Universiteit
Leiden

Profiling and optimizing the DP3 radio astronomy pipeline

Sam van Hoijtema (s3255522)

Supervisors:
Prof. dr. Rob van Nieuwpoort & Dr. Chris Broekema

Bachelor Thesis– Informatica

Leiden Institute of Advanced Computer Science (LIACS)
liacs.leidenuniv.nl

July 20, 2026

Abstract

Modern radio astronomy happens as much in software in data processing clusters as it does in physical telescopes. These software telescopes consume large amounts of time and energy processing their data, making it desirable for this software to be as fast and efficient as possible. Optimizing the software used for this processing is an ever ongoing process, to which this thesis contributes.

In this thesis we profiled the DP3 radio astronomy pipeline to find bottlenecks, which we then solved using SIMD instructions. We did this with a new sine, cosine, and exponential functions using the AVX2 family of instruction set extensions. Our functions have an acceptable precision of 14 significant decimal digits, compared to the ideal 15 to 17 significant decimal digits required for <1 ULP precision.

We found our custom functions to be between 15% and 76% faster compared to the fastest alternative we tested. Compared to the standard LibC implementation we achieve a speedup of up to 926% when compiled with `-O3`. After we implemented our functions into DP3, the pipeline component achieved a speedup of 44.4% compared to the unmodified version.

1 Introduction

Since its inception, radio astronomy has become increasingly reliant on large scale computing. Telescopes went from independent radio dishes, to networks of dishes joined together with interferometry which rely on computers to achieve their full resolution. Even more modern telescopes, such as the Low-Frequency Array (LOFAR)[1] or the upcoming Square Kilometre Array (SKA)[2], rely so much on their processing pipelines that they are considered software telescopes. The large amount of data collected by these telescopes requires an equally large amount of time, energy, and hardware to process into a useful form. Because science projects like these have a limited budget, both in time and money, it is desirable to have processing pipelines be as fast as possible.

In this thesis we investigate the Default Processing Pipeline (DP3), a component for radio astronomy pipelines developed by ASTRON[3], with the final goal of optimizing its execution time. The process of optimization is split in several stages. First, the codebase has to be profiled to find functions that take up a disproportionate amount of time, which can be targeted for optimization. One or more methods of optimization then have to be chosen and implemented. Finally, the newly optimized code has to be compared to the old version to confirm it is faster.

After profiling we decided to focus on Single Instruction Multiple Data (SIMD) as our optimization strategy. Specifically, created custom SIMD implementations of sine, cosine, and the exponential function using AVX2 intrinsics in C++. We then compare our custom functions to several existing implementations, and implement the fastest in DP3.

Figure 1: The LOFAR Superterp stations in Exeloo, Netherlands[4].

1.1 Background

LOFAR, beamforming, and DP3

The Low-Frequency Array (LOFAR) is a radio telescope consisting of many simple omni-directional antennas sensitive to radio waves between 10 MHz to 240 MHz[1]. In total the telescope consists of approximately 20,000 antennas spread across Europe. The project is led by ASTRON in the Netherlands. By treating the output of the antennas as phased array feeds the antennas can be pointed as a group to observe a specific part of the sky, instead of receiving signals from the entire sky at once[5]. This beamforming process happens in software as a part of a large processing pipeline, split between low-end hardware embedded in the antennas as well as a more powerful central compute cluster. Besides beamforming the pipeline is also tasked with other methods of removing unwanted noise from the signal, such as FM radio.

An important step of the beamforming process is done by the DP3 pipeline[3]. DP3 was initially developed by ASTRON for use with LOFAR specifically, but has since been used for other radio telescopes as well. A project is currently ongoing to optimize DP3 one component at a time to decrease execution time and electricity usage. Initially we intended to look at EveryBeam[6], a dependency of DP3, as this was flagged by field experts as a likely bottleneck. However, as can be seen in section 2.1, the EveryBeam library only takes up a few percent of the total execution time. Instead we further investigate the Predict stage in DP3, which is where all calls to EveryBeam originate.

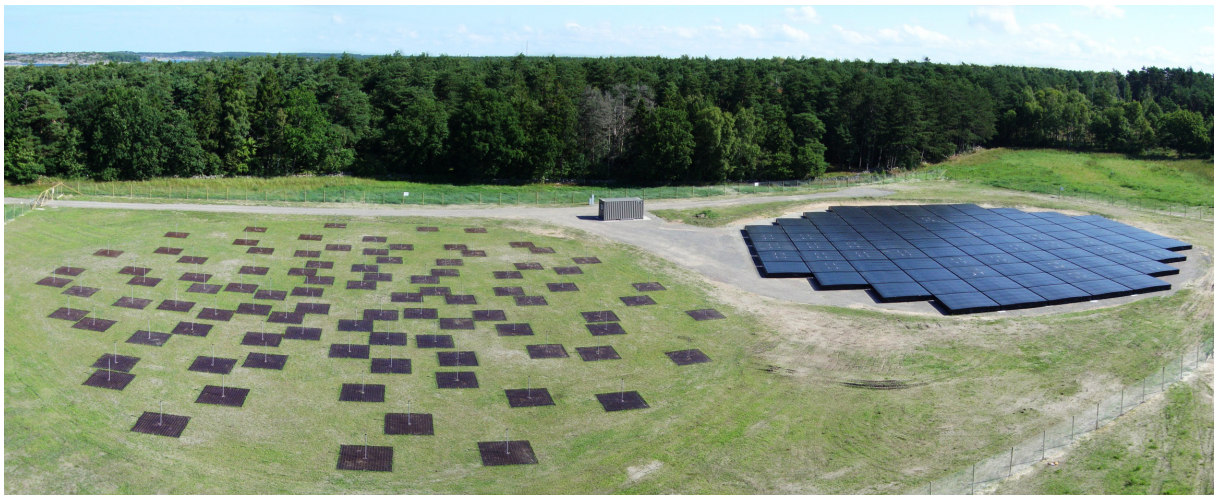

Figure 2: LOFAR low-band (left) and high-band (right) antennas in Onsala, Sweden[7].

Parallelism and AVX2

Pipelines like DP3 that are tasked with processing large amounts of data make heavy use of parallelism to achieve their goal in a reasonable amount of time. Currently DP3 implements multithreading and Single Instruction Multiple Data (SIMD). More detail on different parallelization strategies can be found in section 2.2.

We chose to focus on SIMD for this thesis. SIMD instruction set extensions have been commonly available in x86 since the late 1990s, with more extensions being added regularly. The processor is extended with large registers that can contain multiple values side by side, and operations can be done on all values in a register at once. Since the 1990s, SIMD on x86_64 has evolved into the AVX-

512 instruction set extension, which includes 512-bit registers that can be used to represent 8 64-bit values or 16 32-bit values. This thesis will focus on the 256-bit wide AVX2, AVX-512's predecessor. This older extension was introduced in 2013 and is included in most available processors, unlike AVX-512 which at time of writing can only be found in certain newer high performance processors. Of note in AVX2 is the fused multiply add (FMA) family of instructions, which compute $\pm a * b \pm c$ in one instruction in the same time it takes to execute a single addition or multiplication instruction. We used these instructions extensively in our implementations.

Computing sine, cosine, and the exponential function with SIMD
Two kinds of SIMD implementations for sine, cosine, and the exponential function exist: 1-to-1 and many-to-many functions. 1-to-1 functions take one floating point number as input, and return one floating point number as output. These function use a mix of polynomial approximations and lookup tables. Polynomial approximations can be sped up with SIMD by computing multiple iterations at once, but lookup tables do not benefit from SIMD. Because of this, 1-to-1 functions are not ideal for the kind of optimizations we aim for in DP3.

The other kind of function, many-to-many, takes an entire vector of numbers as input and returns a vector of matching outputs. Because a lookup table cannot be used to look up multiple numbers at once, these functions are limited to only polynomials and similar approximations. One iteration of a polynomial series can be calculated for multiple numbers at once, since the polynomial constants are the same for each input number.

The most well known polynomial approximation is the Taylor series[8], which is a polynomial series approximating a function around a reference point, typically zero. As the input strays further from the reference point, the series loses accuracy. To correct this, the input can be folded to a range close to the reference point. For the sine and cosine function this can easily be done by using the periodic nature of the functions, folding the inputs from a range of $(-\infty, \infty)$ to for example a range of $[-\pi, \pi]$. The exponential function is not periodic and requires a more complicated approach. We use bit manipulation to convert the input into an integer and fractional component, and then compute these separately as powers of two. For more about the exponential function, and our implementations in general, see section 3.

Instead of using a Taylor series, we chose Remez's method for finding polynomial approximations[9]. Finding a polynomial series by hand with Remez's is more complicated than for a Taylor series, so we used the existing lolremez tool to find the polynomials[9]. Additionally, we used Horner's method[10] which allows us to evaluate polynomials with only chained multiply and add instructions. AVX2 has fast chained multiplication and addition built in with its FMA family of instructions, as discussed earlier in this section.

1.2 Related work

Fast and energy efficient code is highly desirable in radio astronomy, and as such much previous work has been done to optimize DP3 and other pipeline components. Earlier research has been carried out by Sclocco et al. implementing beamforming using multithreading, CUDA, OpenCL, and SIMD[11]. While this proved that parallelization is effective for beamforming, it uses the SSE instruction set extension for its SIMD implementation, which is outdated compared to the instructions available on the modern LOFAR systems.

A different approach to increasing efficiency of the pipeline was investigated and implemented by Chege et. al.[12], who proved that lossy compression can be used on the data who used lossy compression on LOFAR data to reduce it to a quarter of its original size without significantly impacting the quality of the data.

Other parts of radio astronomy pipelines are being optimized as well. For example the GPU accelerated correlator and beamformer Cobalt from Broekema et al. [13], or the parallelized improvements made to WSClean by Rubeis et al.[14].

2 Methodology

For this thesis we used the DAS-6 research compute cluster[15] for all stages of the optimization process. Specifically, we used a node on the ASTRON instance of DAS-6 containing an AMD EPYC 7282 processor. We compiled all code on this processor with `g++` with the `-O3` and `-march=native` options. We configured DP3 to use 32 threads for all runs, and assigned it a matching 32 hardware threads. A brief implementation in C(++) is included in Appendix A: C(++) code, and the full project code can be found on GitHub[1].

2.1 Profiling

To profile DP3 we used an input dataset and a run command provided by a DP3 profiling guide[16] using the perf profiler[17]. Perf can statistically sample a program's call stack while it runs, giving a good indication of how much time the program spends in each function.

Before starting the thesis, field experts indicated they suspected the EveryBeam library[6], one of DP3's dependencies, to be a major bottleneck. Because of this our profiling focused around the Predict stage in DP3, which is where all calls to EveryBeam originate from. However, as can seen in Table 1 and Figure 3, the EveryBeam library makes up only about 4% of the benchmarks' execution time.

Function	Time (s)	Percentage
`__ieee754_exp_fma`	321.06	25.490
`epx@@GLIBC_2.29`	152.09	12.075
`__cos_fma`	94.19	7.478
`__sin_fma`	92.05	7.308
EveryBeam	50.60	4.017
Other	549.61	43.63

Table 1: Time the initial DP3 run spent in EveryBeam, sin, cos, and exp functions.

[1] https://github.com/sam-van-hoijtema/avx2_sin_cos_exp

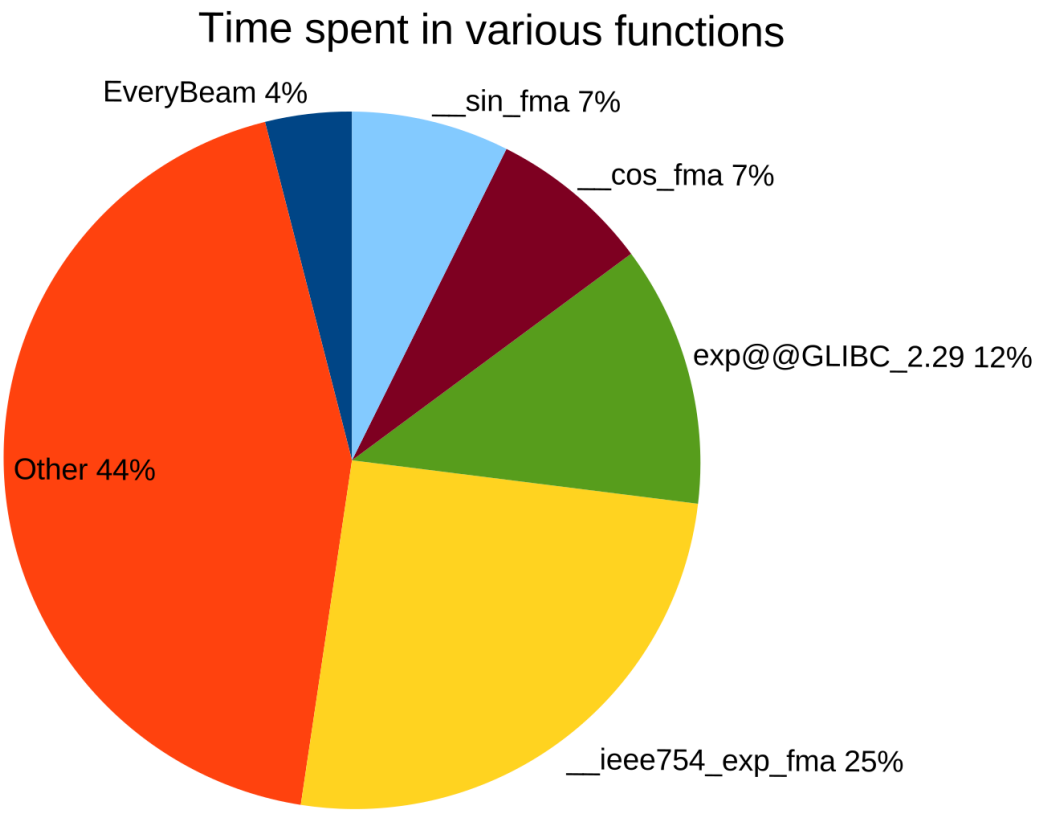

Figure 3: Time the initial DP3 run spent in EveryBeam, sin, cos, and exp functions.

Besides EveryBeam, Table 1 and Figure 3 highlight four other functions. `__ieee754_exp_fma` and `epx@@GLIBC_2.29` are both implementations of the exponential function, `__cos_fma` is an implementation of the cosine function, and `__sin_fma` is an implementation of the sine function. All four of these functions originate from calls to LibC via `std::exp`, `std::cos`, and `std::sin` respectively. Together, these functions make up more than half of the execution time of the DP3 run. These are the functions we focus on for optimization.

2.2 Optimization strategy

DP3 spends most of its time processing large multi-dimensional arrays of floating point numbers. This applies to the four bottleneck functions from section 2.1 that we want to target for optimization. Since the code surrounding these functions does not have any risk of race conditions, the main way to optimize them is with parallelization. Three methods for parallelization are available: hardware accelerators, multithreading, and Single Instruction Multiple Data (SIMD) instruction set extensions.

Hardware accelerators
The best performance for large array operations is typically achieved with hardware accelerators such as GPUs or NPUs. For scientific computing the most popular option is Nvidia's CUDA, which is not currently used in DP3. Using it could greatly increase the performance of the pipeline, although large sections of code would have to be rewritten for it to be effective. Copying a regular matrix into a CUDA buffer has a large overhead which can quickly negate the benefits of parallelization. The major structural changes necessary to effectively implement CUDA in DP3 would take more time than is allocated for this project. Additionally, the DP3 pipeline is required to run on desktop computers or laptops which may not support CUDA.

Other APIs for hardware acceleration besides CUDA exist, many of which will function on a much wider range of hardware. However, these APIs require the same extensive structural changes as CUDA and tend to be more time consuming to work with. In general, we concluded that hardware accelerators are not worth pursuing for this project.

Multithreading
Multithreading is already implemented in parts of DP3, using the threadpool interface from the AOCommon library [18]. Specifically, the Predict pipeline stage is split into a number of threads specified by the user at the start of the program, after which most of the stage is run fully in parallel. This implementation causes some duplicate work to be done between threads, but is over all close to optimal. Fully optimizing it would result in a small speedup while requiring a rewrite of large portions of the pipeline. Improving multithreading in DP3 is a viable option for optimization, albeit not a preferred one due to the large time cost in exchange for only a small improvement.

SIMD
Finally, parallelization can be implemented by using Single Instruction Multiple Data (SIMD) instruction set extensions. SIMD is already used in DP3 through pragmas, which give hints to the compiler about where to insert SIMD instructions. This has the benefit of being easy to implement, but leaves the details of the implementation up to the compiler. If the compiler cannot find a way to safely insert SIMD instructions, the code will not be parallelized. Finding out where the compiler did or did not insert SIMD instructions requires either reading the generated assembly or spending extra time profiling.

The loops from which the four bottleneck functions were called from all use SIMD pragmas. However, reading the assembly generated during the profiling in section 2.1, we found that the compiler only inserted the 1-to-1 functions highlighted in section 2.1. These function are not optimal for processing large arrays like the ones in DP3. For our main optimization strategy, we chose to write new many-to-many functions for the sine, cosine, and exponential function for DP3, making full use of SIMD intrinsics.

2.3 Benchmarking

To confirm that our new functions are faster than the old implementation while being adequately precise, we used three benchmarks: one to measure precision, one to measure theoretical performance, and one to measure real world performance. Details and results for benchmarks can be found in section 4.

To measure accuracy, we compared the results of our functions to their LibC equivalents. The LibC implementations are not infinitely accurate, but since they are already used and accepted in DP3 we will use them as our target. We compared our implementations to LibC on 2^{16} equally spaced double precision floating point numbers, in a range from -20 to 20. More about this benchmark can be found in section 4.1.

We measured theoretical performance by measuring the time it took for the functions to process 2^{21} random double precision floating point numbers, formatted in a continuous array. We did this 2^{10} times for every function to measure standard deviation. We compared our functions to several other existing implementations: LibC[19], Sleef[20], SVML emulated with SIMD Everywhere[21], and XSimd[22]. For a detailed description of these implementations, as well as the results, see section 4.2.

Finally, we implemented our functions in DP3 and re-ran the profiling command. We ran tests with only the exponential function implemented, only the sine and cosine function implemented, both functions implemented, and with the original unmodified code. For each of these combinations we ran four tests to ensure reliable results. The results can be found in section 4.3.

3 Custom function implementation

All functions presented in this thesis are many-to-many, written for AVX2. Since these functions are meant to be called repeatedly on large arrays, they are set to always inline to avoid repeated stack (de)allocations. Furthermore, we used the SIMD Everywhere library to make sure the code still runs on processors without AVX2 support[21]. This library can translate non-native SIMD instructions to natively supported ones, without impacting native instructions.

During testing we found that our functions spent approximately half their time on `vbroadcastsd` instructions. This x86_64 instruction takes a single double precision floating point value and initializes an AVX2 vector with four duplicates of that value. For every term $c_n x^n + c_{n-1} x^{n-1} + ... + c_1 x + c_0$ in our polynomial approximations, the compiler would generate a `vbroadcastsd` instruction for each $c_n, c_{n-1}, ..., c_1, c_0$ for every function call. To prevent these unnecessary initializations, we define an array of AVX2 vectors initialized with $c_n, c_{n-1}, ..., c_1, c_0$ before calling one of our functions. A pointer to that array is then passed to the function, and the function reads the constants from the array as necessary. This way, `vbroadcastsd` instructions only generated once for each term, saving time. For ease of use we defined the arrays as macros in the same header file as the functions.

Besides polynomial approximations we also pursued an approach for sine and cosine involving the inverse square root $\frac{1}{\sqrt{x}}$. We later abandoned this approach, but by then had already written and tested an `invsqrt()` function. This function turned out to be faster than calling the AVX2 divide and square root functions separately. Pseudocode for this function is included in Appendix C: Inverse Square Root.

3.1 Exponential function

The exponent e^d can be implemented simply as a Taylor series[8]. The Taylor series of e^d converges quickly, allowing for a fast and accurate approximation around $d = 0$ without the need for additional logic. However, the accuracy of this approximation quickly drops as d gets further away from 0. To make the function accurate for any arbitrary d, a more steps are needed.

Because IEEE 754 floating point is formatted as $m * 2^p$, powers of two can be calculated efficiently using bit manipulation. e^d can be rewritten as a power of two with the identity $e^d = 2^{d/ln(2)} = 2^x$. This new exponent $x = d/ln(2)$ is then split into an integer component x_i and a fractional component x_f. We use a polynomial approximation for 2^{x_f}, which is highly accurate since x_f is guaranteed to be close to 0. x_i is then added to the exponent of the floating point representation of 2^{x_f}. In other words, given the floating point number $2^x = m * 2^p$, we compute $2^x = 2^{x_i+x_f} = 2^{x_f} * 2^{x_i} = m_f * 2^{p_f} * 2^{x_i} = m_f * 2^{p_f+x_i}$.

Finally, we replaced the Taylor series with a faster converging polynomial found using Remez's method [9]. We used the lolremez tool to find the polynomial constants[23]. See Figure 4 for pseudocode, and Appendix A: C(++) code for a C(++) implementation including the polynomial constants.

```
exp(d) {
  64-bit float: d, x, x_fraction, result
  64-bit integer: x_integer
  out: result

  x = d / ln(2)
  x_integer = x
  x_fraction = x - x_integer

  result = 0
  for p in polynomial_constants:
  result = result * x_fraction + p

  result = bitwise_to_float(bitwise_to_int(result) + (x_integer << 52))
}

```

Figure 4: Pseudocode for the proposed exponential function.

3.2 Sine

The sine function converges slower than the exponential function. A pure Taylor series with double precision accuracy has constants that require rounding to fit in a double precision floating point register, reducing accuracy. Part of the solution is to fold all possible inputs into the range $[-\frac{\pi}{4}, \frac{\pi}{4}]$ using the symmetrical nature of the sine function. Additionally, a different polynomial series can be used. The polynomial constants for this implementation were found using Remez's method, again using the lolremez tool. Instead of approximating $sin(x)$ the polynomial approximates $sin(2\pi x)$, since this function is easier to fold to $[-\frac{\pi}{4}, \frac{\pi}{4}]$. Converting from $sin(x)$ to $sin(2\pi x)$ can be done with only one multiplication. See Figure 5 for pseudocode, and Appendix A: C(++) code for a C(++) implementation including the polynomial constants.

```
sine(x) {
  64-bit float: x2, result
  out: result

  x = (x - floor(x)) / 2π
  x = max(min(0.5 - x), -0.5 - x)

  x2 = x * x

  double result = 0
  for p in polynomial_constants:
  result = result * x2 + p

  result = result * x
  return result
}

```

Figure 5: Pseudocode for the proposed sine function.

3.3 Cosine

The cosine function is identical to the sine function, shifted by a quarter phase. Creating a function from scratch using the same method as sine is possible, but a faster approach is to adjust for the quarter phase difference during folding. This also eliminates one maximum and one subtract instruction, making the function slightly faster. See Figure 6 for pseudocode, and Appendix A: C(++) code for a C(++) implementation including the polynomial constants.

```
cosine(x) {
  64-bit float: x, x2, result
  out: result

  x = (x - floor(x)) / 2π
  x = min(0.25 - x, 0.25 + x)

  x2 = x * x

  double result = 0
  for p in polynomial_constants:
  result = result * x2 + p

  result = result * x
}

```

Figure 6: Pseudocode for the proposed cosine function.

3.4 Sincos

Some methods of calculating sine or cosine allow for easy modification to calculate both at the same time at little extra cost. This is not the case for this thesis's proposed sine function. Both sine and cosine start by multiplying by $\frac{1}{2\pi}$ and removing the integer component, which can be done once for both functions in a sincos implementation. The benefit of this is minor. Alternating between polynomial iterations of sine and cosine might allow for slightly better cache performance by immediately re-using each constant once. This also allows the processor to chain together more fused multiply add instructions, allowing for better pipelining behavior. These last two optimizations are likely to be made by the compiler with the `-O3` option. See Figure 7 for pseudocode.

```
sincos(x) {
  64-bit float: x, x2, sin_result, cos_result
  out: sin_result, cos_result

  x = ()x - floor(x)) / 2π

  cos_x = min(0.25 - x, 0.25 + x)
  sin_x = max(min(0.5 - x), x - 0.5)

  sin_x2 = sin_x * sin_x
  cos_x2 = cos_x * cos_x

  sin_result = 0
  cos_result = 0
  for p in polynomial_constants:
  sin_result = sin_result * sinx2 + p
  cos_result = cos_result * cos_x2 + p

  sin_result = sin_result * sin_x
  cos_result = cos_result * cos_x
}

```

Figure 7: Pseudocode for the proposed sincos function.

4 Benchmarking and results

To confirm the proposed function's usefulness in real world applications, we performed three kinds of benchmarks. We tested the accuracy of the function, theoretical performance on a large continuous array, and practical performance in the DP3 pipeline. We compiled all tests with the `g++` compiler with `-O3` optimization. The `-march=native` flag is also required to ensure AVX2 load and stores only happen on 32-byte aligned memory.

4.1 Accuracy benchmarks

Inaccuracies in SIMD sine, cosine, and exponential functions can arise from multiple sources. Some error is inherent in the polynomial approximations used, although this can largely be mitigated. Inputs to the functions are reduced to always be close to 0, where the polynomial is most accurate. The amount of terms in the function can then be adjusted for the theoretical error to fall below the smallest errors representable by double precision floating point. Inaccuracies also are also introduced in the reduction step, especially in the first step in the sine and cosine functions. This division, implemented as a multiplication in the actual C(++) code, always loses some accuracy due to the nature of the floating point format.

Discrepancies between custom functions and the LibC standard can also arise from the nature of AVX2 instructions. For example, the AVX2 implementation uses fused multiply add instructions, which compute a multiplication and addition in sequence without rounding in between. An equivalent scalar function uses a separate multiply and add instruction which *do* round in between. This discrepancy in rounding means that the output of an equivalent AVX2 and scalar function might not always be bit for bit identical.

To measure different kinds of errors we used three variants of each function: the LibC implementation, the proposed AVX2 implementation, and a scalar function using the same approach as the proposed AVX2 implementation. The error inherent to the function can be measured by comparing the LibC and scalar implementations. The error arising from AVX2 instructions can be measured by comparing the AVX2 and scalar implementations. Finally, the total error is measured by comparing our AVX2 and LibC implementations.

The accuracy of the approximation $\hat{y}$ can be measured as either the absolute error $|y - \hat{y}|$ or relative error $\frac{|y-\hat{y}|}{y}$. For the sine and cosine the relative error can be misleading when measured over a range larger than half the period of the function, since the error approaches infinity as x approaches 0 every half period. The opposite is true for the exponential function. Because the function grows exponentially the absolute error rapidly increases, while the amount of correct significant digits stays the same. For these reasons we give the relative error for the exponential function, and the absolute error for the sine and cosine function.

Errors for the exponential function can be found in Figure 9 and errors for the sine and cosine functions can be found in figure 8. Since the sincos function uses the same implementation as the individual sine and cosine functions, its errors are the same as the individual functions. As can be seen in the figures, the absolute and relative errors both stay well below 10^{-14} on an input range from -20 to 20.

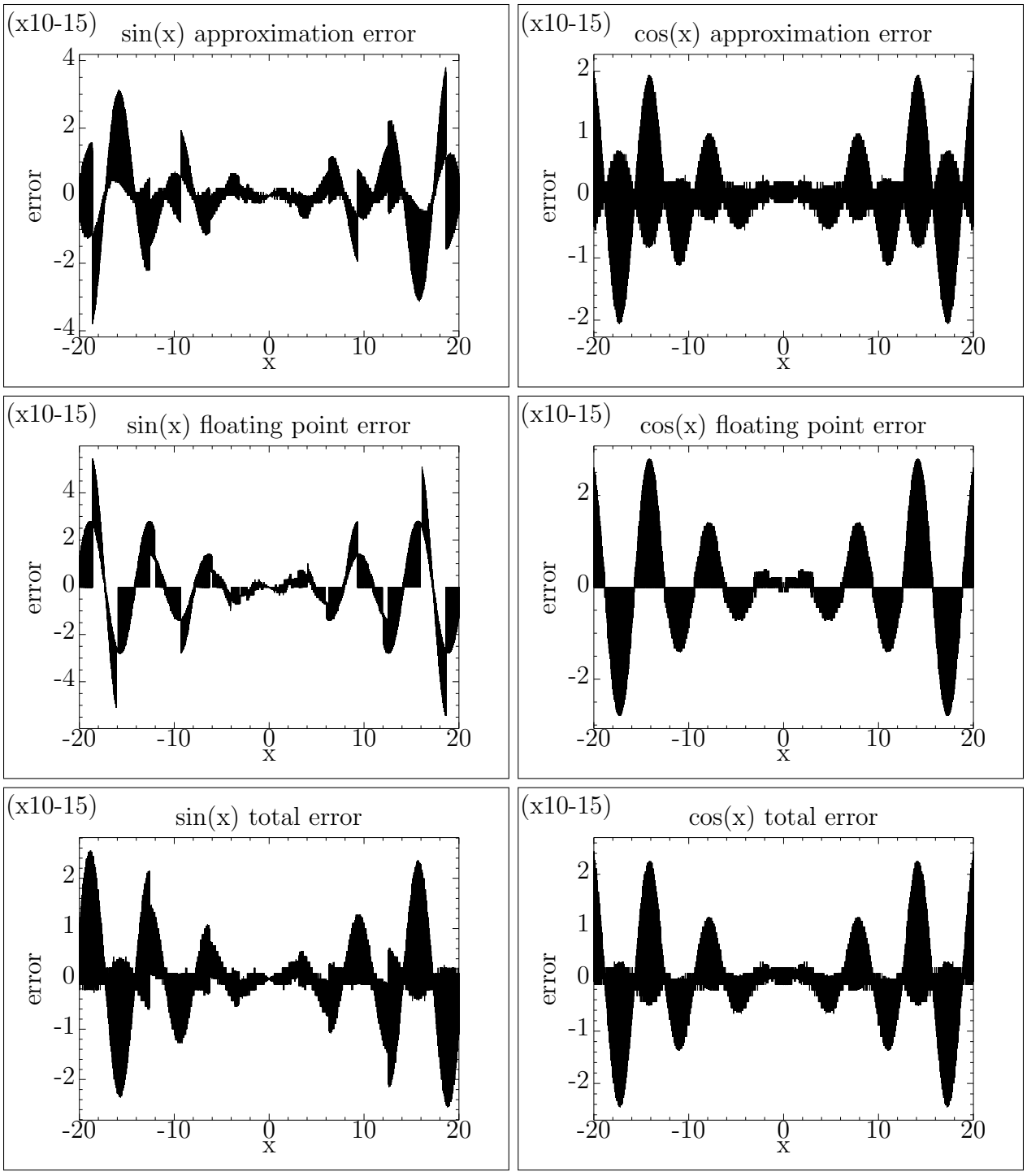

Figure 8: Error in the sine and cosine functions, caused by the polynomial approximation, floating point inaccuracies between SIMD and scalar functions, and the combined error of both.

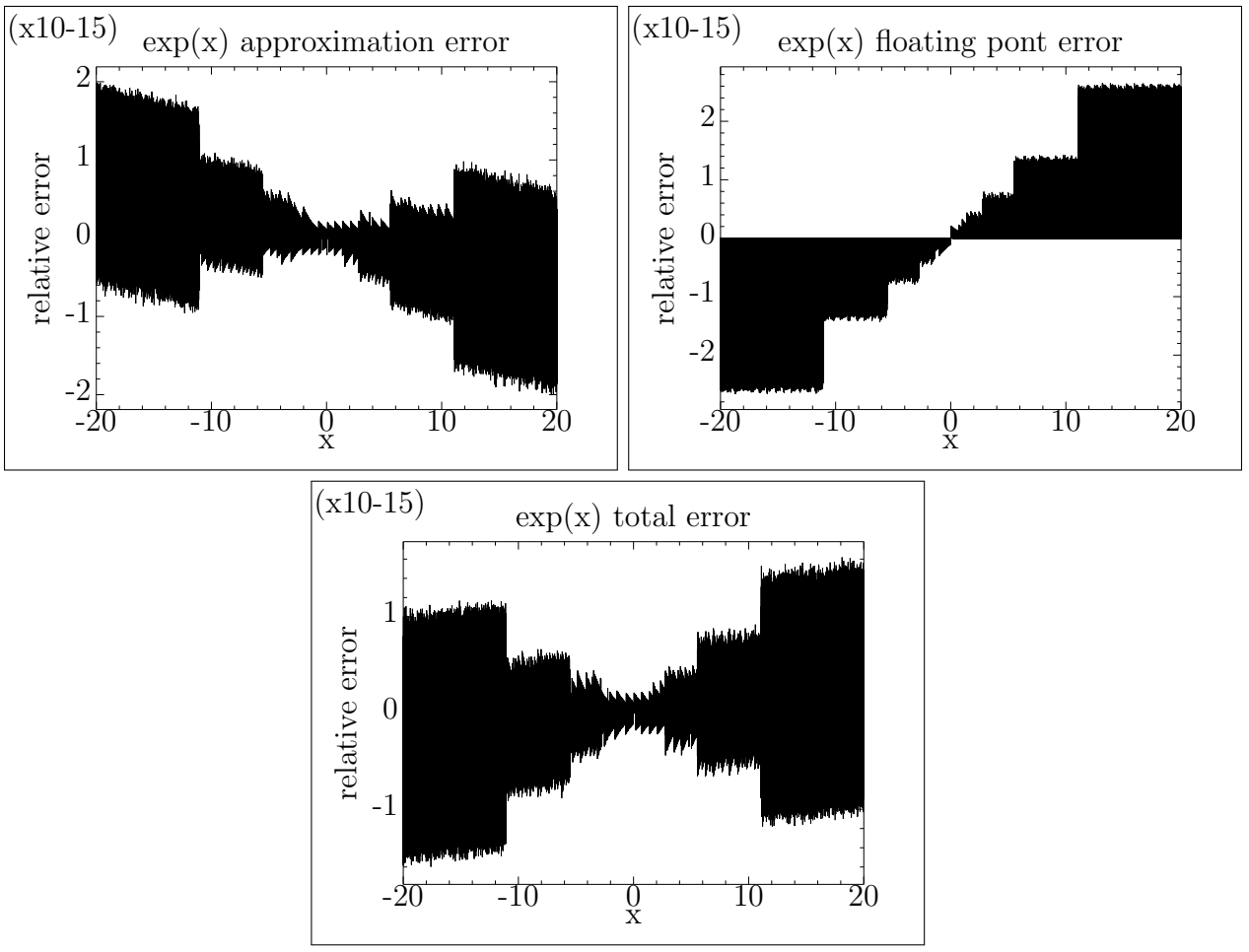

Figure 9: Relative error in the exponential function, caused by the polynomial approximation, floating point inaccuracies between SIMD and scalar functions, and the combined error of both.

4.2 Theoretical performance

To test the theoretical performance of our functions, we compared them to existing AVX2 double precision implementations of sine, cosine, and the exponential function. We chose LibC[19], Sleef[20], SVML emulated through SIMD Everywhere[21], and XSimd[22] for comparison. LibC is included twice, once when compiled normally and once when compiled with the `-ffast-math` option. These implementations are a mix of 1-to-1 and many-to-many.

LibC and `-ffast-math`
The implementations of sine, cosine, and the exponential function in LibC are all 1-to-1 SIMD functions by default. When compiled with the `-ffast-math` flag, these functions are turned into many-to-many functions instead based on the hardware available. Enabling this flag disables certain safety features in GCC, such as checks to prevent incorrect results with infinite values or NANs[2]. For large code bases like DP3 `-ffast-math` is not recommended, but for simple programs such as this benchmark it causes no problems, so results are included for completeness.

Sleef
Sleef is a library that provides 1-to-1 vectorized implementations of most elementary functions. For sine and cosine two version are included with 1 ULP and 3.5 ULP precision, both of which will be included in the results. For the exponential function only a 1 ULP function is provided.

SIMD Everywhere
SVML is a SIMD instruction set extension made by Intel. It includes more advanced SIMD instructions compared to AVX2 including sine, cosine, and the exponential function. However, the instruction set extension is closed sourced and only optimized for Intel branded processors, making it unsuitable for a codebase like DP3. Instead the SIMD Everywhere library is used, which translates non-native SIMD instructions into native ones. Specifically, SIMD Everywhere unpacks the vectors it gets as input and uses functions from LibC to compute each number individually.

XSimd
XSimd is a library that offers many-to-many functions through C++ operator overloading. It also aims to be hardware agnostic, making it useful for the DP3 codebase. In fact, XSimd is already included in parts of DP3, giving it the benefit of not needing additional dependencies.

To compare the theoretical performance of the proposed functions against the four chosen libraries, we used a large array of random numbers. We used an array of 2^{21} random double precision floating point numbers, and carried out the test 2^{10} times. We recorded the execution time for all 2^{10} random arrays for each function and then calculated the mean and standard deviation, which are shown in charts 10 and 11. Because the standard deviation is small the error bars can be hard to see, so the data is also presented as tables in Appendix B: Data tables.

A comparison of the implementations without `-ffast-math` can be seen in Figure 10. As can be clearly seen, our proposed implementation is significantly faster than all 1-to-1 implementations. The many-to-many XSimd implementation is much closer, but still slower than our implementation. Figure 11 shows a comparison between our functions and LibC when compiled with `-ffast-math`.

[2]For a full explanation of the effects of `-ffast-math`, see https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

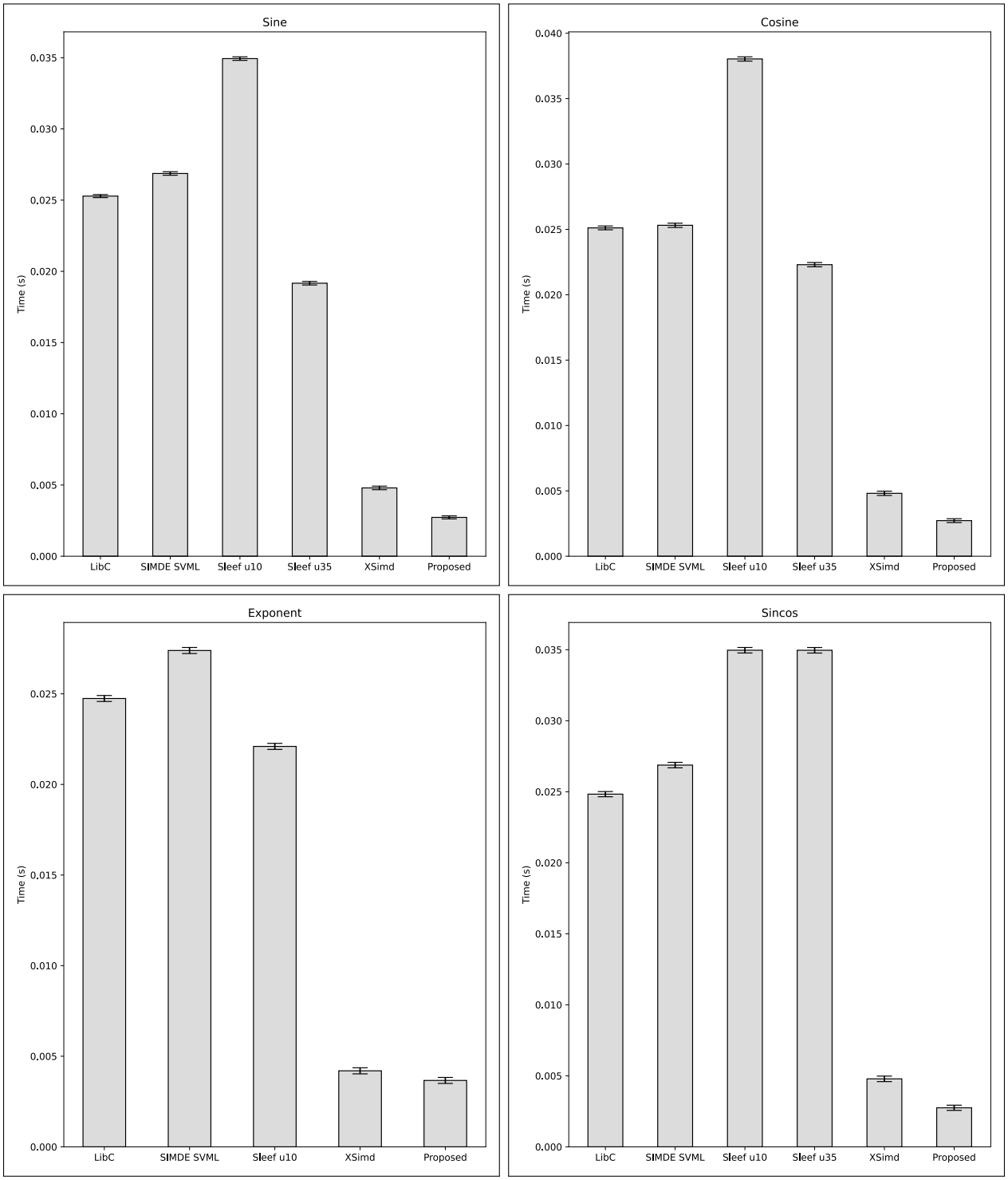

Figure 10: Time taken to process 2^{21} numbers for different implementations of sine, cosine, sincos, and the exponential function.

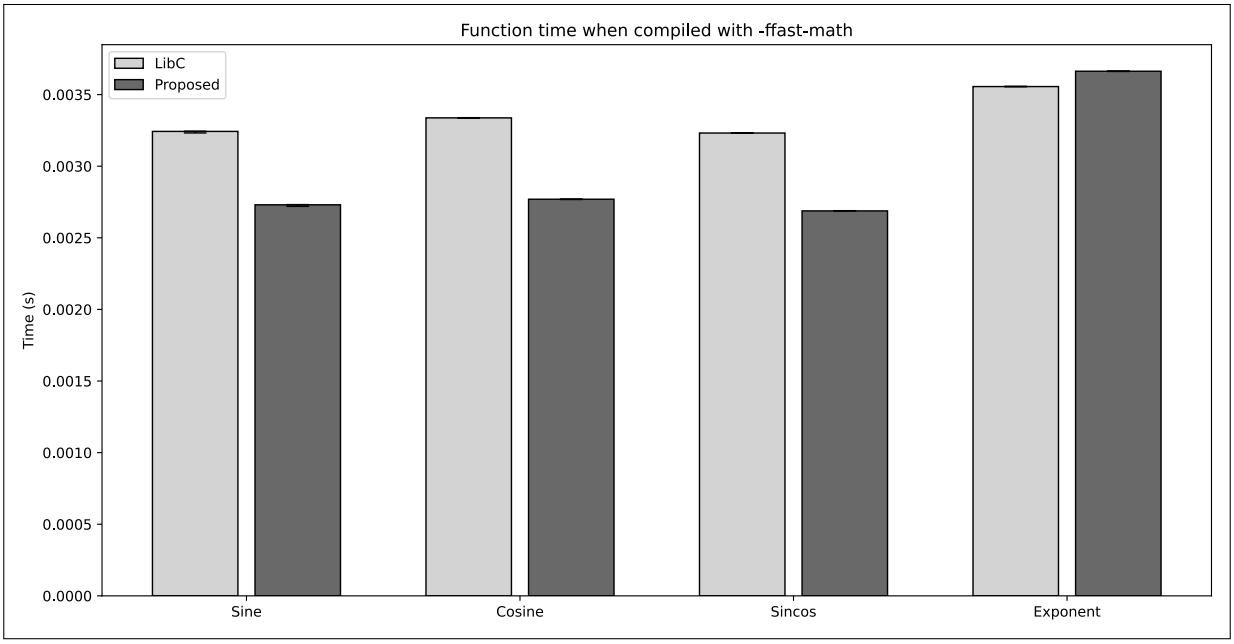

Figure 11: Time taken to process 2^{21} numbers for the LibC and proposed implementations of sine, cosine, sincos, and the exponential function. Functions were compiled with `-ffast-math`.

4.3 DP3 performance

Finally, we added the proposed functions into the DP3 pipeline and compared it to the unmodified version. The `-march=native` flags needed to be added to the CMakeList.txt file to ensure AVX2 loads and stores only happen on 32-byte aligned memory. Small sections of code around the function calls also had to be modified to use AVX2 instructions, again using SIMD Everywhere. We ran tests with only the exponential function implemented, only the sine and cosine function implemented, both functions implemented, and with the original unmodified code. We used perf to record where the program spent its execution time, in the same way as in 2.1. We ran each test four times to ensure reliable results. The data from the benchmarks can be found in 12 as a chart, as well as in table form in Appendix B: Data tables.

Figure 12 shows difference in the execution time between the original and modified versions of DP3. The exponential and sin/cos functions disappear from their respective runs because they are inlined. Inlined functions appear on the stack as part of their parent function, making them effectively invisible to perf. On average, the original benchmark took 901.0 seconds, the sin/cos modified version took 819.3 seconds, the exp modified version took 670.8 seconds, and the fully modified version took 624.1 seconds. This translates to an improvement of respectively 9.98% and 34.3% for the sin/cos and exp modified version, for a combined 44.4% improvement.

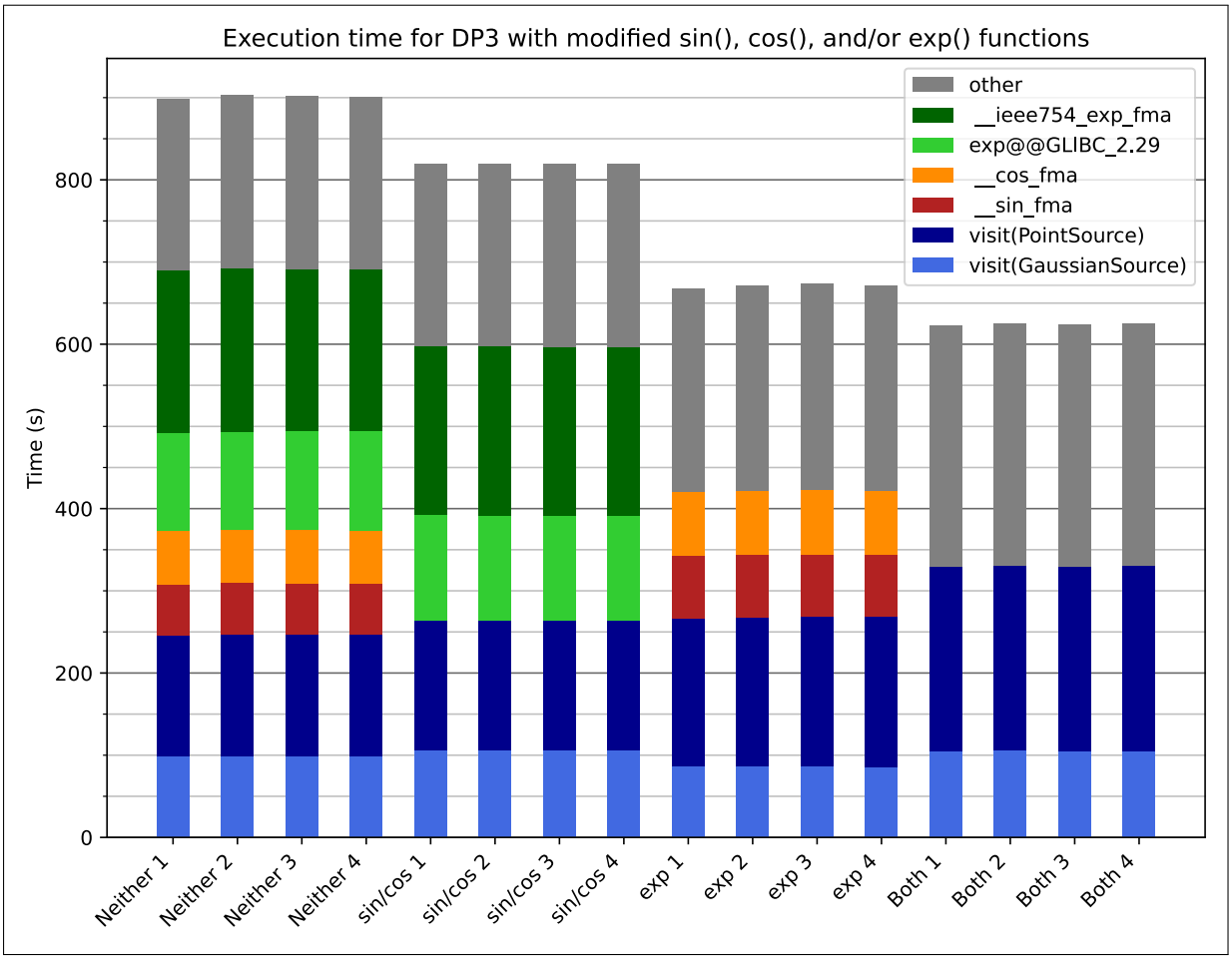

Figure 12: Wall clock execution time of DP3, including and not including the modified sine/cosine and exponent functions. Each variant was tested 4 times.

5 Discussion

5.1 Approximation error

While the error of less than 10^{-14} we achieved is adequate for most use cases, it could be much more accurate. All the libraries that we used for comparison have an accuracy of 1 ULP or less, with the exception of the 3.5 ULP functions from Sleef. While the polynomials we chose all have accuracies that should give <1 ULP precision, the argument reduction done beforehand does not. Specifically, the first step of dividing by 2π or $ln(2)$ (implemented as multiplications with the inverse) loses accuracy for larger input values. This could be mitigated by calculating the high and low halves of the reduction separately, at a small performance cost. This would likely put the proposed implementations on par with XSimd, both in accuracy as well as execution time. Alternatively, if the $< 10^{-14}$ accuracy is deemed adequate, polynomials with fewer terms could be used to achieve a small speedup.

5.2 Sincos performance

The sincos function we presented is only marginally faster than the proposed sine and cosine function executed separately. This is inherent to the way the functions are implemented. A single line of C(++) code can be re-used, the chain of fused multiply add instructions could give better CPU pipelining behavior, and because the polynomial constants are the same for each input in the vector the function might get better cache performance. The benefits from these effects are minor however, and when computing sine and cosine separately the compiler might already add these optimizations.

Instead of re-using the polynomial approximation from sine and cosine, a different algorithm such as CORDIC could be used instead[24]. With algorithms like these the process for calculating sine and cosine is identical for all but the final step, meaning there is very little difference between calculating only the sine or cosine, and calculating both at once. Additionally, a fast sincos function would double as a fast tangent function at the cost of only one additional divide instruction. More research is needed to determine if this approaches like CORDIC converge fast enough to save time compared to the polynomial approximation we used.

5.3 Further application to DP3

The DP3 benchmark we used only focused on one of the possible 42 pipeline stages available in DP3. While some areas of DP3 are already optimized, others still need work. This is a possible direction for future research. The functions in this thesis can likely be re-used elsewhere in DP3, but to find these areas additional profiling needs to be carried out. Profiling the entire pipeline introduces additional logistical challenges, such as long execution times on the available hardware.

Furthermore, not all parts of DP3 use double precision floating point values. It is possible that rounding occurs in other parts of DP3, making the part of the 14 significant decimal digit precision superfluous. If this is true, iterations can be removed from the polynomial approximations to speed up the functions even further, since the extra accuracy is not required. More research is needed to confirm if this is possible.

5.4 AVX-512

The functions in this thesis are all implemented using AVX2. We decided on this because the DAS-6 server as well as any other devices we have access to are relatively old, and do not support more modern instruction set extensions. Specifically the AVX-512 family of instructions is interesting, allowing for a theoretical speedup of up to 2× on top of the results shown here. We wrote all C(++) code with AVX-512 in mind, such that only some macros have to be edited to convert all functions to native AVX-512. The use of SIMD Everywhere means that code written for AVX-512 should still compile and perform reasonably on hardware limited to AVX2.

5.5 OpenMP and other pragmas

This thesis focused exclusively on explicit SIMD functions. SIMD can also be inserted by adding pragmas, which tell the compiler that a certain section of code can be converted to SIMD. DP3 already uses the `ivdep` pragma that is built into GCC, and could easily add the widely used OpenMP library of pragmas to achieve something similar. We chose not to investigate these options since the compiler can decide not to insert SIMD. Verifying that SIMD is inserted correctly every time the project is compiled would require studying the relevant sections of assembly, which would have not fit in the time allocated for this thesis. This is a possible area for future research.

5.6 Effects of other optimizations to DP3

While this thesis focuses on optimization through SIMD, we also applied other small optimizations where practical. For instance, the new functions were set to always inline by adding the GCC `always_inline` attribute to their declarations. Given that the standard LibC functions appeared in the perf call stack and therefore were not always inlined, it is likely that some of the final DP3 speedup is due to inlining. We also had to convert some existing scalar DP3 code to SIMD before and after calling our new functions. These may have brought an additional speedup as well.

6 Conclusion

In this thesis we presented implementations of the sine, cosine, and exponential functions which compute four double precision floating point numbers in parallel using AVX2. This was done after profiling a section of the DP3 pipeline and finding over half of its wall clock time to be spent in sine, cosine, and exponent functions. Additionally, we created a sincos function that computes both the sine and cosine marginally faster than the individual sine and cosine functions.

We demonstrated that the proposed exponent function has a relative error near 10^{-15} when compared to LibC for inputs in a range from -20 to 20. The proposed sine and cosine functions have an absolute error near 10^{-15} compared to LibC in the same input range. We then compared our functions to existing implementations in a time benchmark on large arrays of random numbers. The libraries used for comparison were LibC, SVML emulated through SIMD Everywhere, XSimd, and Sleef. We showed the LibC exponential function to be faster than the proposed function when compiled with `-ffast-math`. The proposed functions were faster than existing implementations in all other tests.

Finally, we added the new functions to the relevant section of DP3 and compared the wall clock execution time before and after. After adding the new functions execution time decreased from an average of 901.0 to 624.1 seconds for a total 44.4% speedup.

All four functions have room for improvement still. The error of the functions is acceptable for most applications, but well above the ideal of <1 ULP. The execution time of the functions, especially that of sincos, could also be reduced further in the future. Finally, more profiling of DP3 should be done to possibly apply the same optimizations to other parts of the pipeline.

References

[1] van Haarlem, M. P. et al. "LOFAR: The LOw-Frequency ARray". In: *A&A* 556 (2013), A2. DOI: 10.1051/0004-6361/201220873. URL: https://doi.org/10.1051/0004-6361/201220873.

[2] Peter E. Dewdney et al. "The Square Kilometre Array". In: *Proceedings of the IEEE* 97.8 (2009), pp. 1482–1496. DOI: 10.1109/JPROC.2009.2021005.

[3] T. J. Dijkema et al. *DP3: Streaming processing pipeline for radio interferometric data*. Astrophysics Source Code Library, record ascl:2305.014. May 2023. ascl: 2305.014.

[4] LOFAR / ASTRON. *LOFAR Superterp*. 2010. URL: https://commons.wikimedia.org/wiki/File:LOFAR_Superterp.jpg (visited on 07/12/2026).

[5] W. A. van Cappellen et al. "Apertif: Phased array feeds for the Westerbork Synthesis Radio Telescope: System overview and performance characteristics". In: *Astronomy & Astrophysics* 658 (Feb. 2022), A146. ISSN: 1432-0746. DOI: 10.1051/0004-6361/202141739. URL: http://dx.doi.org/10.1051/0004-6361/202141739.

[6] *EveryBeam*. https://gitlab.com/aroffringa/aocommon. (Visited on 07/12/2026).

[7] Onsala rymdobservatorium/Leif Helldner. *LOFAR Onsala Space Observatory*. 2011. URL: https://commons.wikimedia.org/wiki/File:Lofar_onsala_20110706.jpg (visited on 07/12/2026).

[8] Brook Taylor. *Methodus incrementorum directa & inversa*. Londini : typis Pearsonianis: prostant apud Gul. Innys ad Insignia Principis in Coemeterio Paulino, 1715.

[9] Sherif A Tawfik. "Minimax approximation and Remez algorithm". In: *Lecture Notes* (2005).

[10] C Sidney Burrus et al. "Horner's method for evaluating and deflating polynomials". In: *DSP Software Notes, Rice University, Nov* 26 (2003).

[11] Alessio Sclocco et al. "Radio Astronomy Beam Forming on Many-Core Architectures". In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. 2012, pp. 1105–1116. DOI: 10.1109/IPDPS.2012.102.

[12] J. K. Chege et al. "The impact of lossy data compression on the power spectrum of the high-redshift 21 cm signal with LOFAR". In: *Astronomy amp; Astrophysics* 692 (Dec. 2024), A211. ISSN: 1432-0746. DOI: 10.1051/0004-6361/202451367. URL: http://dx.doi.org/10.1051/0004-6361/202451367.

[13] P. Chris Broekema et al. "Cobalt: A GPU-based correlator and beamformer for LOFAR". In: *Astronomy and Computing* 23 (Apr. 2018), pp. 180–192. ISSN: 2213-1337. DOI: 10.1016/j.ascom.2018.04.006. URL: http://dx.doi.org/10.1016/j.ascom.2018.04.006.

[14] Emanuele De Rubeis et al. *Accelerating radio astronomy imaging with RICK*. 2024. arXiv: 2411.07321 [astro-ph.IM]. URL: https://arxiv.org/abs/2411.07321.

[15] Henri Bal et al. " A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term ". In: *Computer* 49.05 (May 2016), pp. 54–63. ISSN: 1558-0814. DOI: 10.1109/MC.2016.127. URL: https://doi.ieeecomputersociety.org/10.1109/MC.2016.127.

[16] W. van Breukelen. *Running and profiling a pipeline component: DP3*. (Visited on 07/12/2026).

[17] *perf*. https://perfwiki.github.io/main/useful-links/. Version 6.12.95. (Visited on 07/12/2026).

[18] A. Offringa. *AOCommon*. https://gitlab.com/aroffringa/aocommon. (Visited on 07/12/2026).

[19] Inc. Free Software Foundation. *GNU C Library*. https://sourceware.org/glibc/libc.html. Version 2.41. (Visited on 07/12/2026).

[20] Naoki Shibata and Francesco Petrogalli. "SLEEF: A Portable Vectorized Library of C Standard Mathematical Functions". In: *IEEE Transactions on Parallel and Distributed Systems* 31.6 (2020), pp. 1316–1327. DOI: 10.1109/TPDS.2019.2960333.

[21] *SIMD Everywhere*. https://github.com/simd-everywhere/simde. Version 0.8.2. (Visited on 07/12/2026).

[22] J. Mabille and S. Corlay. *XSimd*. https://github.com/xtensor-stack/xsimd. (Visited on 07/12/2026).

[23] S. Hocevar. *lolremez*. https://github.com/samhocevar/lolremez. Version 0.7. (Visited on 07/12/2026).

[24] Nitesh Kumar Sharma, Shanti Rathore, and M. R. Khan. "A Comparative Analysis on Coordinate Rotation Digital Computer (CORDIC) Algorithm and Its use on Computer Vision Technology". In: *2020 First International Conference on Power, Control and Computing Technologies (ICPC2T)*. 2020, pp. 106–110. DOI: 10.1109/ICPC2T48082.2020.9071514.

[25] Intel Corporation. *Intel intrinsics Guide*. (Visited on 07/12/2026).

7 Appendix A: C(++) code

This section contains C(++) code for the exponential and sine/cosine functions. Some minor optimizations have been left out for brevity. Notably, the polynomial constants have been included in the function itself, rather than being defined outside of the function to save time on initializing vectors like discussed in section 3. Also note that the code uses macros instead of intrinsics to be more legible and easier to modify.

7.1 Exponent function

The proposed exponential function relies on a cast from a vector of integers to a vector of doubles. This function does not exist for 64-bit values in AVX2, so it is done via bit manipulation. In other instruction sets lines 28 to 31 can be replaced by a single macro or intrinsic.

```
1  static inline vec_double custom_mm256_exp_pd(const vec_double d) {
2    // Divide by ln(2)
3    vec_double x = mul_pd(d, init_pd(1/0.6931471805599453094));
4
5    // Split x into integer and fraction components
6    vec_double x_integer = floor_pd(x);
7    vec_double x_fraction = sub_pd(x, x_integer);
8
9    // Calculate the fraction component with a polynomial
10   vec_double t = init_pd(3.1574284495200578e-15);
11   t = fma_pd(t, x_fraction, init_pd(6.8332114942789827e-14));
12   t = fma_pd(t, x_fraction, init_pd(1.3691023151418430e-12));
13   t = fma_pd(t, x_fraction, init_pd(2.5677549503920060e-11));
14   t = fma_pd(t, x_fraction, init_pd(4.4455387343856381e-10));
15   t = fma_pd(t, x_fraction, init_pd(7.0549123713751233e-09));
16   t = fma_pd(t, x_fraction, init_pd(1.0178086006608814e-07));
17   t = fma_pd(t, x_fraction, init_pd(1.3215486786624682e-06));
18   t = fma_pd(t, x_fraction, init_pd(1.5252733804068388e-05));
19   t = fma_pd(t, x_fraction, init_pd(0.00015405350393390572));
20   t = fma_pd(t, x_fraction, init_pd(0.0013333558146428428));
21   t = fma_pd(t, x_fraction, init_pd(0.0096181291076284665));
22   t = fma_pd(t, x_fraction, init_pd(0.055504108664821583));
23   t = fma_pd(t, x_fraction, init_pd(0.24022650695910072));
24   t = fma_pd(t, x_fraction, init_pd(0.69314718055994529));
25   t = fma_pd(t, x_fraction, init_pd(1.0));
26
27   // Convert x_integer to from double type to int type
28   x_integer = add_pd(x_integer, init_pd(0x0018000000000000));
29     vec_int exponent = sub_pi(bitcast_pd_pi(x_integer),
30     bitcast_pd_pi(init_pd(0x0018000000000000))
31   );
32
33   // Calculate the integer component with a shift
34   exponent = slli_pi(exponent, 52);
35
36   // Merge the integer and fraction result
37   exponent = add_pi(exponent, bitcast_pd_pi(t));
38   t = bitcast_pi_pd(exponent);
39
40   return t;
41 }
```

7.2 Sine, cosine, and sincos function

The proposed sine and cosine functions are nearly identical, differing only by one line. We provide one function with both lines, one of which has to be commented out before use. The function can also be easily modified to create a sincos function.

```
1
2  static inline vec_double custom_mm256_cos_pd(vec_double x) {
3    x = mul_pd(x, init_pd(1/(2*M_PI)));
4
5    x = sub_pd(x, round_pd(x));
6
7    // Only this step differs between sin and cos
8    // One of them should be enabled, one disabled
9    // enable this line for sine
10   x = max_pd(min_pd(x, sub_pd(init_pd(0.5), x)), sub_pd(init_pd(-0.5), x));
11   // enable this line for cosine
12   x = min_pd(sub_pd(init_pd(0.25), x), add_pd(init_pd(0.25), x));
13
14   vec_double x2 = mul_pd(x, x);
15
16   vec_double t = init_pd(0.100856328715300938);
17   t = fma_pd(x2, t, init_pd(-0.717723579656548246));
18   t = fma_pd(x2, t, init_pd(3.819927050429970912));
19   t = fma_pd(x2, t, init_pd(-15.094641620507466994));
20   t = fma_pd(x2, t, init_pd(42.058693923785165622));
21   t = fma_pd(x2, t, init_pd(-76.705859752798496383));
22   t = fma_pd(x2, t, init_pd(81.605249276073406089));
23   t = fma_pd(x2, t, init_pd(-41.341702240399756224));
24   t = fma_pd(x2, t, init_pd(6.283185307179586473));
25   return t;
26 }
```

8 Appendix B: Data tables

This section includes the data in figures 10, 11, and 12 formatted as tables.

8.1 Array benchmark time

The data in the following table corresponds to Figure 10.

Function	Mean time (s)	Standard deviation
sin LibC	0.025284380419180	0.000105921612837
sin SIMDE SVML	0.026871592272073	0.000123534447248
sin Sleef u10	0.034936328185722	0.000124436427725
sin Sleef u35	0.019167597871274	0.000124914219692
sin XSimd	0.004795578541234	0.000124208496250
sin Proposed	0.002726264065132	0.000105935656389
cos LibC	0.025112946983427	0.000145485738853
cos SIMDE SVML	0.025310893077403	0.000162358934730
cos Sleef u10	0.038033361779526	0.000162850612845
cos Sleef u35	0.022299806820229	0.000162858139733
cos XSimd	0.004810565151274	0.000162840859382
cos Proposed	0.002722084056586	0.000145494453271
sincos LibC	0.024830349721014	0.000182376338187
sincos SIMDE SVML	0.026875967858359	0.000193145194014
sincos Sleef u10	0.034963388228789	0.000194550542422
sincos Sleef u35	0.034959994256496	0.000195555533512
sincos XSimd	0.004787107463926	0.000193545761717
sincos Proposed	0.002749597653747	0.000182385352915
exp LibC	0.024742531822994	0.000165649848674
exp SIMDE SVML	0.027390739880502	0.000167870847350
exp Sleef u10	0.022098413668573	0.000167966151650
exp XSimd	0.004196269204840	0.000167877362208
exp Proposed	0.003664195304736	0.000165656259601

Table 2: Wall clock time on 2^{21} random numbers for different implementations of sin(), cos(), exp(), and sincos(). Time is averaged over 2^{10} runs. Compiled with `g++ -O3`.

8.2 Array benchmark time with `-ffast-math`

The data in the following table corresponds to Figure 11.

Function	Mean time	Standard deviation
sin LibC	0.003242966486141	0.000010721995470
sin Proposed	0.002730449195951	0.000001391255599
cos LibC	0.003337412374094	0.000001772616756
cos Proposed	0.002769435988739	0.000001437194047
sincos LibC	0.003556066658348	0.000001468909805
sincos Proposed	0.003663174342364	0.000001468948928
exp LibC	0.003231715178117	0.000001482188652
exp Proposed	0.002688003005460	0.000002023217898

Table 3: Wall clock execution time on 2^{21} random numbers for the proposed and LibC implementations of sin(), cos(), exp(), and sincos(). Time is averaged over 2^{10} runs. Compiled with `g++ -O3 -ffast-math`.

8.3 DP3 execution time

The data in the following tables corresponds to Figure 12.

Run	1	2	3	4
visit(GaussianSource)	210637	210733	210738	211191
visit(PointSource)	313110	313233	313267	312984
__sin_fma	134156	134101	133797	133727
__cos_fma	137938	136759	136655	137006
exp@@GLIBC_2.29	255039	254047	255807	255804
__ieee754_exp_fma	420970	420403	419630	420009
Total	1917379	1915531	1916153	1915854
Time (s)	898.08	902.53	902.26	901.07

Table 4: Samples collected by perf in different functions during 4 runs of DP3, without modifications. Also including the total execution time reported by DP3.

Run	1	2	3	4
visit(GaussianSource)	214508	214939	214971	214381
visit(PointSource)	319157	318772	318115	318119
__sin_fma	0	0	0	0
__cos_fma	0	0	0	0
exp@@GLIBC_2.29	257769	258389	257232	257288
__ieee754_exp_fma	414271	415514	414604	415382
Total	1652983	1655017	1653659	1653858
Time (s)	819.57	819.02	818.88	819.56

Table 5: Samples collected by perf in different functions during 4 runs of DP3, with our custom sine and cosine functions implemented. Also including the total execution time reported by DP3.

Run	1	2	3	4
visit(GaussianSource)	151072	151296	150740	150757
visit(PointSource)	318249	318282	319047	319623
__sin_fma	133609	132847	133060	132696
__cos_fma	136328	136789	13621 9	136418
exp@@GLIBC_2.29	0	0	0	0
__ieee754_exp_fma	0	0	0	0
Total	1175127	1176420	1176382	1176736
Time (s)	667.6	671.31	672.94	671.25

Table 6: Samples collected by perf in different functions during 4 runs of DP3, with our custom exponential function implemented. Also including the total execution time reported by DP3.

Run	1	2	3	4
visit(GaussianSource)	162418	163526	162912	162621
visit(PointSource)	347326	346986	347213	347709
__sin_fma	0	0	0	0
__cos_fma	0	0	0	0
exp@@GLIBC_2.29	0	0	0	0
__ieee754_exp_fma	0	0	0	0
Total	964155	966486	964241	965902
Time (s)	622.25	624.96	624.24	624.9

Table 7: Samples collected by perf in different functions during 4 runs of DP3, with all our custom functions implemented. Also including the total execution time reported by DP3.

9 Appendix C: Inverse Square Root

During development of the sine and cosine function, we investigated an approach that required computing the inverse square root. While AVX2 has built in functions for division and square root, there is no built in instruction for the inverse square root. Both of these instructions are relatively costly, with 18+14=32 cycles of latency and a throughput of 12+8=20 CPI. When we add up the values from the Intel intrinsics reference [25], we can see that the old "Quake III" fast inverse square root algorithm should be faster. This function when implemented in AVX2 has a theoretical latency of 19-21, and a theoretical throughput of only 3 CPI. Note that the magic number has to be adjusted for a 64-bit function.

We did not pursue the sine/cosine algorithm requiring this function after it became clear that it had too many branches to efficiently compute with AVX2. However, the fast inverse square root function is provided as pseudocode and C(++) for completeness.

```
invsqrt(x) {
  64-bit int: i
  64-bit float: x2, t, result
  out: result

  x2 = x*x
  i = bitwise_to_int(x)
  i = 0x5fe6eb50c7b537a9 - (i >> 1)
  result = bitwise_to_float(i)

  result = result * (threehalfs - x2 * result * result)
}
```

```
static inline vd_t custom_mm256_invsqrt_pd(vd_t number) {
  vec_int i;
  vec_double x2, y, t;
  const vec_double threehalfs = init_pd(1.5);

  x2 = mul_pd(number, init_pd(0.5));
  y = number;
  i = bitcast_pd_pi(y);
  i = sub_pi(init_pi(0x5fe6eb50c7b537a9), srli_pi(i, 1));
  y = bitcast_pi_pd(i);
  y = mul_pd(y, fnma_pd(x2, mul_pd(y, y), threehalfs));
  y = mul_pd(y, fnma_pd(x2, mul_pd(y, y), threehalfs));

  return y;
}
```