course staff has explicitly allowed in writing. Experienced programmers may identify ways to solve problems in the project using more advanced topics that had not been discussed in lecture by the date the project was posted, but you may not utilize those techniques!
- Remember, “magic numbers” are bad and need to be avoided
- Remember, “duplicated code” is bad and needs to be avoided

- Remember, identifier naming is important. Name your variables, constants, etc., using a descriptive name using the style described in lecture
- Remember, you must ensure your program builds successfully using the c++98 standard specifically
- Remember, you must run valgrind to ensure it does not identify any issues in the execution of your program

Error Checking

You need not worry about error checking the type of the user input in this program. You **may** assume that the user will enter data of the correct data type when prompted (in other words, when the program expects the user to enter an integer value, we will only enter integer values, etc.). Note: This is usually a horrible assumption. As you learn error handling techniques in this class, you will be required to use them, but for this project, just concentrate on implementing the functions as described. During your testing, if you accidentally input a value with an incorrect type, your program will likely act in an unexpected way. That is "normal" and we will learn how to overcome that later in the course.

You are required to perform error check on certain values that the user enters as input. For example, the user is prompted for a whether they want to provide an angle in degrees or radians. We promise that we will only enter a char value at that prompt, but the char value we enter may not be a valid option, and your program needs to recognize that situation. Similarly, when providing the number of terms to utilize, only values in a specific range are allowed. Again, we will input an int value as expected, but you need to perform some error checking to ensure that the value provided falls within the allowed range.

Design and Implementation Details

As mentioned above, for this project, you are required to implement the project *exactly* as described here using the *exact* output strings, including punctuation and spacing, provided in the sample outputs. Make sure you implement and utilize the exact functions specified and do not add any additional functions, parameters, etc. You will likely have more "freedom" in your design and implementation as the course progresses. This effectively means that you do not have any control over the general program design for this project.

It is strongly recommended that you implement this program in a piece-wise fashion. That is, start with the degreesToRadians() function, which should be the simplest one to implement. Write a main() to go along which calls the function to see that it works as expected (i.e. write a "driver program" for the sole purpose of testing your degreesToRadians function). Test your program at this point. Once this is all working, you could add the code in main to prompt the user for degrees or radians and then prompt for the angle and, if they indicated degrees, call the degreesToRadians function you've written and tested to be sure. Note: if following this suggested implementation, you have not yet written any of the functionality related to approximating sines, etc., you are simply setting up a framework for the program and making sure it works as you go. Once you've added that code, run your program and provide some

user inputs to ensure your program accepts inputs properly and can successfully call the degreesToRadians function and output the results.

Once that is fully working implement another function (or, if the function is relatively complex, implement a part of a function) and then test the new functionality. Continue implementing and testing little bits of additional functionality at a time until the project is completed. To help you get used to this, I've described the functions in the order I would recommend implementing them.