Assignment 4 – Calc Report Template

Jackson Dawson

CSE 13S – Winter 24

Purpose

Audience for this section: Pretend that you are working in industry, and write this paragraph for your boss. You are answering the basic question, “What does this thing do?”. This section can be short. A single paragraph is okay.

Do not just copy the assignment PDF to complete this section, use your own words.

This program provides the ability to compute a number of mathematical operations. This includes addition, multiplication, subtraction, division, as well as more complicated functions such as sin, cos, tan, and finding the absolute value of a given integer. It utilizes a stack to compute simple operations using reverse polish notation, and iterative series logic to approximate complex functions. The program and helper functions are well tested, which can be compiled using the Make file. For the complex functions, parameters are given by the user in radians, and returned as a value between 0 and 2*pi.

Questions

Please answer the following questions before you start coding. They will help guide you through the assignment. To make the grader’s life easier, please do not remove the questions, and simply put your answers below the text of each question.

- Are there any cases where our sin or cosine formulae can’t be used? How can we avoid this?
  Our sine and cosine formulae may not be usable in cases where the input angle is very large, leading to numerical instability due to the repetitive nature of calculating sine or cosine using their series expansions. To avoid this, we can normalize the input angle by using the periodicity of sine and cosine functions, bringing the angle within a range where the series converges more rapidly and reliably.
- What ways (other than changing epsilon) can we use to make our results more accurate? [1]
  We can utilize trigonometric identities to reduce the computation of sine or cosine for larger angles to those of smaller angles. For example, using the identity sin(x) = sin(x - 2*pi*k) and cos(x) = cos(x - 2*pi*k) for any integer k can help reduce the input angle to a more manageable range.
- What does it mean to normalize input? What would happen if you didn’t?
  If the input wasn’t normalized, our calculation would be forced to compute something much larger than is necessary. Every angle can be represented by some value between 0 and 2*pi. Thus, by normalizing it as its equivalent angle between these values, the computation can be made much more efficient.
- How would you handle the expression 321+? What should the output be? How can we make this a more understandable RPN expression?
  This expression should evaluate to ”3.00... 3.00...”. The three numbers should be pushed onto the stack, then the top two will be added upon the plus sign’s execution. I suppose a more understandable RPN expression would just be ”3 3”, but it is fine as is.

[1]hint: Use trig identities

- Does RPN need parenthesis? Why or why not?
  No, RPN does not need parenthesis. Since the order that each number and operator is inputted specifies the order in which each calculation occurs, the purpose for using parenthesis is made null.

Testing

List what you will do to test your code. Make sure this is comprehensive. [2] Be sure to test inputs with delays and a wide range of files/characters.

Testing the basic calculations and the complex functions separately will be important. To test the stack calculations, I will try a variety of inputs, such as 2 digit addition/subtraction/etc, as well as multi-digit inputs. I will try characters that are invalid, as well as no characters at all. Trying input with only operators vs only digits will also be useful. For the complex functions, I will try inputs within the bounds of 0 to 2pi, as well as inputs outside of this. I will not compare the results against the standard libraries to check for accuracy within a range, as they may not be available in the grading pipeline.

How to Use the Program

Audience: Write this section for the user of your program. You are answering the basic question, "How do I use this thing?". Don't copy the assignment exactly; explain this in your own words. This section will be longer for a more complicated program and shorter for a less complicated program. You should show how to compile and run your program. You should also describe any optional flags or inputs that your program uses, and what they do.

To show "code font" text within a paragraph, you can use `\lstinline{}`, which will look like this: `text`.

For a code block, use `\begin{lstlisting}` and `\end{lstlisting}`, which will look like this:

```
Here is some code in lstlisting.
```

And if you want a box around the code text, then use `\begin{lstlisting}[frame=single]` and `\end{ lstlisting}`
which will look like this:

```
Here is some framed code (lstlisting) text.
```

Want to make a footnote? Here's how.[3]

Do you need to cite a reference? You do that by putting the reference in the file `bibtex.bib`, and then you cite your reference like this[1][2][3].

Start by compiling the program to make it usable. This is achieved by typing the following commands into the terminal:

```
make format
make
```

Now the program "calc" is ready to use. Now run "calc" to start computing.

```
./calc
```

You will now be prompted to input a single-line mathematical expression in Reverse Polish Notation.
e.g.

```
> 3 2 +
```

This expression will print "5.0000000000". But you can do more advanced calculations as well! Such as:

```
> 20 s
```

[2]This question is a whole lot more vague than it has been the last few assignments. Continue to answer it with the same level of detail and thought.
[3]This is my footnote.

This will print the result of sin(20) = 0.9129452507

To end the program's execution, press:

```
> {CTRL + d}
```

To clean up after yourself, do:

```
make clean
```

You're all done!

Program Design

Audience: Write this section for someone who will maintain your program. In industry you maintain your own programs, and so your audience could be future you! List the main data structures and the main algorithms. You are answering the basic question, "How is this thing organized so that I can have a chance of fixing it?". This section will be longer for a more complicated program and shorter for a less complicated program.

Main Data Structures:

- A stack (implemented as an array) is used to manage numbers during calculation. This stack allows for last-in, first-out (LIFO) operations, which are essential for handling the operands in mathematical expressions.
- Function pointers to unary and binary operations are used to abstract the calculation operations. This allows for flexibility in the implementation of operations and the potential to extend the calculator's functionality.

Main Algorithms:

- Parsing input strings to identify numbers and operators, handling them appropriately based on their type (binary or unary operations, numbers).
- Mathematical functions both using the C standard library ('libm') and custom algorithms for trigonometric and other basic math functions. Such functions include Sine, Cosine, Tangent, Absolute Value, and Square-Root.

Pseudocode

Give the reader a top down description of your code! How will you break it down? What features will your code have? How will you implement each function.

The program follows a structured flow:

1. Parse command line options to determine if standard math library functions should be used.
2. Continuously prompt the user for input until termination (e.g., EOF).
3. Process each line of input by:
   - Splitting it into tokens (numbers and operators).
   - For each token, determine if it's a number or operator.
   - If it's a number, push it onto the stack.
   - If it's an operator, apply it to the appropriate number(s) from the stack.
   - Handle errors appropriately at each step.
4. Output the final stack contents after processing each line.

Function Descriptions

For each function in your program, you will need to explain your thought process. This means doing the following

- The inputs of every function (even if it's not a parameter)
- The outputs of every function (even if it's not the return value)
- The purpose of each function, a brief description about a sentence long.
- For more complicated functions, include pseudocode that describes how the function works
- For more complicated functions, also include a description of your decision making process; why you chose to use any data structures or control flows that you did.

Do not simply use your code to describe this. This section should be readable to a person with little to no code knowledge. **DO NOT JUST PUT THE FUNCTION SIGNATURES HERE. MORE EXPLANATION IS REQUIRED.**

- **NormalizeAngle(double x)**: Normalizes an angle to the range [0, 2PI). Inputs include the angle x in radians. Outputs the normalized angle. The purpose is to ensure trigonometric functions operate within a standard range for consistency and accuracy.
- **Abs(double x)**: Returns the absolute value of x. Inputs include the number x. Outputs the absolute value. It provides a basic mathematical function used in various operations within the program.
- **Sqrt(double x)**: Calculates the square root of x using the Newton-Raphson method. Inputs include the number x. Outputs the square root.
- **Sin, Cos, Tan(double x)**: Compute the sine, cosine, and tangent of x, respectively. Inputs include the angle x in radians. Outputs the trigonometric result. These functions showcase both normalization of angles and iterative calculation methods for trigonometric functions.
- **main(int argc, char *argv[])**: The entry point of the program. Handles command-line arguments, input prompting, and processing calculations. It integrates all components of the program, coordinating user input, calculation processing, and output.
- **isNumber(const char *str)**: Checks if a given string represents a valid number. The input is a string pointer `str`. The output is a boolean value, returning 1 if the string is a number and 0 otherwise. The purpose is to validate inputs before processing them as numbers.
- **parseCommandLineOptions(int argc, char *argv[], int *use_libm)**: Parses command-line options to configure the program's behavior, particularly whether to use the standard math library (`libm`) for trigonometric functions. Inputs include the argument count `argc`, argument vector `argv[]`, and a pointer to the `use_libm` flag. Outputs are the setting of the `use_libm` flag based on user input and potentially terminating the program if the help option is selected or invalid options are provided.
- **promptForInput(void)**: Displays a prompt to the user, indicating the program is ready to accept input. There are no inputs or outputs to this function.
- **processCalculations(char *input, int use_libm)**: Coordinates the parsing and calculation of user input. Inputs include the user's input string `input` and the `use_libm` flag indicating whether to use the standard math library. The output is the display of calculation results or error messages. This function is the core of the program, orchestrating the flow from input to processed result, handling numbers, operators, and errors according to the program logic.
- **handleNumber(char *token, int *errFlag)**: Processes a token identified as a number, pushing it onto the calculation stack. Inputs include the token string `token` and a pointer to the error flag `errFlag`. Outputs include potentially updating the error flag if the stack is full.

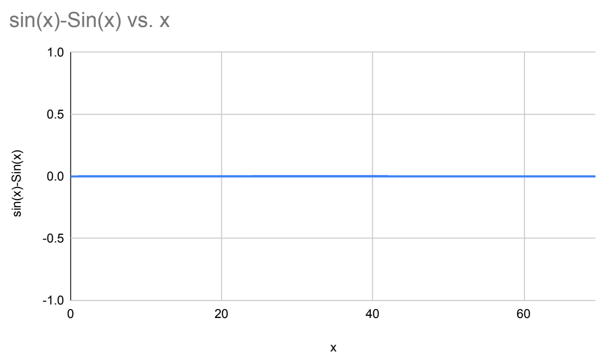

Figure 1: sin(x) - Sin(x)

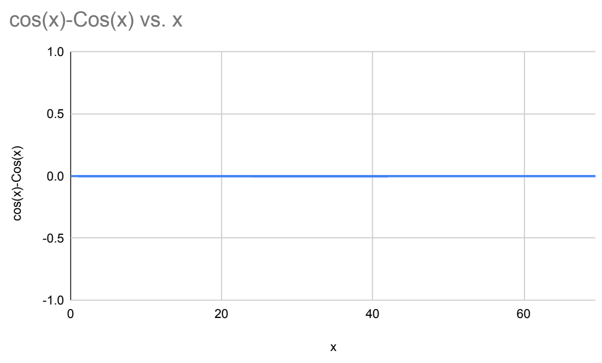

Figure 2: cos(x) - Cos(x)

- **handleBinaryOperator(char token, int *errFlag)**: Executes a binary operation based on the operator token provided. Inputs include the operator token `token` and a pointer to the error flag `errFlag`. The function determines which binary operation to perform and applies it to the top two numbers on the stack.
- **handleUnaryOperator(char token, int use_libm, int *errFlag)**: Executes a unary operation based on the operator token and the `use_libm` flag. Inputs include the operator token `token`, a flag indicating whether to use the standard math library `use_libm`, and a pointer to the error flag `errFlag`.
- **handleErrors(char *token, int *errFlag)**: Manages error reporting based on the input token and updates the error flag. Inputs include the problematic token `token` and a pointer to the error flag `errFlag`. Outputs include printing error messages specific to the type of error encountered (bad character or string).

Results

Follow the instructions on the pdf to do this. In overleaf, you can drag an image straight into your source code to upload it. You can also look at `https://www.overleaf.com/learn/latex/Inserting_Images`

References

[1] Wikipedia contributors. C (programming language) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/C_(programming_language), 2023. [Online; accessed 20-April-2023].

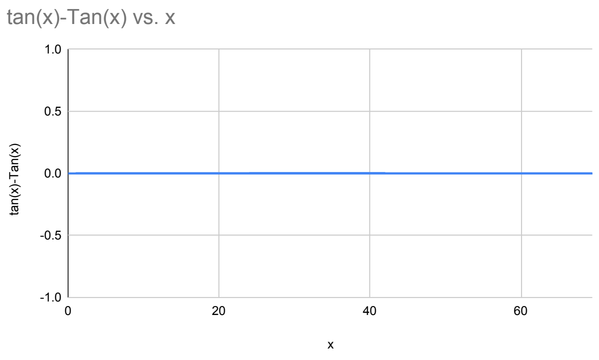

Figure 3: tan(x) - Tan(x)

[2] Robert Mecklenburg. *Managing Projects with GNU Make, 3rd ed.* O'Reilly, Cambridge, Mass., 2005.

[3] Walter R. Tschinkel. Just scoring points. *The Chronicle of Higher Education*, 53(32):B13, 2007.