Next Silicon: CM Home Assignment

Djordje Zivanovic
LinScale

April 26, 2025

Contents

1 Introduction

This report [1] contains the responses to the tasks stated in the home project pdf. It is accompanied with the repository [2] that contains reproducible solutions with the installation instructions alongside: experiments and tests according to the task requirements.

2 The Existing Implementation: Code Analysis and Documentation

In this section we analyze the existing code shown in Algorithm 1.

2.1 Code Drawbacks

The code is written for C and follows the following bad practices:

1. Reusing the variable multiple times, `(float)M_PI` and `2.0f * (float)M_PI`, (lines 3, 4, 6, 6, 5, 7 of Algorithm 1);

2. Not using auto in order to automatically deduce types since the results of all the statements are known;

3. Cleaning if loop to be more understandable: `fmodf` returns the result in the range $(-2\pi, 2\pi)$. Then, now it is obvious that one checks whether the number is outside of the range $[-\pi, \pi]$, and then updates `x` for 2π period, so the further code operates with the number in the range $[-\pi, \pi]$;

4. Adding more verbosity ();

5. Reusing variable names -> more verbose names should be used in order to improve readability of the code. The c ompiler will optimize for the least number of variables/registers to be used;

6. Renaming function names and migrating these functions to the corresponding headers and sources that would contain the custom maths functions.

2.2 Numerical and Implementation Drawbacks

Here, I will give state several main drawbacks in terms of the implementation and numerical accuracy. The division by In the next subsection, I will list the drawbacks related to the method itself.

[1] report
[2] next-silicon-maths

1: **Input:** A float (IEEE-754) number
2: **Output:** A float (IEEE-754) sine value of this number computed using Taylor Series.
3: **Steps:**

```
float fp32_custom_sine(float x)
{
    x = fmodf(x, 2.0f * (float)M_PI);
    if (x > (float)M_PI)
        x -= 2.0f * (float)M_PI;
    else if (x < -(float)M_PI)
        x += 2.0f * (float)M_PI;
    float result = 0.0f;
    float term = x;
    float x_squared = x * x;
    int sign = 1;
    for (int n = 1; n <= 7; n += 2)
    {
        result += sign * term;
        sign = -sign;
        term = term * x_squared;
        term = term / (float)(n + 1);
        term = term / (float)(n + 2);
    }
    return result;
}
```

Algorithm 1: Algorithm with Code Listing

2.3 Mathematical Analysis

Let us state the general Taylor series formula that is implemented in Algorithm 1.

Theorem 2.1 (Theorem 5.19 from [1, p. 113]): Let f be a function having finite n-th derivative $f^{(n)}$ everywhere in an open interval (a, b) and assume that $f^{(n-1)}$ is continuous on the closed interval $[a, b]$. Then, for every x in $[a, b], x \neq c$, there exists a point x_1 interior to the interval joining x and c such that

$$f(x) = f(c) + \sum_{k=1}^{n-1} \frac{f^{(k)}(c)(x-c)^{(k)}}{k!} + \frac{f^{(n)}(x_1)}{n!}(x-c)^n.$$

A corollary of Theorem 2.1 when we set $c = 0$ is a Maclaurin Series.

Corollary 2.2 (Maclaurin Series): Let f be a function having finite n-th derivative $f^{(n)}$ everywhere in an open interval (a, b) and assume that $f^{(n-1)}$ is continuous on the closed interval $[a, b]$. Assume that $c \in [a, b]$. Then, for every x in $[a, b], x \neq 0$, there exists a point x_1 interior to the interval joining x and 0 such that

$$f(x) = f(0) + \sum_{k=1}^{n-1} \frac{f^{(k)}(0)(x)^{(k)}}{k!} + \frac{f^{(n)}(x_1)}{n!}x^n.$$

Let us now set $a = -\pi, b = \pi$ and $f(x) = sin(x)$. The Maclaurin Series becomes for sin function:

Corollary 2.3: For $x \in \mathbb{R}, x \neq 0$ and $n \in \mathbb{N}$, and $0 < |x_1| < |x|$ the approximation of a degree n is

$$sin(x) = \sum_{i=0}^{\lfloor \frac{n}{2} \rfloor} (-1)^i \frac{x^{2i+1}}{(2i+1)!} + L(x, n).$$

where

$$L(x,n) = \begin{cases} \frac{f^{(n+2)}x_1}{(n+2)!}x^{n+2}, & \text{if } n \bmod 2 = 0, \\ \frac{f^{(n+1)}x_1}{(n+1)!}x^{n+1}, & \text{if } n \bmod 2 = 1. \end{cases}$$

Thus, the method stated in Algorithm 1 has an algorithmic error $L(x, 7)$ which is smaller than $\frac{x^8}{8!}$. Now, we obtain

2.4 Accuracy and Correctness Failures

Based on Corollary 2.3, we can notice that Algorithm 1 is incorrect for $x = 0$. Similar, the farther the x is from 0, the error is larger since x^8 grows exponentially. There are several ways to solve these problems:

2.5 Experiments

The test plan will cover the following experiments:

1. Different number distributions
   (a) Equally distanced numbers between certain multiplicands of $\frac{\pi}{2}$; We will vary the distance and provide statistical analysis with plots for each of such distance and multi‘plicand.
   (b) Normally distributed numbers around certain multiplicands of $\frac{\pi}{2}$; We will vary the variance and provide statistical analysis with plots for each of the multiplicands and variances.
2. Edge cases
   (a) the multiplicands of $\frac{\pi}{2}$;
   (b) large numbers (close to the absolute minimum and maximum for the float numbers);
   (c) normal distributions around the multiplicands of $\frac{\pi}{2}$;
   (d) `nans`.
3. Numbers

For each of these experiments we would compute the relative error in comparison to the value computed by the $\sin x$ provided in `cmath`.

3 Additional Algorithms

For the problems stated in Subsection 2.4 there are several ways to approach:

1. Have another Taylor series expansion for numbers around π and $-\pi$. In this case we would manually calculate the $\sin x$ for π and $-\pi$.
2. Add more terms in Taylor series expansion. It is important to find the optimal degree for the balance of accuracy and performance.
3. Implement one or more of the alternative methods: Minimax Polynomial Approximation, Chebyshev Polynomial Expansion [2], Lookup Table with Linear Interpolation and Lookup Table with Spline or Cubic Interpolation.

Here we will focus on the implementation of the Chebyshev Polynomial Expansion.

Definition 3.1 ([2, p .233]): For a number $n \in \mathbb{N}$ and a real number x, the Chebyshev polynomial of degree n is denoted $T_n(x)$ and defined as

$$T_n(x) = \cos(n \arccos x).$$

Theorem 3.2: For $n \in \mathbb{N}$,and $x \in \mathbb{R}$ the Chebyshev polynomial of degree n satisfies the formula:

$$T_n = 2xT_n(x) - T_{n-1}(x),$$

where $T_0(x) = 1$ and $T_1(x) = x$.

Let us introduce definitions that would help with our computations.

Definition 3.3 ([2, p .234]): For a number $j, N \in \mathbb{N}$, the Chebyshev coefficient of degree n is denoted $c_n(x)$ and defined as

$$c_j(N) = \frac{2}{N} \sum_{k=0}^{N-1} f\left(\cos\left(\frac{\pi(k+\frac{1}{2})}{N}\right)\right) \cos\left(\frac{\pi j(k+\frac{1}{2})}{N}\right).$$

Now the following formula holds:

Theorem 3.4: Let $N \in \mathbb{N}$, $x \in \mathbb{N}$, and a function $f : [-1, 1] \mapsto \mathbb{R}$. Let f be a continuous and bounded function. Now, the follows holds:

$$f(x) \approx \sum_{k=0}^{N-1} (c_k(N) \cdot T_k(x)) - \frac{1}{2} c_0(N).$$

We implement the approximation from Theorem 3.4 in Algorithm 3. Chebyshev coefficients are computed in Algorithm 2. Since Chebyshev coefficients from Definition 3.3 are defined for the input $[-1, 1]$, but our input is in the range $[-\pi, \pi]$, we need to map the input to $[-1, 1]$ by uniformly scaling the input to this range (Algorithm 3 line 18). However, we have to scale back the input to $[-\pi, \pi]$ for the function `f` in Algorithm 2 (lines 4, 5, 14). The rest of Algorithm 2 follows Definition 3.3. Algorithm 3 has two more improvements. First, the algorithm uses the optimized `fmodf` for the accuracy implemented in the function `optimizedFModf2Pi` that maps the input from the set of real numbers to the set $[-2\pi, 2\pi]$. Further, instead of following Theorem 3.4, the algorithm implements Clenshaw's formula [2, p .237] for the better stability.

References

[1] Tom M Apostol. Mathematical analysis. Narosa Publishing House Pvt. Ltd., 1985.

[2] William H Press. *Numerical recipes 3rd edition: The art of scientific computing.* Cambridge university press, 2007.

Input: A function `f`, the number of coefficients to compute `numCoeffs` and bounds of the interval `a` and `b`.

Output: A return vector that contains the computed values.

```cpp
std::vector<float> computeChebyshevCoefficients(std::function<float(float)> f, uint32_t numCoeffs, float a, float b) {
    std::vector<float> vCoeffs(numCoeffs, 0.f);

    float bma = 0.5f * (b - a);
    float bpa = 0.5f * (b + a);
    for (uint32_t j = 0u; j < numCoeffs; j++) {
        float sum = 0.f;

        for (uint32_t k = 0u; k < numCoeffs; k++) {
            float leftTheta = std::numbers::pi_v<float> * (k + 0.5f) / numCoeffs;
            float rightTheta = leftTheta * j;
            float leftCos = std::cos(leftTheta);
            float rightCos = std::cos(rightTheta);
            sum += f(leftCos * bma + bpa) * rightCos;
        }
        vCoeffs[j] = sum * 2.0f / numCoeffs;
    }

    return vCoeffs;
}
```

Algorithm 2: Computing Chebyshev Coefficients

1: **Input:** A float (IEEE-754) number and the degree `chebDegreeN` used in Chebyshev approximation.
2: **Output:** A float (IEEE-754) sine value of this number computed using Chebyshev approximation.

```
float nextSiliconSineFP32Chebyshev(float x, uint32_t
    chebDegreeN)
{
    static constexpr auto PI_F = pi_v<float>;
    static constexpr auto TWO_PI_F = 2 * PI_F;

    float xPiRange = x;
    if (std::abs(xPiRange) >= TWO_PI_F)
    {
        xPiRange = optimizedFModf2Pi(xPiRange);
    }
    if (std::abs(xPiRange) > PI_F)
    {
        xPiRange -=  boost::math::sign(xPiRange) *
            TWO_PI_F;
    }

    auto b = PI_F;
    auto a = -PI_F;
    auto y = (2.0f * xPiRange - a - b) / (b - a);
    auto y2 = 2.f * y;
    auto chebCoeffs = computeChebyshevCoefficients(::sinf,
        chebDegreeN, a, b);

    float dMPlusTwo = 0.f;
    float dMPlusOne = 0.f;+

    // Clenshaw's formula
    for (std::size_t k = chebDegreeN - 1; k > 0; k--) {
        float bCurr = y2 * dMPlusOne - dMPlusTwo +
            chebCoeffs[k];
        dMPlusTwo = dMPlusOne;
        dMPlusOne = bCurr;
    }

    return  y * dMPlusOne - dMPlusTwo +  chebCoeffs[0] *
        0.5;
}
```

Algorithm 3: Sine using Taylor series: Existing Method