Assignment 3 – Calculator

Sebastian Avila

CSE 13S – Fall 2023

Purpose

The purpose of this program is to implement a calculator. The calculator takes input in Reverse Polish Notation. This means that the program takes numerical values first, followed by an operator.

How to Use the Program

To use this program, there are several files that must be included in a single directory: `Makefile`, `mathlib.h`, `messages.h`, `operators.h`, `stack.h`, `calc.c`, `mathlib.c`, `operators.c`, and `stack.c`. With these files in the same directory, use command `make clean` to remove any preexisting object files. Then, use command `make format` to format all the header and source files. Then, to compile the program, use command `make all`. After the program successfully compiles, use `./calc` to run the program using custom trig function implementations. The program should begin running and wait for user input. To use the calculator, enter an expression in Reverse Polish Notation and press enter. Valid operations are as followed: (s) sin, c (cos), t (tan), a (absolute value), r (square root), + (addition), - (subtraction), * (multiplication), / (division), and % (modulus). The program will print the answer and then wait for input again. To end the program press `^c` or `^d`.

The program has three different command line argument options:

- `./calc -h`: This prints instructions to the screen. These instructions describe the usage of the program. It shows what the different command line arguments do.
- `./calc -m`: This makes the program use libm trig function implementations instead of custom ones.
- `./calc`: When given no arguments, the program uses custom trig function implementations.

The program uses several optional compiler flags:

- `-Wall`: This flag enables all warning messages.
- `-Werror`: This flag turns all warnings into errors.
- `-Wextra`: This flag enables extra warning flags that are not enabled by `-Wall`.
- `-Wstrict-prototypes`: This flag warns if a function is declared or defined without specifying the argument types.
- `-pedantic`: This flag issues all the warnings demanded by strict ISO C and ISO C++.
- `-lm`: This flag links the math.h library. This allows the program to access and use the functions from the math.h library.

Program Design

The program begins by checking the command line arguments. The options are listed in the How To Use the Program section. If an in correct argument is received, the program prints an error message and usage message. The program sets a pointer to a constant variable of type `unary_operator_fn` to either `my_unary_operators` or to `libm_unary_operators`, depending on whether the command line argument `-m` was inputted or not. After this, the program prompts for input. Input is read by the program using `fgets` with a buffer size of 1024 bytes. This input is then spilt into single characters using `strtok_r()`. Each token is then sent to a function to check whether it is a double. If it is the function stores the double in a location given to it and returns true. This double is then pushed to a stack. If the function returns false, the program checks if the token is an operator. If it is an operator, the program checks whether it is a unary operator or binary operator. It accesses the correct operator using a constant array, then sends the operator function to another function which applies the operator and pushes the result to the stack. If the operation does not complete successfully, the function returns false and an error message is printed. If `EOF` is received, input stops being read and the stack is printed. If `EOF` is received again, the program prints the stack and exits. Once the input has been parsed, the program prompts for input again.

Algorithms

Sin

This program estimates the value of sin(x) using the Maclaurin Series.

$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} + ... \tag{1}$$

The program first normalizes x to within 0 to 2π. The summation is calculated using a while loop. Rather than going to infinity, the program continues summing until the term is less than a number epsilon (ϵ). It creates variable to represent different parts of the Maclaurin series. The variable `sign` represents $(-1)^n$. Every loop, it multiplies itself by -1 to switch between positive and negative as it does in the Maclaurin series. The variables `power` and `power_multiplier` represent x^{2n+1}. With each loop, `power` is multiplied with `power_multiplier` to get x^3, x^5, x^7, etc. The variable `previous_factorial` represents the factorial that is skipped between each term. For example between the terms $\frac{x^3}{3!}$ and $\frac{x^5}{5!}$, the factorial 4! is skipped. The skipped factorial is calculated by $2n$ each loop after n is incremented. The variable `factorial_value` represents what the current factorial is equal to. It uses itself and `previous_factorial` and $2n+1$ (from $2n+1$ in $(2n+1)!$ in the series) to calculate its value each loop. The variable `n` is the same as n in the series. The term is calculated in the same way it is in the summation: $\frac{(-1)^n}{(2n+1)!} x^{2n+1}$ or $\frac{\text{sign}*\text{power}}{\text{factorial\_value}}$. Term is initialized equal to x because the first term of the series is x. `n` is initialized to 1 because the loop starts with adding the first term (which is already known) to the result, then calculating the next term which is when $n = 1$.

```
double Sin(double x)
    x = x % 2pi
    if x < 0 then
        x = x + 2pi

    result = 0
    sign = 1
    term = x
    power = x
    power_multiplier = x * x
    factorial_value = 1
    previous_factorial = 2
    n = 1
```

```
    while |term| >= EPSILON
        result = result + (sign * term)
        sign = sign * -1
        factorial_value = factorial_value * previous_factorial * (2 * n + 1)
        power = power * power_multiplier
        term = power / factorial_value
        n = n + 1
        previous_factorial = 2 * n

    return sum
```

Cos

This program estimates the value of cos(x) using the Maclaurin Series.

$$\cos(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} + ... \tag{2}$$

The program first normalizes x to within 0 to 2π. The summation is calculated using a while loop. Rather than going to infinity, the program continues summing until the term is less than a number epsilon (ϵ). It creates variable to represent different parts of the Maclaurin series. The variable `sign` represents $(-1)^n$. Every loop, it multiplies itself by -1 to switch between positive and negative as it does in the Maclaurin series. The variables `power` and `power_multiplier` represent x^{2n}. With each loop, `power` is multiplied with `power_multiplier` to get x^2, x^4, x^6, etc. The variable `previous_factorial` represents the factorial that is skipped between each term. For example between the terms $\frac{x^2}{2!}$ and $\frac{x^4}{4!}$, the factorial 3! is skipped. The skipped factorial is calculated by $2n + 1$ each loop before `n` is incremented. The variable `factorial_value` represents what the current factorial is equal to. It uses itself and `previous_factorial` and $2n$ (from $2n$ in $(2n)!$ in the series) to calculate its value each loop. The variable `n` is the same as n in the series. The term is calculated in the same way it is in the summation: $\frac{(-1)^n}{(2n)!} x^{2n}$ or $\frac{\text{sign}*\text{power}}{\text{factorial\_value}}$. Term is initialized equal to 1 because the first term of the series is x. `n` is initialized to 1 because the loop starts with adding the first term (which is already known) to the result, then calculating the next term which is when $n = 1$.

```
double Cos(double x)
    x = x % 2pi
    if x < 0 then
        x = x + 2pi

    result = 0
    sign = 1
    term = 1
    power = x * x
    power_multiplier = x * x
    factorial_value = 1
    previous_factorial = 1
    n = 1

    while |term| >= EPSILON
        result = result + (sign * term)
        sign = sign * -1
        factorial_value = factorial_value * previous_factorial * 2 * n
        term = power / factorial_value
        previous_factorial = 2 * n + 1
        n = n + 1
        power = power * power_multiplier
```

```
    return result
```

Tan

To implement tan(x) the program uses the functions it contains for sin(x) and cos(x). This is possible because Tan($\frac{\pi}{2}$) will not be undefined. This is true because of two reasons: the IEEE 754 double precision standard cannot perfectly represent pi in binary and the Taylor-Maclaurin series used to approximate Cos($\frac{\pi}{2}$) will not converge to 0. Therefore, it will instead to converge to a very, very small number resulting in a very, very large value of Tan($\frac{\pi}{2}$). This reasoning is given in the assignment pdf. [1]

```
double Tan(double x)
    sin = Sin(x)
    cos = Cos(x)
    result = sin / cos
    return result
```

Sqrt

The program implements $\sqrt{x}$ by using the Babylonian method. In this method, you start with an initial guess. Then calculate the next guess using the formula: newguess = (oldguess + x/oldguess)/2. You then repeat the calculation using the new guess until the difference between the current guess and the previous guess is smaller than a specified tolerance value. In the program, it first checks if x is greater than 0. If it is not, then the function return `nan`. This is because you can only take the square root of positive numbers. The program sets initial guess as a variable `old` and which is initialized to 0. The new guess is a variable `new` and is also initialized to 0. The function loops and repeats the formula for the new guess until the absolute value of the `new` − `old` is less than Epsilon (ϵ) which is the tolerance value.

```
double Sqrt(double x)
    if x < 0
        return nan

    double old = 0.0
    double new = 0.0

    while (absolute value of (old - new) > EPSILON)
        old = new
        new = 0.5 * (old + (x / old))

    return new
```

Function Descriptions

stack

- `bool stack_push(double item)`: This function takes in a double and returns either true or false (1 or 0). It adds the parameter `item` to the top of the stack and increases the stack size.

```
bool stack_push(double item)
    if stack_size == STACK_CAPACITY then
        return false
    else
        stack[stack_size] = item
```

```
        increment stack_size
        return true
```

- `bool stack_peek(double *item)`: This program takes in a pointer `item`. It returns false if the stack is empty. Otherwise, it sets the pointer to the top of the stack and return true.

```
bool stack_peek(double *item)
    if stack is empty then
        return false
    else
        *item points to top of stack
        return true
```

- `bool stack_pop(double *item)`: This function takes in a pointer `item`. It returns false if the stack is empty. Otherwise, it sets the pointer to the top of the stack, decreases the stack size by 1, and returns true.

```
bool stack_pop(double *item)
    if stack is empty then
        return false
    else
        *item points to top of stack
        decrease stack_size by 1
        return true
```

- `void stack_clear(void)`: This function takes no parameters and does not return anything. It sets the stack size to 0.

```
void stack_clear(void)
    set stack_size to empty
```

- `void stack_print(void)`: This function takes no parameters and does not return anything. It prints the elements of the stack from bottom to top. This function is from Assignment 3: Calculator [1]

```
void stack_print(void)
    if stack is empty
        return

    print the first element with 10 decimal places
    for i, 1 to i < stack_size
        print stack[i] with 10 decimal places and a space before
```

mathlib

- `double Abs(double x)`: This function takes in a double `x` and returns a double `result`. Its purpose is to calculate the absolute value of x and return it.

```
double Abs(double x)
    if x >= 0 then
        result = x
        return result
    else
        result = -1 * x
        return result
```

- `double Sqrt(double x)`: This function takes in a double `x` and returns a double `result`. If `x` is not positive, the function returns `nan`. Otherwise, this function calculates the square root of x and returns the result. This function was given in the assignment instructions. [1] The explanation of this algorithm and its psuedocode is in the Algorithms section of this report.
- `double Sin(double x)`: This function takes in a double `x` and returns a double `result`. The double `x` is an angle in radians. It calculates sin(x) and returns the result. The explanation of this algorithm and its psuedocode is in the Algorithms section of this report.
- `double Cos(double x)`: This function takes in a double `x` and returns a double `result`. The double `x` is an angle in radians. It calculates cos(x) and returns the result. The explanation of this algorithm and its psuedocode is in the Algorithms section of this report.
- `double Tan(double x)` This function takes in a double `x` and returns a double `result`. The double `x` is an angle in radians. It calculates tan(x) and returns the result. The explanation of this algorithm and its psuedocode is in the Algorithms section of this report.

operators

- `bool apply_binary_operator(binary_operator_fn op)`: This function takes in an operator and accesses the global stack. It then pops the first 2 elements on the stack, and calls the op function with those as its arguments (the first element popped is the right-hand side, and the next element popped is the left-hand side). Finally, it pushes the result to the stack. [1] If there are not two elements to pop, the function will return false. It will return true on success.

```
bool apply_binary_operator(binary_operator_fn op)
    if stack_size < 2 then
        return false

    assert stack pop(arg1) true
    asser stack pop(arg2) true
    result = op(arg2, arg2)
    assert stack push(result) true
    return true
```

- `bool apply_unary_operator(unary_operator_fn op)` This function takes in an operator, and accesses the global stack. It then applies the operator to the first element on the stack and pushes the result to the stack. If there are not enough elements to pop, it returns false or an error. This function is given in the assignment instructions. [1]

```
bool apply_unary_operator(unary_operator_fn op)
    if stack_size < 1 then
        return false

    double x
    assert stack_pop(x) true
    double result = op(x)
    assert stack_push(result) true
    return true
```

- `operator_add(double lhs, double rhs)`: This function takes in two doubles: lhs and rhs. Its calculates the sum of the lhs (left hand side) and rhs (right hand side) and returns the result.

```
operator_add(double lhs, double rhs)
    result = lhs + rhs
    return result
```

- `operator_sub(double lhs, double rhs)`: This function takes in two doubles: lhs and rhs. Its calculates the difference of the lhs (left hand side) and rhs (right hand side) and returns the result.

```
operator_sum(double lhs, double rhs)
    result = lhs - rhs
    return result
```

- `operator_mul(double lhs, double rhs)`: This function takes in two doubles: lhs and rhs. Its calculates the product of the lhs (left hand side) and rhs (right hand side) and returns the result.

```
operator_add(double lhs, double rhs)
    result = lhs * rhs
    return result
```

- `operator_div(double lhs, double rhs)`: This function takes in two doubles: lhs and rhs. Its calculates the quotient of the lhs (left hand side) and rhs (right hand side) and returns the result.

```
operator_div(double lhs, double rhs)
    result = lhs / rhs
    return result
```

- `bool parse_double(const char *s, double *d)`: This function takes two parameters: a constant character pointer s and a double pointer d. It attempts to parse a double-precision floating point number from the string s. If the string is not a valid number it returns false. Otherwise, it stores the number in the location pointed to by d and returns true. This function is given in the assignment instructions. [1]

```
bool parse_double(const char *s, double *d)
    char *endptr;
    double result = convert string s to double
    if endptr != s then
        *d = result
        return true
    else
        return false
```

Psuedocode

```
main
    unary_operators = my_unary_operators

    while (opt = getopt) != -1
        switch opt
            case h:
                print usage message
                return 0
                break
            case m:
                unary_operators = my_unary_operators
            default:
                print usage message
                return 1
```

```
while true
    x = 0
    input_line[1024]
    error = false

    prompt for input
    if input is ^d then
        break loop
    for i, 0 to length of input line
        if input_line[i] == '\n' then
            intput_line[i] = '\0'

    token = word in input_line
    while token != NULL and error = false
        if parse_double(word, x) is true then
            if stack_push false then
                print error stack full message
                clear stack
                error = true

        else if word is longer than 1 character then
            print error invalid string message
            error = true
        else if binary_operators[word] != NULL or binary_operators[word] != 0 then
            if apply_binary_operator(binary_operators[word]) is false then
                print error invalid binary operation message
                clear stack
                error = true;
        else if unary_operators[word] != NULL or unary_operators[word] != 0 then
            if apply_unary_operator(unary_operators[word]) is false then
                print error message
                clear stack
                error = true
        else
            print error invalid character message
            clear stack
            error = true

        print stack
        clear stack
        if error = false then
            print new line
return 0
```

Error Handling

- Unknown operation: If the program encounters an unknown operation such as 'l', it will print an error message and re-prompt the user for input. It will discard all previous input.
- Invalid command line argument: If the program receives and invalid command line argument, it will print an error message and usage instructions. It will also terminate the program.

- Invalid binary operation: If the program attempts to calculate a binary operation with only a single number, an error message will be printed, input will be cleared, and it will prompt for input again.
- Invalid unary operation: If the program attempts to calculate a unary operation with no number on the stack, it will print an error message, clear the stack, and re-prompt the user for input.
- Exceeding stack capacity: If the stack capacity is exceeded at anytime, the program will print an error message, clear the stack, and re-prompt the user for input.

Testing

The code is ran with valgrind to ensure there are no memory leaks. This is done with commands `make clean` then `make debug` then `valgrind ./calc`. It is also compiled with scan-build to catch any errors. This is done with command `make scan-build`.

Testing Functions

To test the functions, a `test.c` program is utilized. This c program contains asserts and function calls that confirm the output or result of each function is as expected. It contains a range of inputs meant to simulate all possible cases. This is to make sure that there are no unexpected results and that everything functions as intended. Some of the tests are as follows:

```
// testing stack
double a = 0.0;

stack_clear();
assert(stack_peek(&a) == false);
assert(stack_pop(&a) == false);

/*fill in stack*/
for (int i = 1; i <= STACK_CAPACITY; ++i) {
assert(stack_push(1.0 * i) == true);
}

assert(stack_push(65.0) == false);

// testing operators

assert(apply_binary_operator(operator_add));
assert(apply_binary_operator(operator_sub));
assert(apply_binary_operator(operator_mul));
assert(apply_binary_operator(operator_div));

assert(apply_unary_operator(my_unary_operators['s']));
assert(apply_unary_operator(my_unary_operators['c']));
assert(apply_unary_operator(my_unary_operators['t']));
assert(apply_unary_operator(my_unary_operators['r']));

stack_clear();
assert(apply_binary_operator(operator_add) == false);
assert(apply_binary_operator(operator_sub) == false);
assert(apply_binary_operator(operator_mul) == false);
assert(apply_binary_operator(operator_div) == false);

// testing mathlib
```

```
    assert(Abs(-10) == 10);
    assert(Abs(3.4) == 3.4);

    assert(Abs(Sin(2) - sin(2)) < EPSILON);
    assert(Abs(Sin(10*M_PI) - sin(10*M_PI)) < EPSILON);
    assert(Abs(Cos(2) - cos(2)) < EPSILON);
    assert(Abs(Sin(-3*M_PI) - sin(-3*M_PI)) < EPSILON);
    assert(Abs(Tan(5*M_PI) - tan(5*M_PI)) < EPSILON);
```

Testing RPN Framework

To test the overall functionality of `calc.c`, the `diff` command is used to check whether the output of the program matches the calc binary reference. To do this, an input file will be created containing a series of valid and invalid inputs. The input file has inputs to cover every error message in order to confirm the program is running as desired. The errors the inputs cover can be found in the Error Handling section where they are listed. The inputs are as follows:

```
input.txt
   5 5 +
   5 6 -
   1 3 *
   6 2 /
   10 4 %
   1 +
   +
   3 s
   2 c
   11 t
   -5 a
   64 r
   c
   -2 r
   x
   5 sin
   1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
        36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65
   5 0 /
```

This input will be given to the calc binary and the output of stdout and stderr will be printed in two different files. This is done with the command:

```
./calc_ref < input.txt > expected_out.txt 2> expected_err.txt
```

where `calc_ref` is the binary, `expected_out.txt` contains stdout, and `expected_err.txt` contains stderr. The same is done with the programs executable:

```
./calc < input.txt > out.txt 2> err.txt
```

where `calc` is the executable, `out.txt` contains stdout, and `err.txt` contains stderr. Then diff is used as `diff expected_out.txt out.txt` and `diff expected_err.txt err.txt`. This will print the differences between the two files showing what needs to be changed or fixed. This process is shown in Fig. 12

Testing Math

The file `test.c` also contains asserts and function calls to test the math algorithms included in the program. These are listed in the Algorithms section. An additional program is created: `graphs.c`. I learned how to

use file pointers from the class textbook: Section 7.5, pages 142-145. [2] This program prints the x values separated by commas and the y values separated by commas into a file: `graphs.csv`. The y values are the differences between the `math.h` libraries trig functions and the trig functions within `mathlib.c`. This file can then be taken to a spreadsheet and a graph can be made. Fig. 1 - 5 show the graphs derived from `graphs.c`. Fig. 1 shows that the margin of error between the trig functions is much larger for tangent than sine or cosine. This is because the created tangent function is inaccurate at the vertical asymptotes. Even with this, the created tangent function has a maximum deviation of 5.81e−12 from the `math.h` tangent function (Fig. 5). For sine and cosine (Fig. 2 and Fig. 3), the maximum difference is much smaller: 9.90e−15 and 1.2e−14 respectively. Fig. 4 shows that although not exactly the same, the differences between the sine functions, and the differences between the cosine functions are similar. To make the deviation even smaller, one would need to set EPSILON to a smaller value.

```
graphs.c
    file pointer = open file "graphs.csv" in write mode
    print to file "x , sin - Sin, cos - Cos, tan - Tan"
    for i, 0 to 2pi, i + 0.1
        print to file "i, sin(i) - Sin(i), cos(i) - Cos(i), tan(i) - Tan(i)"

    close file
    return 0
```

Results

The program runs as intended. The command line arguments do what they should. Fig. 6 shows all the variations of arguments. The difference between `./calc -m` and `./calc` is not apparent in the figure, but it works as it should. The argument `-x` accounts for any argument that is not an accepted argument. The error is caught and the message is printed correctly. Fig. 7 shows each of the binary operators working correctly and showing the correct error handling for when the binary operators do not work. Fig. 8 shows each of the unary operators working correctly and each type of error handled correctly. Fig. 9 shows the last error to be handled which is that of attempting to push too many numbers to the stack.

The results of my math functions were as expected. All three trig functions were very close to the `math.h` library's trig functions. To see how I arrived at this conclusion read the Testing Math section. If I wanted my sine and cosine functions to be more precise, I would only have to change the value of epsilon in the algorithms. Making the value of epsilon smaller would make my functions more precise. However, for the purpose of this program, the trig functions created work perfectly. This is because the program only prints out 10 decimal places. The sin and cosine functions match the `math.h` library sine and cosine functions when only looking at up to 10 decimal places. The tan function can vary slightly, but it is still very similar to the `math.h` library tangent function. This is shown in Fig. 10 and Fig. 11

References

[1] Dr. Keery Veenestra and TAs. Assignment 2: Calculator. `https://git.ucsc.edu/cse13s/fall-2023-section-01/resources`, Fall 2023.

[2] Dennis Ritchie and Brian Kernighan. *The C Programming Language. 2nd Edition.* Prentice Hall, 1978.

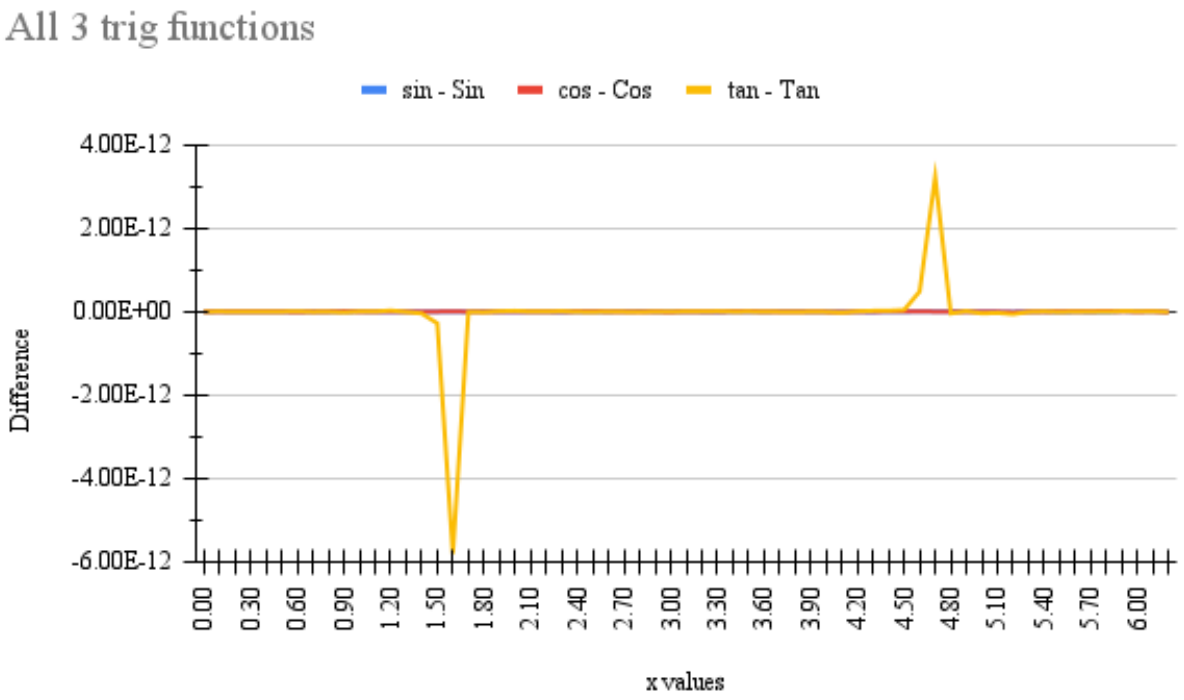

Figure 1: All 3 trig functions overlayed

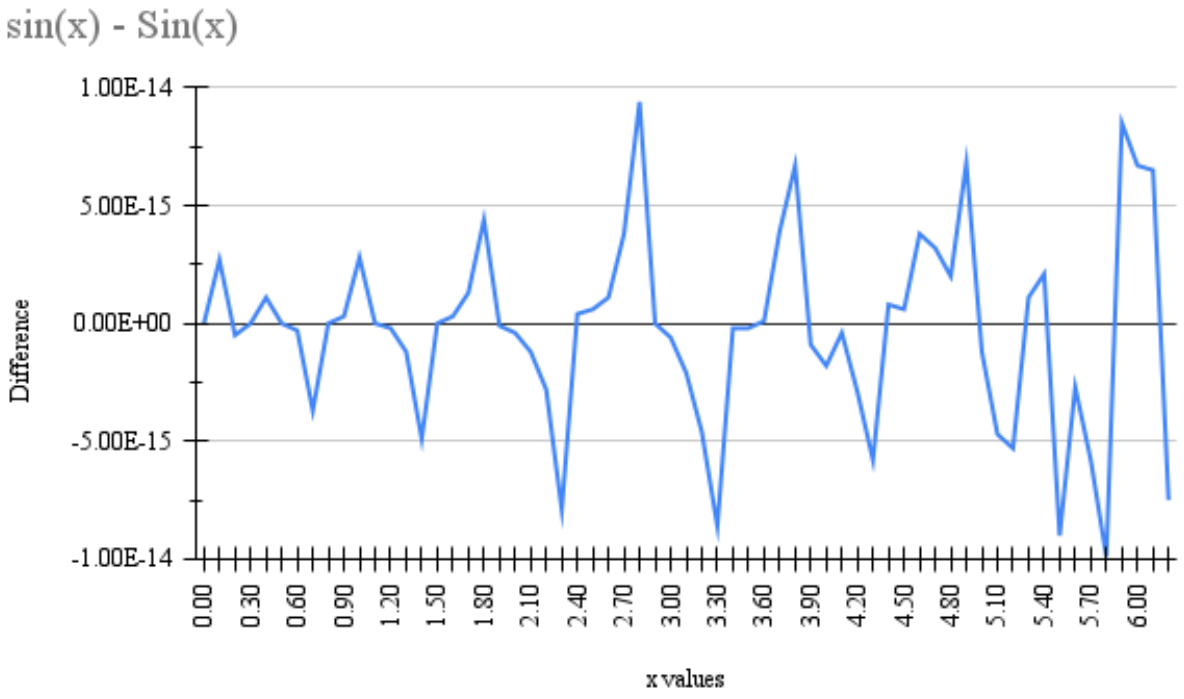

Figure 2: Difference between sin(x) and Sin(x) graph

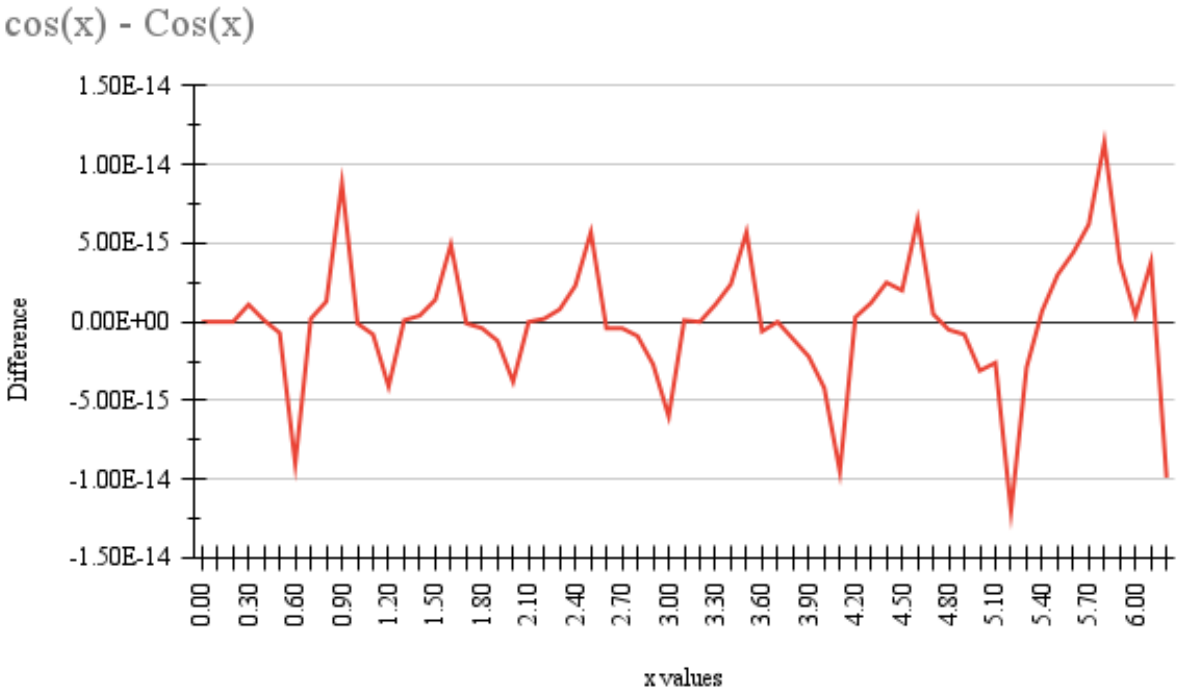

Figure 3: Difference between cos(x) and Cos(x) graph

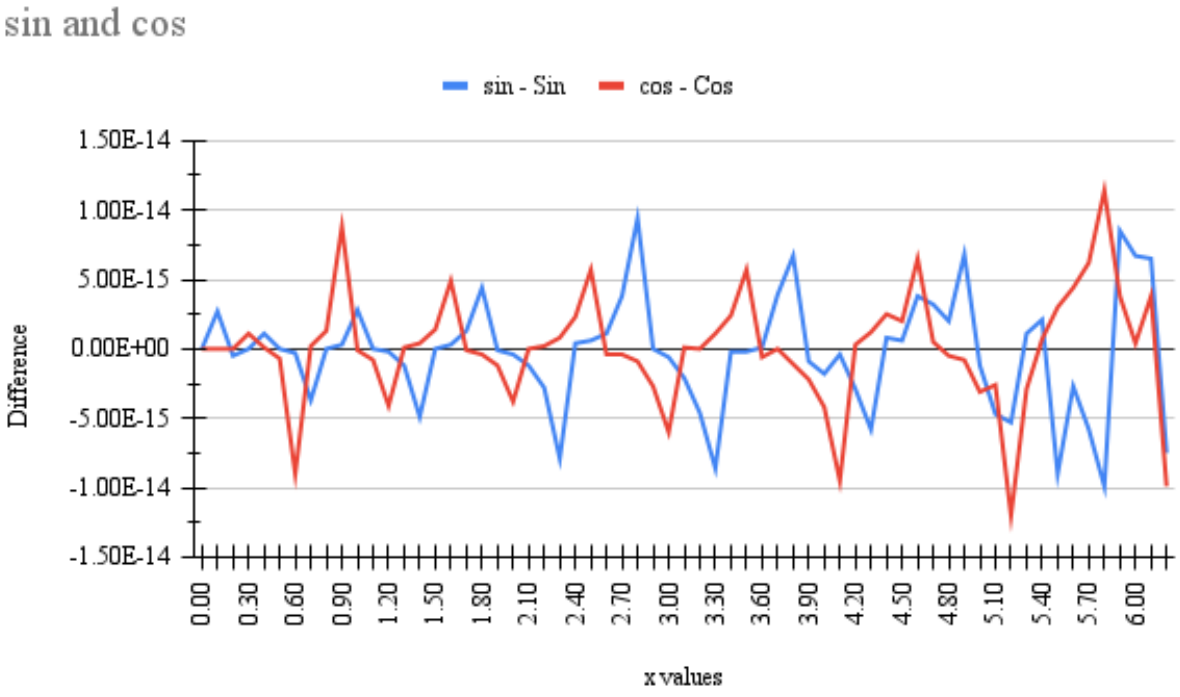

Figure 4: Difference between sin(x) - Sin(x) and cos(x) - Cos(x) graph

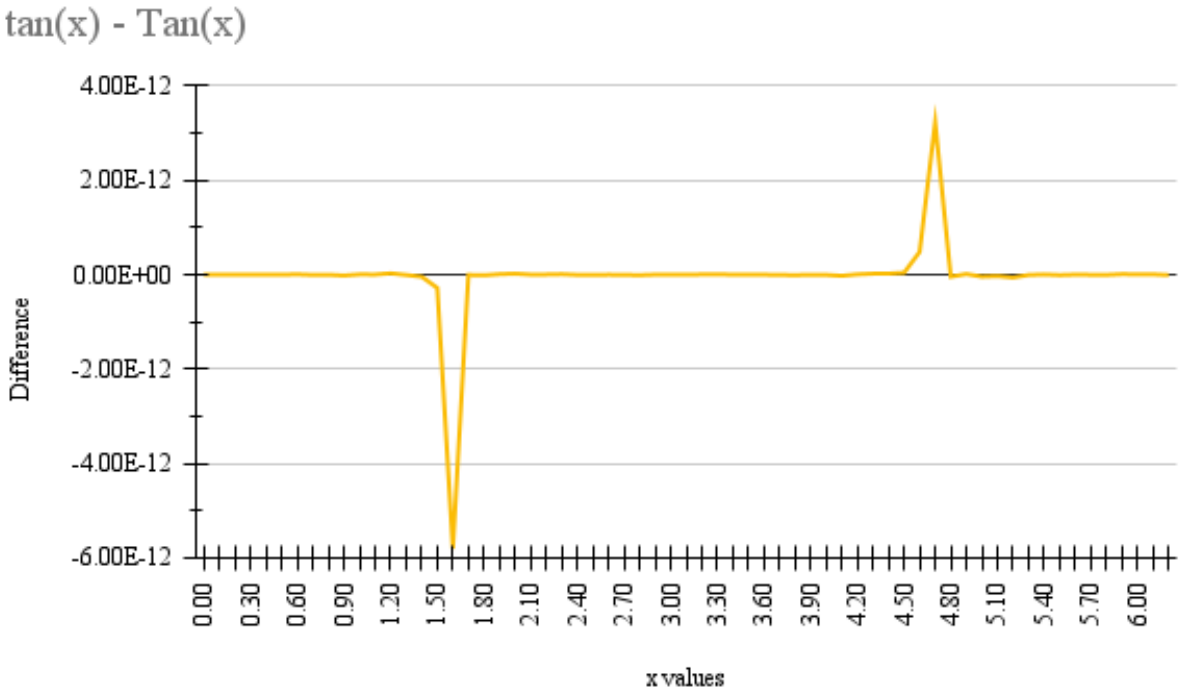

Figure 5: Difference between tan(x) and Tan(x) graph

```
savila35@cse13s-vm:~/cse13s/asgn3$ ./calc -h
usage: ./calc [-mh]
    -m: use libm trig function implementations instead of custom ones.
    -h: show help
savila35@cse13s-vm:~/cse13s/asgn3$ ./calc -hm
usage: ./calc [-mh]
    -m: use libm trig function implementations instead of custom ones.
    -h: show help
savila35@cse13s-vm:~/cse13s/asgn3$ ./calc -m
> 3.14 s
0.0015926529
savila35@cse13s-vm:~/cse13s/asgn3$ ./calc
> 3.14 s
0.0015926529
> savila35@cse13s-vm:~/cse13s/asgn3$ ./calc -x
./calc: invalid option -- 'x'
usage: ./calc [-mh]
    -m: use libm trig function implementations instead of custom ones.
    -h: show help
savila35@cse13s-vm:~/cse13s/asgn3$
```

Figure 6: All command line arguments

```
> savila35@cse13s-vm:~/cse13s/asgn3$ ./calc
> 1 2 +
3.0000000000
> 10 9 -
1.0000000000
> 2 4 *
8.0000000000
> 10 5 /
2.0000000000
> 6 3 %
0.0000000000
> 1 +
error: not enough values on stack for binary operator
> -
error: not enough values on stack for binary operator
> savila35@cse13s-vm:~/cse13s/asgn3$
```

Figure 7: Binary Operators

```
savila35@cse13s-vm:~/cse13s/asgn3$ ./calc
> 1 s
0.8414709848
> 1 c
0.5403023059
>  1 t
1.5574077247
> -5 a
5.0000000000
> 16 r
4.0000000000
> 0 cos
error: unknown operation "cos"
> r
error: not enough values on stack for unary operator
> 5 x
error: unknown operation 'x'
> savila35@cse13s-vm:~/cse13s/asgn3$
```

Figure 8: Unary Operators

```
savila35@cse13s-vm:~/cse13s/asgn3$ ./calc
> 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65
error: insufficient stack space to push 65.0000000000
> savila35@cse13s-vm:~/cse13s/asgn3$
```

Figure 9: Stack is full error

```
savila35@cse13s-vm:~/cse13s/asgn3$ ./calc
> 3 s
0.1411200081
> 3 c
-0.9899924966
> 3 t
-0.1425465431
> savila35@cse13s-vm:~/cse13s/asgn3$ ./calc -m
> 3 s
0.1411200081
> 3 c
-0.9899924966
> 3 t
-0.1425465431
> savila35@cse13s-vm:~/cse13s/asgn3$
```

Figure 10: Trig function comparison

```
savila35@cse13s-vm:~/cse13s/asgn3$ ./calc -m
> 1.57 t
1255.7655915008
> savila35@cse13s-vm:~/cse13s/asgn3$ ./calc
> 1.57 t
1255.7655915061
> savila35@cse13s-vm:~/cse13s/asgn3$
```

Figure 11: Tangent difference

```
savila35@cse13s-vm:~/cse13s/asgn3$ ./calc_ref < input1.txt > expect_out1.txt 2> expect_err1.txt
savila35@cse13s-vm:~/cse13s/asgn3$ ./calc < input1.txt > output1.txt 2> err1.txt
savila35@cse13s-vm:~/cse13s/asgn3$ diff expect_out1.txt output1.txt
savila35@cse13s-vm:~/cse13s/asgn3$ diff expect_err1.txt err1.txt
savila35@cse13s-vm:~/cse13s/asgn3$
```

Figure 12: Testing using diff