Lab3 Report

Sriraj Vasanta
CS24BTECH11066

September 2025

Design Approach

Helper Functions

The two helper functions below, power and factorial, are used in the main functions.

1. Power

A float register initialised to single precision 1.0 is iteratively multiplied with the value in fa0, until the value in a0 becomes 0, which is decreased by 1 after each iteration.

2. Factorial

A register initialised to 1 is iteratively multiplied with integers from 2 to n.

Main Functions

1. Exponent

$$e^x = \sum_{n=1}^{\infty} \frac{x^{n-1}}{(n-1)!}$$

The n^{th} term uses n-1, for power and factorial. The program computes the sum of the terms from the end and the decreases n, stored in the register a0, until it reaches 0.

2. Sine

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

The result is accumulated from the end i.e. the n^{th} term, until the value stored in the register a0 is zero. Each term is added/subtracted based on whether the term is odd/even, done using branch instructions.

3. Cosine

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

The only difference between the expansion of sine and cosine is the ending value, which are 2n-1 and 2n-2 respectively.
The logic is same for both sine and cosine, except for the above mentioned difference.

4. Natural logarithm

$$\ln(1+x) = \sum_{n=1}^{\infty} (-1)^{n+1} \frac{x^n}{n!}$$

This is can also be written as:

$$-ln(1+x) = \sum_{n=1}^{\infty} \frac{(-x)^n}{n!}$$

This slight manipulation is done to avoid writing branches for addition/subtraction of terms like in sine and cosine. The additive inverse of the above result is returned at the end.
The float argument, present in the register fa0, is changed as $fa0 = fa0 - 1$ to accomodate for the above expansion.
Single precision NaN is returned if the float argument is less than or equal to 0.

5. Inverse

$$\frac{1}{x} = \frac{1}{1-(1-x)} = \sum_{n=1}^{\infty} (1-x)^{n-1}, \quad if \quad |1-x| < 1$$

The logic for this is self-explanatory from the above expression.
Single precision NaN is returned if the float argument is ≤ 0.0 or ≥ 2.0.

If the number of functions to be computed is given as ≤ 0 or the number of terms to be calculated in any function is given as ≤ 0, then single-precision NaN is returned.

Verification approach

The values from a python code for the same question were compared with the values returned by the assembly code.
The code is present at the end of the document for reference.

Issues Encountered

- Slightly different values from the ones given the problem statement's example were obtained. This was probably due to the unusual approach of the implementation, where accumulation of terms is from the n^{th} term and not from the start.
- A lot of registers were being overwritten due to nested function calls. This was later resolved by appropriate caller/callee saves.

Code for Verification

```python
import numpy as np
nan32 = np.float32('nan')

# a!
def factorial(a: int) -> int:
    result = 1
    while a > 0:
        result *= a
        a -= 1
    return result

# a^n
def power(a: float, n: int) -> np.float32:
    a = np.float32(a)
    if n == 0:
        return np.float32(1)
    if a == np.float32(0):
        return np.float32(0)
    result = np.float32(1)
    while n > 0:
        result *= a
        n -= 1
    return np.float32(result)

# exp(a, n)
def exp(a: float, n: int) -> np.float32:
    a = np.float32(a)
    result = np.float32(0)
    while(n > 0):
        num = power(a, n-1)
        den = factorial(n-1)
        result += np.float32(num / den)
        n -= 1
    return np.float32(result)

# sin(a, n)
def sin(a:float, n:int) -> np.float32:
    a = np.float32(a)
    result = np.float32(0)
    while(n > 0):
        temp = 2*n-1
        num = power(a, temp)
        den = factorial(temp)
        # Add to the result if odd term
        if(n%2):
            result += np.float32(num/den)
        # Subtract from the result if even term
```

```
        else:
            result -= np.float32(num/den)
        n -= 1
    return np.float32(result)

# cos(a, n)
def cos(a:float, n:int) -> np.float32:
    a = np.float32(a)
    result = np.float32(0)
    while(n > 0):
        temp = 2*n-2
        num = power(a, temp)
        den = factorial(temp)
        # Add to the result if odd term
        if(n%2):
            result += np.float32(num/den)
        # Subtract from the result if even term
        else:
            result -= np.float32(num/den)
        n -= 1
    return np.float32(result)

# ln(a, n)
def ln(a:float, n:int) -> np.float32:
    if(a <= 0):
        return nan32

    a = a-1
    a = -1*a
    a = np.float32(a)
    result = np.float32(0)
    while(n > 0):
        num = power(a, n)
        den = n
        result += np.float32(num/den)
        n -= 1
    return np.float32(-1*result)

# inv(a, n)
def inv(a:float, n:int) -> np.float32:
    if(a<=0 or a>=2):
        return nan32

    a = 1-a
    a = np.float32(a)
    result = np.float32(0)
    while(n > 0):
        result += np.float32(power(a, n-1))
        n -= 1
    return np.float32(result)
```

```
98
99
100
101 # Example usage
102 a = float(input())
103 n = int(input())
104 print(ln(a, n))
```