Week 6 Programming Assignment

Steffen Petersen — au722120

October 10th 2022

Here is the link for my repository, in which you will find all the edited code files and such.
https://github.com/Aarhus-University-ECE/assignment-7-SirQuacc

1

(Code answer) In the lecture we discussed how we could implement an approximate sine function using the Taylor series equation:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

(a)

Write this function in a C program so that it calculates the sine function with precision up to n Taylor series terms, e.g. the above example shows 4 terms. Your sine function should accept both the x value and n precision as input. The signature of your function should be:

```
double taylor_sine(double x, int n)
{
  /* implement your function here */
}
```

Write your Taylor series implementation of sine as a **library** consisting of: a header file and a source file (with no **main()** function).

Everything can be found in the repository, respectively in the /include/taylor_sine.h file and /src/taylor_sine.c file, but below is also the standalone taylor_sine function.

```c
double taylor_sine(double x, int n)
{
    double sine = 0; //Initialize variable to return
    int swapCount = 0; //Keep track of whether to add or subtract from previous result in the for loop.
    for (int i = 1; swapCount <= n; i+=2){
        if(swapCount%2 == 0){ //even swapCount, when the next bit is added to the previous
            sine += pow(x, i)/factorial(i); //Culcalation based on the taylor sine formula
            swapCount++;
        }
        else{ //uneven swapCount, when the next bit is subtracted from the previous
            sine -= pow(x, i)/factorial(i);
            swapCount++;
        }
```

```
        }
        return sine;
    }
```

(b)

Write some tests for different values of x (try both small and large input values), and compare your function output with the ANSI C sin function. Write your test program (which will have a **main()** function) separately from your library, i.e. you should only include the library header file. Compile your test program by linking with your Taylor Sine library.

This has been done in the /src/main.cpp as the main() function, where it runs through a series of input x values with a series of different precision values.

(c)

Answer the following questions using your test program, and please **provide your answers as comments in your test program**: Which intervals of input x did your function give a similar result to the ANSI C sin function? What impact did increasing the precision have (i.e. increasing the number of Taylor series terms)?

Generally speaking the lower values of x are easier to get right, even with lower precision, and when the x values are higher, it's necessary to have a higher precision, before it becomes a reasonably valid result. That said, with too high x-values, my function at least, also appears to run into overflow problems, meaning results are way out of wack compared to the sin() function from the standard math.h library.
And as expected with too high precision, meaning lots of taylor series terms, it will also run into problems, and the result will output an error.

2

(Code answer) Stacks are containers where items are retrieved according to the order of insertion, independent of content. *Stacks* maintain *last-in, first-out* order (LIFO). The abstract operations on a stack include:

- `Push(x,s)` – Insert item `x` at the top of stack `s`.
- `Pop(s)` – Return (and remove) the top item of stack `s`.
- `Initialize(s)` – Create an empty stack.
- `Full(s), Empty(s)` – Test whether the stack can accept more pushes or pops, respectively. There is a trick to this, see if you can spot it!

Note that there is no element search operation defined on standard stacks.
Defining these abstract operations enables us to build a stack module to use and reuse without knowing the details of the implementation. The easiest implementation uses an array with an index variable to represent the top of the stack. An alternative implementation, using linked lists, is better because it can't overflow.

Stacks naturally model piles of objects, such as dinner plates. After a new plate is washed, it is pushed on the top of the stack. When someone is hungry, a clean plate is popped off the top. A stack is an appropriate data structure for this task since the plates don't care which one is used next. Thus one important application of stacks is whenever order *doesn't* matter, because stacks are particularly simple containers to implement.

(a)

Implement a stack based on singly-linked lists as discussed in the lecture.

The implementation can be seen in the repository, as a combination of the stack.c and stack.h files in the /src and /include directories respectively.

(b)

Test your implementation. You should expect the following "laws" to hold for any implementation of a stack. *Hint:* you should enforce these conditions using *assert* statements:

(A) After executing `Initialize(s);` the stack `s` must be empty.
(B) After executing `Push(x,s); y = Pop(s);` the stack `s` must be the same as before execution of the two commands, and `x` must equal `y`.
(C) After executing `Push(x0,s); Push(x1,s); y0 = Pop(s); y1 = Pop(s);` the stack `s` must be the same as before execution of the two commands, `x0` must equal `y1`, and `x1` must equal `y0`.

Remark: Stack order is important in processing any properly nested structure. This includes parenthesised formulas (push on a "(", pop on ")"), recursive program calls (push on a procedure entry, pop on a procedure exit — we will be discussing *recursion* in Lecture 9), and depth-first traversals of graphs (push on discovering a vertex, pop on leaving it for the last time).

These tests are all present in /tests/src/tests.cpp and have been run with CMake to confirm.