

Next Silicon: CM Home Assignment

Djordje Zivanovic

LinScale

April 26, 2025

Contents

1	Introduction	2
2	The Existing Implementation: Code Analysis and Documentation	2
2.1	Code Drawbacks	2
2.2	Numerical and Implementation Drawbacks	2
2.3	Mathematical Analysis	4
2.4	Accuracy and Correctness Failures	5
2.5	Experiments	5
3	Additional Algorithms	6
3.1	Preliminaries and Implementation	6
3.2	Experiments: Accuracy and Performance	10

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 and readability improvements:

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 with names and code structure;
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.

1. Reducing number of `2.0f * M_PI` computations;
2. Expressing `2.0f * M_PI` as `2 * M_PI` -> this should be expressed as a sum of two `M_PIs`

¹report

²next-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:**

```
1 float fp32_custom_sine(float x)
2 {
3     x = fmodf(x, 2.0f * (float)M_PI);
4     if (x > (float)M_PI)
5         x -= 2.0f * (float)M_PI;
6     else if (x < -(float)M_PI)
7         x += 2.0f * (float)M_PI;
8     float result = 0.0f;
9     float term = x;
10    float x_squared = x * x;
11    int sign = 1;
12    for (int n = 1; n <= 7; n += 2)
13    {
14        result += sign * term;
15        sign = -sign;
16        term = term * x_squared;
17        term = term / (float)(n + 1);
18        term = term / (float)(n + 2);
19    }
20    return result;
21 }
```

Algorithm 1: Algorithm with Code Listing

3. Transforming `term = term / (n + 1)` and then `term = term / (n+2)` to `term = term / ((n+1) * (n+2))`. Instead of two float divisions and two stores, these are two integer additions (tend to be faster) and only one float division and one store.
4. Unrolling the loop by starting from `n = 3`, we know the first element will always be `x`. We can thus skip the computations in the loop for `n = 1`. We also, do not need to compute the next term in advance, only for the current `n`. Thus we are saving the computations after result updated (sign change, and term updating).
5. we also can accumulate the divisions by integers in a variable.
6. sign multiplication by the term, can be replaced by if else in order to avoid on FLOP for `sign * term`.

In the next subsection, I will list the drawbacks related to the method itself.

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 $\frac{x^8}{8!}$. However, for large numbers this can be quite huge.

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 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 multiplicand.
 - (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 (larger than ULP);
 - (c) normal distributions around the multiplicands of $\frac{\pi}{2}$;
 - (d) `nans`.

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

Here we list the additional algorithms we have chosen.

3.1 Preliminaries and Implementation

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.

In Algorithm 2, all the downsides listed in the previous section with regards to the provided implementation have been fixed. In particular, we added more terms and optimized the existing code.

. 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:

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

```

1      float nextSiliconSineFP32Taylor(float x, uint32_t
      degreeEnd)
2      {
3          static constexpr auto PI_F = pi_v<float>;
4          static constexpr auto TWO_PI_F = 2 * PI_F;
5          static constexpr auto TAYLOR_DEGREE_START = 3u;
6          static constexpr auto TAYLOR_DEGREE_NEXT_INC = 2u;
7
8          if (std::abs(x) == 0.f) { return 0; }
9
10         auto xPiRange = fmodf(x, TWO_PI_F);
11         if (std::abs(xPiRange) > PI_F)
12         { xPiRange -= boost::math::sign(xPiRange) * TWO_PI_F; }
13
14         auto result = xPiRange;
15         auto term = xPiRange;
16         const auto xPiRangeSquared = xPiRange * xPiRange;
17         auto sign = 1;
18         std::vector<float> vFact(degreeEnd + 1, 0);
19         vFact[1] = 1;
20
21         for (auto taylorDegree = TAYLOR_DEGREE_START;
22              taylorDegree <= degreeEnd;
23              taylorDegree += TAYLOR_DEGREE_NEXT_INC)
24         { vFact[taylorDegree] = vFact[taylorDegree - 2] * (
25           taylorDegree - 1) * taylorDegree; }
26
27         for (auto taylorDegree = TAYLOR_DEGREE_START;
28              taylorDegree <= degreeEnd;
29              taylorDegree += TAYLOR_DEGREE_NEXT_INC)
30         {
31             sign = -sign;
32             term = term * xPiRangeSquared;
33             auto termAdd = term / float(vFact[taylorDegree]);
34             if (sign == 1)
35             {
36                 result += termAdd;
37             }
38             else
39             {
40                 result -= termAdd;
41             }
42         }
43         return result;
44     }

```

Algorithm 2: Sine using Taylor series: Optimized Method

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 4. Chebyshev coefficients are computed in Algorithm 3. 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 4 line 18). However, we have to scale back the input to $[-\pi, \pi]$ for the function f in Algorithm 3 (lines 4, 5, 14). The rest of Algorithm 3 follows Definition 3.3. Algorithm 4 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.

- 1: **Input:** A function f , the number of coefficients to compute `numCoeffs` and bounds of the interval a and b .
- 2: **Output:** A return vector that contains the computed values.

```

1      std::vector<float> computeChebyshevCoefficients(std::
      function<float(float)> f, uint32_t numCoeffs, float a,
      float b) {
2          std::vector<float> vCoeffs(numCoeffs, 0.f);
3
4          float bma = 0.5f * (b - a);
5          float bpa = 0.5f * (b + a);
6          for (uint32_t j = 0u; j < numCoeffs; j++) {
7              float sum = 0.f;
8
9              for (uint32_t k = 0u; k < numCoeffs; k++) {
10                 float leftTheta = std::numbers::pi_v<float> *
                    (k + 0.5f) / numCoeffs;
11                 float rightTheta = leftTheta * j;
12                 float leftCos = std::cos(leftTheta);
13                 float rightCos = std::cos(rightTheta);
14                 sum += f(leftCos * bma + bpa) * rightCos;
15             }
16             vCoeffs[j] = sum * 2.0f / numCoeffs;
17         }
18
19         return vCoeffs;
20     }

```

Algorithm 3: 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.

```

1      float nextSiliconSineFP32Chebyshev(float x, uint32_t
2      chebDegreeN)
3      {
4          static constexpr auto PI_F = pi_v<float>;
5          static constexpr auto TWO_PI_F = 2 * PI_F;
6
7          float xPiRange = x;
8          if (std::abs(xPiRange) >= TWO_PI_F)
9          {
10             xPiRange = optimizedFModf2Pi(xPiRange);
11          }
12          if (std::abs(xPiRange) > PI_F)
13          {
14             xPiRange -= boost::math::sign(xPiRange) *
15                         TWO_PI_F;
16          }
17
18          auto b = PI_F;
19          auto a = -PI_F;
20          auto y = (2.0f * xPiRange - a - b) / (b - a);
21          auto y2 = 2.f * y;
22          auto chebCoeffs = computeChebyshevCoefficients(::sinf,
23                  chebDegreeN, a, b);
24
25          float dMPlusTwo = 0.f;
26          float dMPlusOne = 0.f;
27
28          // Clenshaw's formula
29          for (std::size_t k = chebDegreeN - 1; k > 0; k--) {
30              float bCurr = y2 * dMPlusOne - dMPlusTwo +
31                          chebCoeffs[k];
32              dMPlusTwo = dMPlusOne;
33              dMPlusOne = bCurr;
34          }
35
36          return y * dMPlusOne - dMPlusTwo + chebCoeffs[0] *
37                  0.5;
38      }

```

Algorithm 4: Sine using Chebyshev Approximation

3.2 Experiments: Accuracy and Performance

Here we compare the performance and accuracy of various methods. **Accuracy.** Based on Figure 1 and 2, we can notice that Chebyshev tend to produce more accurate results, especially farther away from 0. We also notice the more terms we use, the more accurate both methods become. The difference in terms of accuracy between optimized and non-optimized Taylor series implementation is negligible. We also run experiments for large numbers: $1e14$ - $1e15$, however `sin x` provided by `std` library produces incorrect results which we verify by comparing some of our values with `np.sin`.

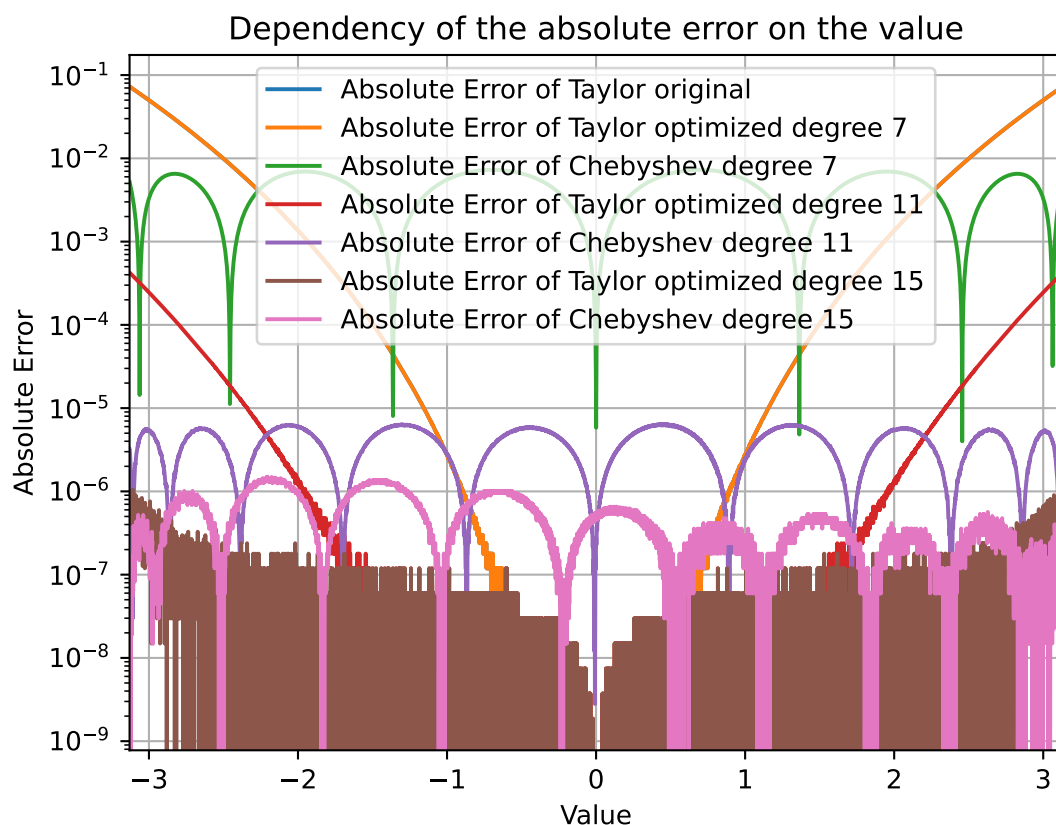


Figure 1: Comparison of absolute error for various methods.

Performance. Based on Figure 3 and 5, we can notice that Chebyshev approximation tends to be slower around one order of magnitude. Similar, as we add more terms, the performance decreases, especially for the Chebyshev approximation. We expected this since the time complexity is higher for the Chebyshev approximation. In Figure 4, we also notice that the optimized Taylor series is a little bit faster than the original one. Furthermore, the performance does not decrease that significantly. Thus, we should use Taylor series with 11 terms in order to balance accuracy and performance. For Chebyshev approximation, 11 terms provide sufficient accuracy (less than $1e-5$).

References

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

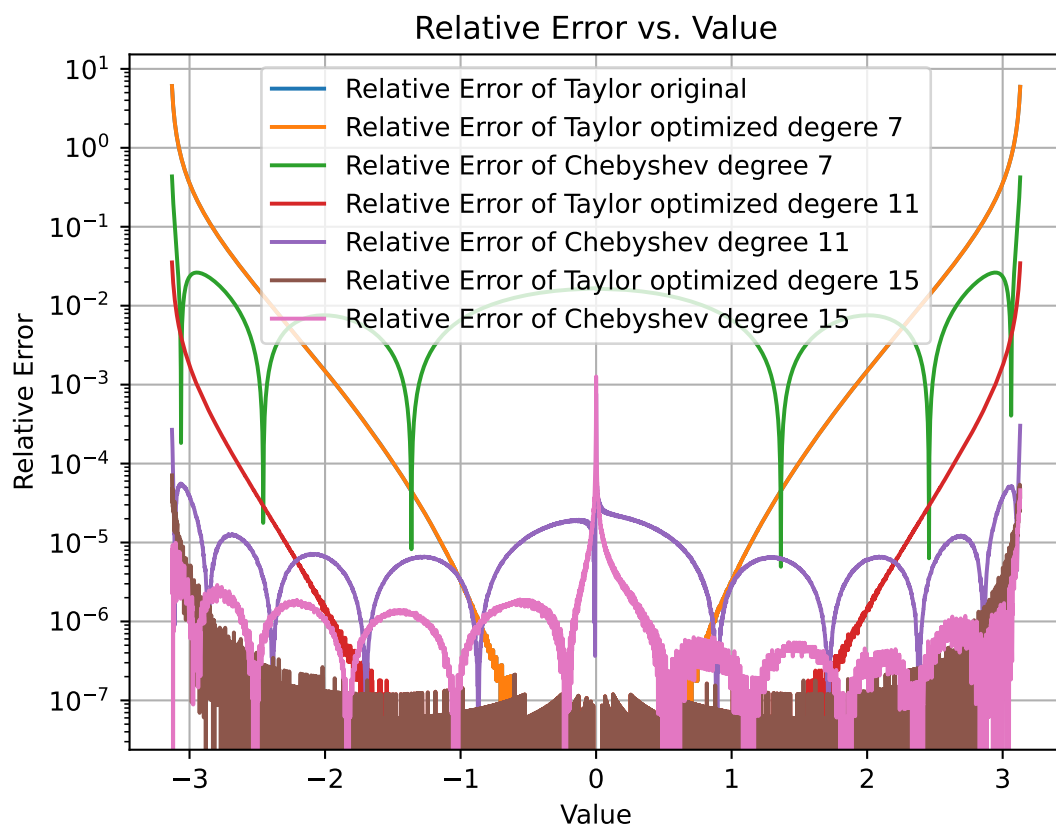


Figure 2: Comparison of relative error for various methods. We compute relative error by comparing to `sin x` provided by `std` library.

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

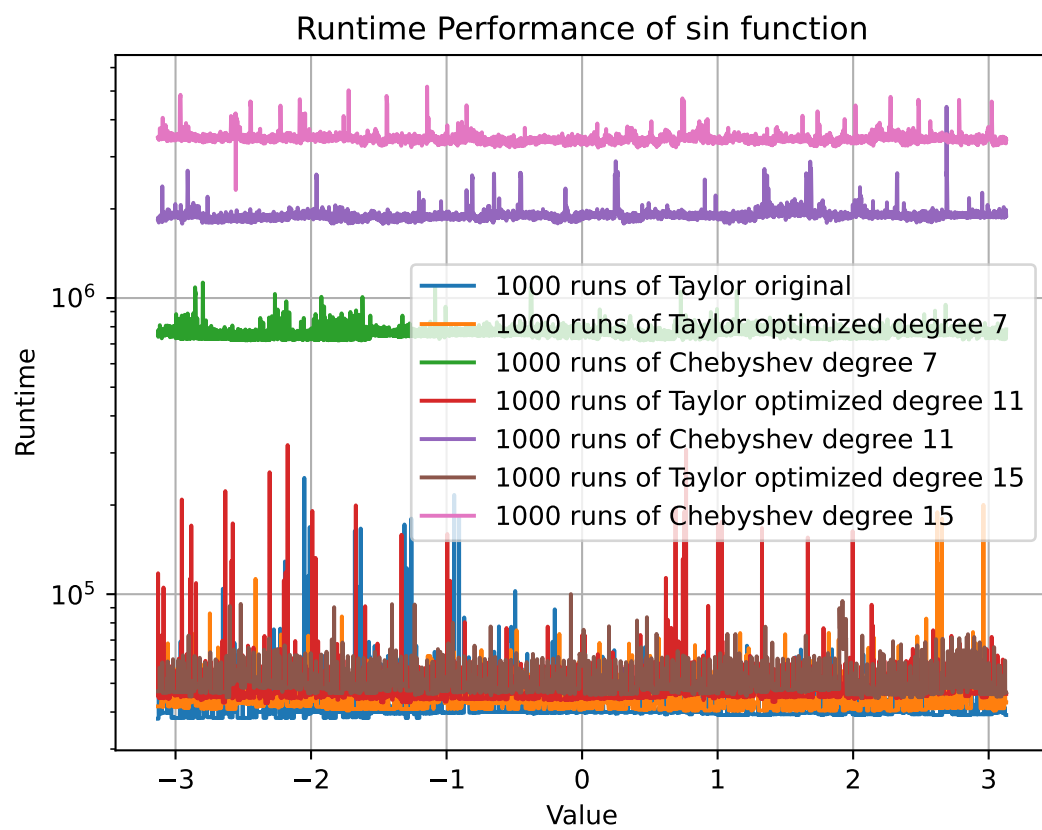


Figure 3: Performance Comparison of Taylor series vs Chebyshev Approximation for 7, 11 and 15 terms.

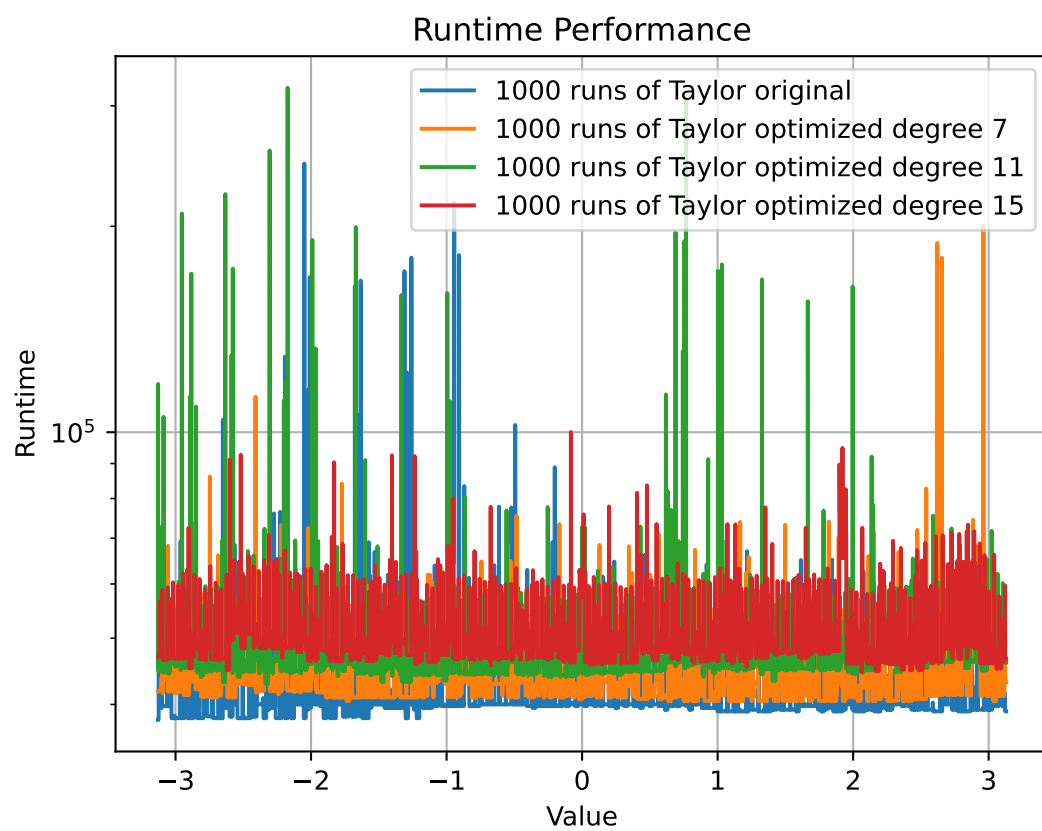


Figure 4: Performance Comparison of Taylor series for 7, 11 and 15 terms.

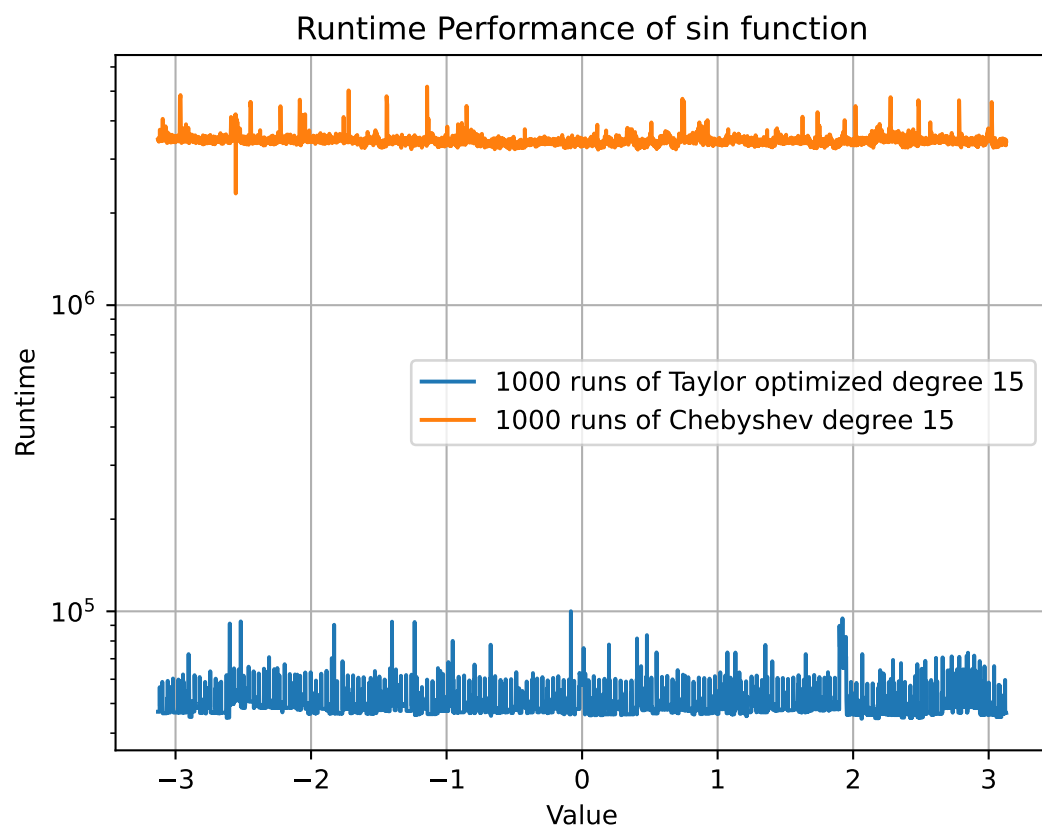


Figure 5: Performance Comparison of Taylor series vs Chebyshev Approximation for 15 terms.