universität wien

MASTERARBEIT | MASTER'S THESIS

Titel | Title

Simulating Arbitrary Floating Point Numbers

verfasst von | submitted by

Jakob David Neuhauser BSc

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien | Vienna, 2024

Studienkennzahl lt. Studienblatt | Degree programme code as it appears on the student record sheet:	UA 066 910
Studienrichtung lt. Studienblatt | Degree programme as it appears on the student record sheet:	Masterstudium Computational Science
Betreut von | Supervisor:	Univ.-Prof. Dipl.-Ing. Dr. Wilfried Gansterer M.Sc.

Acknowledgements

I want to express my gratitude to my supervisor, Prof. Dr. Wilfried Gansterer, for his support and guidance throughout the development of this thesis.

Furthermore, I want to thank Katyayani Asen BA for proofreading the thesis and providing support during the process.

Abstract

Deciding which data types to use inside an algorithm is a core subject when designing multiprecision algorithms. Thereby, a usual difficulty is that using high-precision types makes the calculation more precise but simultaneously requires more computation time. Moreover, on current hardware, typically, only a very limited number of different data types are available to conduct analysis. Because of that, techniques were developed to simulate arbitrary data types. Even though such simulators can depict the behavior of different data types, they lack the ability to showcase the performance differences between them. Therefore, this thesis represents the first prototype of a simulator for Floating Point Numbers, which maintains the relative performance differences implemented in basic C++. Unfortunately, the resulting time dependencies, especially for larger data types, tend to be biased towards longer execution times compared to those produced by a typical CPU. This is due to, for instance, *multiplication* and *division* having a quadratic *time complexity* inside the simulator, which is not the case inside a normal CPU due to optimizations that are not possible to implement efficiently in software. To explore ways to overcome this problem, a straightforward mathematical solution was introduced, which showed promising results.

Kurzfassung

Ein Kerngebiet bei der Entwicklung von Algorithmen, welche Variablen mit verschiedenen Genauigkeiten verwenden, ist es, die Entscheidung zu treffen, wie präzise diese sein sollen. Dabei besteht eine übliche Schwierigkeit darin, dass die Verwendung von Datentypen mit höherer Genauigkeit die Berechnungen zwar präziser machen, jedoch gleichzeitig mehr Rechenzeit erfordert. Weiters sind auf aktueller Hardware in der Regel nur eine sehr begrenzte Anzahl verschiedener Datentypen vorhanden, was Analysen zusätzlich erschwert. Aus diesem Grund wurden Techniken entwickelt, um beliebige Datenformate zu simulieren. Trotz der Tatsache, dass solche Simulatoren das Verhalten verschiedener Typen gut darstellen können, fehlt ihnen die Fähigkeit, die relativen Leistungsunterschiede zwischen diesen aufzuzeigen. Daher stellt diese Arbeit einen ersten Prototyp eines in einfachem C++ programmierten Simulators für Gleitkommazahlen vor, welcher versucht, diese relativen Unterschiede beizubehalten. Leider sind die resultierenden Zeitabhängigkeiten, insbesondere für größere Datentypen, im Vergleich zu den Ergebnissen einer typischen CPU zulasten größerer Zahlen verzerrt. Dies liegt daran, dass beispielsweise *Multiplikation* und *Division* im Simulator eine quadratische Zeitkomplexität haben. Etwas, was bei einer normalen CPU aufgrund von Optimierungen, welche sich typischerweise in Software nicht effizient implementieren lassen, nicht der Fall ist. Um Möglichkeiten aufzuzeigen dieses Problem zu überwinden, wurde eine einfache mathematische Lösung erprobt, welche vielversprechende Ergebnisse zeigt.

Contents

List of Tables

List of Figures

List of Algorithms

Listings

1. Introduction

Floating Point Numbers or short FPNs can be found on almost every computer, and by using them, it is possible to decide how much precision one wants to use for computational processes. It is similar to performing a complicated multistage calculation with pen and paper, where it is impossible to memorize all the sub-results in the head after some point, and one will inevitably write some intermediate results down. Some of those written-down numbers likely have large fractional parts, and by saving the number on paper, one obvious question emerges: How many numbers after the decimal point should be written down?

Of course, having more decimal points will lead to a more exact solution. However, it will also increase the time needed for follow-up calculation. Furthermore, more and more space will be occupied on the paper if a growing portion of the numbers is being saved. Last but not least, if, in the end, only a small precision is needed, all the work done is in vain. Similar problems occur on computers and can be solved by using different representations of an FPN. However, the question of how much precision is actually needed must still be answered. Unfortunately, choosing a smooth transition from one precision to another, as can be done with pen and paper, is impossible for most computers.

The most common Floating Point (FP) precisions are *single* (or also called *float*) and *double*. Both of these FPNs are using the same format, namely the *IEEE 754 standard* [57], but they differ in the number of used bits. For instance, the *single* format uses 32 bits, whereas the *double* format works with 64 bits. Sometimes, another type is also implemented, namely *half* precision, which uses 16 bits. Some modern hardware even supports types, like BF16 and TF32, which are supposed to accelerate computational tasks further. [36]. Nevertheless, these data types are somewhat arbitrary or optimized for specific tasks like BF16 and TF32, which are supposed to optimize deep-learning applications [36]. Still, for other applications, different data types might be more advantageous. Therefore, there is a need to determine which precisions are optimal for different algorithms and applications.

On the one hand, this could be done by implementing all the different FP formats directly at the hardware level on a chip. Nowadays, this could even be done without producing numerous different chips by using *FPGA*s. However, this would still take a lot of time and resources. Another option would be to simulate different precisions in software, and there are already methods to do so [6, 30, 55, 70, 85]. Despite already existing libraries, like *MPFR*[39] and *mpmath* [95], such approaches are not optimal for deciding which precision to use because they do not maintain the relative performance differences of different FP formats.

For instance, when computing an *addition* using two numbers saved in *double* precision, naturally, a better accuracy will be achieved than when the two numbers are stored in *single* precision. Nevertheless, the computation of the latter will be much faster since the numbers only use half the bits. Therefore, it seems logical that when simulating a complex system, one would like to account for the difference in computation time. Consequently, this thesis explores whether it is possible to simulate the behavior of different FPNs by attempting to write such a simulator in C++ and examine its behavior.

1.1. Problem Statement

Choosing the optimal FP precision for computations is essential for writing an efficient computer algorithm. Different FP formats, like *half*, *single*, and *double*, offer various trade-offs in terms of precision and computational speed. Unfortunately, only a very limited set of FPNs are available on typical hardware. Implementing other FP formats in hardware is resource-intensive, and current software alternatives fail to maintain relative performance differences. Therefore, this thesis aims to develop an efficient C++ simulator that emulates arbitrary FPNs, accounting for both accuracy and computational time differences, and examine possible applications of such a simulator. In this sense, the main contributions of this thesis are:

- Implementation of a prototype of an efficient simulator for arbitrary *IEEE 754 standard* [57] conform FPNs, which depicts the relative differences in computation time of different FP formats.
- Analyses of the simulator's behavior and evaluation of its use cases.

1.2. Synopsis

In Chapter 2, important theoretical concepts of this thesis will be introduced. The mathematics of the FP format will be explained, and the subsequent implications for the system will be analyzed. Afterward, the method for *iterative refinement* will be explained in detail since this algorithm will later be used to examine the capabilities of the simulator.

After acquiring the necessary theoretical background, the thesis continues with a thorough analysis of the current state of the art in Chapter 3. First, an overview is given about *mixed precision* and *multiprecision* algorithms and their importance. This is followed by a literature review about how FPNs can be simulated. It will be shown that simulating techniques do exist; however, they are limited since they are either very time expensive (hardware approaches) or lack the ability to emulate correct time relations between data types (software approaches). Lastly, literature providing time measurements of various FP formats is presented for later comparison with the proposed simulator.

Following this review, the general concept of the implementation is introduced in Chapter 4. This includes presenting the algorithms used for writing the operators ($+$, $-$, $*$, $/$) and comparators ($<$, $>$, $\leq$, $\geq$, $==$, $!=$). Besides that, it is also shown how it was ensured that the simulator performs according to *IEEE 754 standard* [57] using statistical methods and how random sparse matrices are constructed within this thesis. Next, a problem is introduced that, for instance, *addition* has a different *time complexity* than *multiplication* when simulated with the proposed simulator, which is not the case in typical CPUs due to parallelization [89]. To address this problem, a basic mathematical model is suggested that primarily involves transforming the results with a factor based on the square root. This approach does not attempt to replicate a CPU exactly but merely attempts to explore possibilities to approach such a problem.

In Chapter 5, the experimental results are presented. Since *single* and *double* are available on most computers, the performance of those two types can be compared to the performance when simulating them with the simulator. For this comparison, *iterative refinement* is not used directly, but important parts of it like Matrix Vector Multiplication (MVM). This is because the analysis can be simplified this way. The vanilla results of the simulator and the transformed results, using the mathematical model, are compared to the native implementation of *single* and *double* on the CPU. It will become clear that the vanilla simulator overestimates the potential speedups when comparing it to the results obtained from a CPU due to the earlier mentioned problem of different time complexities. However, the results also show that the proposed mathematical transformation can mitigate this behavior. After the comparison, the simulator's capabilities will be demonstrated using the whole *iterative refinement* algorithm.

Results and findings will be discussed in Chapter 6 and 7. The downsides of simulating FPNs will be addressed, and potential use cases will be offered. Furthermore, ideas for future work, like improvements or alternations of the simulator, are presented. This chapter will also discuss whether the vanilla simulator's results may also be worth studying, as they might offer an unaltered view of performance, unlike experiments on a CPU where parallelization affects the outcome. Following this chapter, the last one concludes the whole thesis and summarizes it.

1.3. Remarks on Notation

Since notation can sometimes be ambiguous, please find some of the notations used below.

- M_{ij} ..... the element present in the i-th row and j-th column of the matrix$M$
- x_i ..... the i-th element of the vector x
- $x^{(i)}$ ..... the value of the vector x at iteration i
- $\|M\|$ ..... elementwise absolute value applied on the matrix M

2. Theoretical Foundations

2.1. Floating Point Arithmetics

On a finite machine like a computer, a bounded precision for variables is inescapable, and different types of precisions have existed throughout the history of computer science [54]. According to Higham and Mary [54], a landmark was the Fortran 66 standard, which supported *real* (or also called *single*) and *double* precision. Thereby making it possible to write programs using two different sizes of FPNs.

In mathematical terms, an FPN is represented as follows [51]:

$$y = \pm m \times \beta^{e-t} \tag{2.1}$$

The representable system $\boldsymbol{F} \subset \mathbb{R}$ is a subset of the real numbers and is defined by:

- the base β, which is as typically for a computer, 2 throughout this thesis,
- the *precision* t,
- the *exponent* range $e_{min} \leq e \leq e_{max}$.

Generally speaking, therefore, an FPN consists of three bit-arrays; however, the first one, the *sign*, only has one boolean element. If it is *false*, the number is positive and negative if it is *true*. The other two are the *exponent* e and the *mantissa* m. Those are variable in length, and their size is normally stated in available bits. The size of the *mantissa* is also called *precision* t in the context of Equation 2.1. However, *preision* does not always refer to t but sometimes also to the *unit roundoff* u. Both are measures for the accuracy of an FP format but approach the problem from different sides. While t describes the size of the *mantissa*, u describes the spacing between 1.0 and the next larger possible FPN and Higham [51] defines is as follows:

$$u = \frac{1}{2}\beta^{1-t} \tag{2.2}$$

Please also note that since t is the size of the *mantissa* in Equation 2.1, the variable m, in a way, represents digits after the decimal point.

As a consequence of Equation 2.1, an FP format can be defined as $\boldsymbol{F} = \boldsymbol{F}(\beta, t, e_{min}, e_{max})$. Nevertheless, since the *IEEE 754 standard* [57] defines how to calculate e_{min} and e_{max} for a given *exponent* size and typically $\beta = 2$ [54] since computers naturally save data in

Table 2.1.: Characteristics of typical FP systems [51, 54, 55]. Here, the size is given in bits, and the precision accounts for the bits usable for the *mantissa* plus one additional bit for the *sign*. u is the *unit roundoff* and x_{min} and x_{max} the minimal and maximal values that can be represented.

type	size	precision	exponent	u	x_{min}	x_{max}
bfloat16	16	7+1	8	3.91×10^{-3}	1.18×10^{-38}	3.29×10^{38}
half	16	10+1	5	4.88×10^{-4}	6.10×10^{-5}	6.55×10^{4}
single	32	23+1	8	5.96×10^{-8}	1.18×10^{-38}	3.40×10^{38}
double	64	52+1	11	1.11×10^{-16}	2.22×10^{-308}	1.80×10^{308}
fp128	128	113+1	13	9.63×10^{-35}	3.36×10^{-4932}	1.19×10^{4932}

binary format. $\boldsymbol{F}$ in that case can be defined a bit shorter as $\boldsymbol{F} = \boldsymbol{F}(t, e_{size})$.

After the FP format is fixed, it is possible to represent different numbers by changing the values for m and e. Additionally, there is an extra changeable bit, typically 0 when the FPN is positive and 1 otherwise. During this thesis, such a *sign* bit is always present. Commonly, t and e_{size} are defined by the number of usable bits. So, for instance a *single* is using 32 bits, divided into one *sign* bit, 8 *exponent* bits and 23 bits for the *mantissa* (or *fraction*), which is represented graphically in Figure 2.1.

Figure 2.1.: Graphical representation of the *single* format.

The FP formats for *single* and *double* are the most common FP formats. However, in 2008, the *IEEE 754 standard* partly defined a *half precision* format. Despite incomplete definitions, some manufacturers still implemented a version of the format [55]. A few FP formats of interest are depicted in Table 2.1. There are also even lower precision formats like *quarter precision* (8 bit) but those are not standardized [55].

The *IEEE 754 standard* [57] moreover defines special values, like NaN (Not a Number) and *infinity*. To be able to represent those, certain bit patterns are reserved and therefore not available for normal FP usage. Although not all patterns are used, all bit patterns with an *exponent* consisting purely of bits with the value 1 are reserved for special values. Therefore Equation 2.1 becomes:

$$y = \pm m \times \beta^{e-t+1} \tag{2.3}$$

Furthermore, in the normal case, FPNs are normalized because the FP format defined so far would not be unique. Therefore, it is required by the standard to maximize m (by simultaneously decreasing e) as much as possible [57]. This clearly leads to the result that the first digit of m is a *one*, and because the definition is as such, only digits after the decimal point are saved in the *mantissa*.

It is, however, possible to extend the system of normalized FP numbers by so-called *subnormal numbers* [51]. Those do not follow the normalized format explained above and can, therefore, represent even smaller values. So, for instance, the smallest representable normalized FPN within a certain format is $\lambda = \beta^{e_{min}-1}$, but the smallest subnormal number is $\mu = \beta^{e_{min}-t}$. So, they fill the gap between λ and zero [51]. Since subnormal numbers obviously need different treatment on a processor, they can introduce a massive loss in performance, so subnormal numbers are often just rounded towards 0 [4, 33, 105].

2.1.1. Accuracy of Floating Point Numbers

Due to being finite in length, FPNs do not represent all values in perfect accuracy. In other words, there is the possibility of the FPN having an error. According to Higham the standard model in which $x, y \in \boldsymbol{F}$ is in this case as follows:

$$fl(x\ op\ y) = (x\ op\ y)(1+\delta) \tag{2.4}$$

A similar definition can also be found in [43] and [54]. In this mathematical representation, the operator $op \in \{x, -, \cdot, /\}$ and $|\delta| \leq u$ where u is the *unit roundoff* which is a measure for the accuracy of a given FP format $\boldsymbol{F}$. As previously mentioned, the *unit roundoff* is defined as the distance between 1.0 and the next largest possible FPN [51] given $\boldsymbol{F}$ and is also represented in Table 2.1.

Obviously, the lower the *unit roundoff*, the better the precision of the FP format $\boldsymbol{F}$. In literature like [1] and [54], the precision is often stated in terms of u. With that in mind it is indeed noteworthy that having a higher precision format $\boldsymbol{F}(20, 5)$ with *unit roundoff* u_r and a lower precision format $\boldsymbol{F}(10, 5)$ with *unit roundoff* u_l the relationship is as follows: $u_r \leq u_l$, which might be counterintuitive at first glance.

2.1.2. Mixed and Multiprecision Algorithms

In their survey Higham and Mary [54] use the following terminology when talking about *mixed precision* and *multiprecision*:

- A *mixed precision* algorithm uses two or more precisions chosen from a small number of available precisions, typically half, single, and double

precision, provided in hardware, and quadruple precision, provided in software.

- A *multiprecision* algorithm uses one or more arbitrary precisions, which may be problem-dependent and are provided in software. The precision at which results are returned may be fixed (for example, *double* precision) or a parameter. [54, p. 349]

Since these definitions seem reasonable, they will be used throughout this thesis. It is also worth noting that according to this definition, the set of all *mixed precision* algorithms is a subset of all *multiprecision* algorithms when ignoring the type of implementation.

2.2. Solving a Linear System

In order to see the use cases of a simulator as proposed in this thesis, a model problem is required to be able to examine its behavior. There are many options for such a model problem since using different precisions for FPNs is important for a very broad field of algorithms, as will also be shown in Chapter 3. However, one consistently important topic in Computer Science seems exceptionally appropriate, namely efficiently solving a linear system.

Given a non-singular $A \in \mathbb{R}^{n \times n}$ and $x, b \in \mathbb{R}^n$ a linear system, describing a linear set of equations, can be defined as follows [51]:

$$Ax = b \tag{2.5}$$

Typically, A and b are given, and the x is the vector that should be computed. A first thought might be to compute A^{-1}, rewrite Equation 2.5 and compute x by using $x = A^{-1}b$. Although this would work in theory, it is computationally inefficient. It should not be done, and Higham even writes in this book [51, p. 260] that "To most numerical analysts, matrix inversion is a sin." This bad performance is because, for most problems, the inverse is not needed [38]. To further emphasize that inverting a matrix is by no means a practical option, consider the linear system $Ax = b$, where one is normally only interested in finding x. Approaching this by calculating the inverse A^{-1} by, for instance, Gaussian Elimination and partial pivoting has a *time complexity* of $\mathcal{O}(2n^3)$ and is numerically unstable since it amplifies small errors. At the same time, applying a procedure directly on the system is far more stable and has a *time complexity* of only $\mathcal{O}(2n^3/3)$ [51]. A workflow of such a direct application on the system would look like this:

1. Split the *system matrix* A into a *lower triangular matrix* matrix L and an *upper triangular matrix* U using LU Factorization with pivoting: $PA = LU$, where P is the *permutation matrix*.
2. Solve for y in $Ly = Pb$ using *forward substitution*.
3. Solve for x in $Ux = y$ using *backward substitution*.

In this case, $A, L, U, P \in \mathbb{R}^{n \times n}$ and $b, y, x \in \mathbb{R}^n$. The algorithms mentioned above are discussed in the following chapters. Furthermore, Higham [51] also states that this approach is numerically more stable than solving it by calculating the inverse A^{-1}. Although Higham [51] refused to give a formal definition of the term *numerical stability* in his book, Jankowski and Woźniakowski [62] define *numerical stability* as follows:

> Let $\alpha = A^{-1}b$ be the exact solution. A method ϕ is numerically stable if it produces a computed solution y such that
>
> $$\|y - \alpha\| \leq c_1 2^{-t} cond(A) \|\alpha\| \tag{2.6}$$
>
> where c_1 is a constant which depends only on the size n, $c_1 = c_1(n)$, 2^{-t} is the relative computer precision and $cond(A) = \|A\| \left\|A^{-1}\right\|$ is the condition number of A with respect to any norm. [62, p. 304]

Please not that 2^{-t} is just another notation for the *unit roundoff* which is defined in Equation 2.2 since

$$u = \frac{1}{2}\beta^{1-t} = \beta^{-t} = 2^{-t}$$

for $\beta = 2$.

2.2.1. PLU Factorization

The *PLU factorization* is an *LU factorization* with partial pivoting and can also be referred to as Gaussian Elimination with partial pivoting [51]. Pivoting makes the algorithm numerically more stable by interchanging rows of the linear system so that $PA = LU$ instead of $A = LU$ where $P \in \mathbb{R}^{n \times n}$ is the *permutation matrix* which records all row interchanges [51]. The increase in stability is gained because the result of the interchanging is divided by larger values during the algorithm's execution. The algorithm used for the *PLU factorization* in this thesis is given in Algorithm 1. Such an implementation can also be found in [106].

Data: $A \in \mathbb{R}^{n \times n}, b \in \mathbb{R}^n$
Result: $PA = LU$, where $P, L, U \in \mathbb{R}^{n \times n}$
Set up $P = I, L = I$ and $U = A$
for $k = 0, 1, ..., n-1$ **do**

Find $i \in \mathbb{N}_0 = \underset{k \leq i < n}{\arg\max} (|U_{ik}|)$
Interchange U_{ir} and U_{kr} for $r = k, k+1, ..., n-1$
Interchange L_{ir} and L_{kr} for $r = 0, 1, ..., k-1$
Interchange P_{ir} and P_{kr} for $r = 0, 1, ..., n-1$

for $j = k+1, k+1, ..., n-1$ **do**
$L_{jk} = \frac{U_{jk}}{U_{kk}}$
for $i = k+1, k+1, ..., n-1$ **do**
$U_{ji} = U_{ji} - L_{jk} \cdot U_{ki}$
end
end
end

Algorithm 1: PLU Factorization

2.2.2. Forward and Backward Substitution

The algorithm for *forward substitution* is very similar to that for *backward substitution*. The main difference is the indexing since the *system matrix* used for *forward substitution* is a *lower triangular matrix* and the one used for *backward substitution* is an *upper triangular matrix*. The algorithm for *forward substitution* is given in Algorithm 2. When using Algorithm 2 with an *upper triangular matrix* and adapting the indexing, one gets the algorithm for *backward substitution*. Again, a similar implementation can be found in [106].

Data: Lower triangular matrix $L \in \mathbb{R}^{n \times n}, b \in \mathbb{R}^n$
Result: Solution $x \in \mathbb{R}$ for the system $Lx = b$
for $i = 1, 2, ..., n-1$ // $i = n-2, n-3, ..., 0$ `for back. sub.`
do
$x_i = b_i$
for $j = i+1, i+2, ..., n-1$ // $j = n-1, n-2, ..., i+1$ `for back. sub.`
do
$x_i = x_i - L_{ij} \cdot x_j$
end
$x_i = \frac{x_i}{L_{ii}}$
end

Algorithm 2: Forward Substitution

2.2.3. Iterative Refinement

The method of *iterative refinement* is a way to solve a linear system by exploiting the use of *mixed precision* or *multiprecision*. When thinking about workflow by first performing a *PLU factorization* folowed by *forward substitution* and *backward substitution* it is easy to see that the *PLU factorization* has the largest *time complexity*, namely $\mathcal{O}(n^3)$ for the *PLU factorization* compared to $\mathcal{O}(n^2)$ for *forward substitution* and *backward substitution* [54]. The idea is now to calculate the *PLU factorization* in a lower precision, thereby saving computational cost. This procedure will, however, introduce an additional error to the solution, and to account for this, additional refinement steps are added after the *PLU factorization* to reduce that error.

However, before stating the complete algorithm, first, a general version of *iterative refinement* is presented in Algorithm 3. An almost identical algorithm can be found in [18, 54]. Please note that $x^{(0)}$ must not necessarily be the zero vector $\overrightarrow{0}$. Nevertheless, it is typically chosen as such if there is no better initial guess [83].

Data: A non-singular matrix $A \in \mathbb{R}^{n\times n}, b \in \mathbb{R}^n$ and three precisions $u_r \leq u \leq u_l$ (in terms of *unit roundoff*)
Result: Approximate solution $\hat{x} \in \mathbb{R}^n$
$x^{(0)} = \overrightarrow{0}$
for $i = 0 : i_{max}$ *or until converged* **do**
  Compute $r^{(i)} = b - Ax^{(i)}$ in precision u_r .
  Solve $Ad^{(i)} = r^{(i)}$ using precision u_l.
  Update $x^{(i+1)} = x^{(i)} + d^{(i)}$ using precision u.
end

Algorithm 3: General implementation using *iterative refinement*

This very easy implementation of Algorithm 3 does not state which method to use to solve the linear system $Ad^{(i)} = r^{(i)}$. Regardless of how this problem is approached exactly, the relative error $\xi^{(i)}$ when solving the system $Ad^{(i)} = r^{(i)}$ numerically can be calculated according to Equation 2.7, where $d^{(i)}$ is the correct solution and $\hat{d^{(i)}}$ the numerical approximation.

$$\xi^{(i)} = \frac{\left\| d^{(i)} - \hat{d^{(i)}} \right\|_\infty}{\left\| d^{(i)} \right\|_\infty} \tag{2.7}$$

Regarding Algorithm 3 Carson and Higham [18] find the following Theorem in 2018 about the conversion of *iterative refinement* in three precisions. This Theorem is also included in survey [54].

Theorem 2.1. *Let Algorithm 3 be applied with any $x^{(0)}$ to a linear system $Ax = b$, where $A \in \mathbb{R}^{n\times n}$ is non-singular and $x, b \in \mathbb{R}^n$. As long as $\phi^{(i)}$ is sufficiently less than 1, the forward error is reduced on the i-th iteration by a factor of approximately $\phi^{(i)}$ until an iterate $\hat{x}$ is produced for which*

$$\frac{\|\hat{x} - x\|_\infty}{\|x\|_\infty} \lesssim u + 4p\, cond(A, x)u_r \tag{2.8}$$

where

$$\mu^{(i)} = \frac{\left\|A(x^{(i)} - \hat{x}^{(i)})\right\|_\infty}{\|A\|_\infty \left\|x^{(0)} - \hat{x}^{(i)}\right\|_\infty} \leq 1 \tag{2.9}$$

$$\phi^{(i)} = 2\, min(cond(A), \kappa_\infty(A)\mu^{(i)})u_l \; + \; \xi^{(i)} \tag{2.10}$$

and p is the maximum number of non-zeros in any row of [A b].

[54, p. 369]

Theorem 3 shows that the limiting accuracy only depends on the precisions u_r and u but not on u_l, x_0, or on how the linear system is solved as long as $\xi_i \ll 1$ [54]. Higham and Mary also state in their survey [54] that a limiting accuracy of u is obtained by setting $u_r = u^2$, as long as $cond(A, x)u \leq 1$, which is a very interesting result because when wanting to solve a linear system in *working precision* u, the accuracy of a probably computationally very heavy part of the algorithm can be reduced by still being able to acquire precision u. Further, it is important to note that since ϕ_i only depends on precision u_l, the rate of convergence is only dependent on the effective precision u_l of the solver [18].

As mentioned at the beginning of the chapter, *PLU factorization* can be used as a solver. Since the subject of this thesis is to write a simulator, *iterative refinement* is solely used to have an algorithm to test the simulator on. For this purpose, *PLU factorization* followed by *forward substitution* and *backward substitution* is sufficient as a solver. Nonetheless, there are also other options like *GMRES*. A method which is used by [17, 83] and is also mentioned in the survey of Higham and Mary [54].

Algorithm 4 depicts the version of *iterative refinement* which will be used for this thesis. As discussed above, *PLU factorization* is used to solve the system. Furthermore, Algorithm 4 also improves the initial guess x_0 by calculating a first solution using *forward substitution* and *backward substitution* right before the while-loop.

Data: A non-singular matrix $A \in \mathbb{R}^{n \times n}, b \in \mathbb{R}^n$ and three precisions $u_r \leq u \leq u_l$ (in terms of *unit roundoff*)
Result: Approximate solution $x \in \mathbb{R}^n$
$x^{(0)} = \overrightarrow{0}$
Compute the factorization $PA = LU$ using precision u_l.
Solve $LUx_0 = Pb$ by *substitution* using precision u_l.
for $i = 0 : i_{max}$ *or until converged* **do**
  Compute $r^{(i)} = b - Ax^{(i)}$ in precision u_r .
  Solve $LUd^{(i)} = r^{(i)}$ by *substitution* using precision u_l.
  Update $x^{(i+1)} = x^{(i)} + d^{(i)}$ using precision u.
end

Algorithm 4: *Iterative refinement* using *PLU factorization*

At first glance, this might look disadvantageous compared to the steps presented at the beginning of 2.2 because all the steps are done analogously, followed by a for-loop. However, the computationally most expensive part, namely the *PLU factorization* with *time complexity* $\mathcal{O}(n^3)$ which is done at the beginning, is, as can be seen, done in the lower precision u_l. Subsequently, the computation is accelerated. The lost precision is then regained later by the refinement loop with each command only having *time complexity* $\mathcal{O}(n^2)$ or less.

Higham and Mary state in their survey [54] that solving a linear system by using LU *factorization* with precision u_l approximately achieves an error bound of $3n\||A^{-1}|\hat{L}||\hat{U}|\|u_l$. Using Theorem 2.1 we get the following new Theorem as also stated in [54].

Theorem 2.2. *Let Algorithm 4 be applied to a linear system $Ax = b$, where $A \in \mathbb{R}^{n \times n}$ is non-singular and $x, b \in \mathbb{R}^n$. If $\||A^{-1}||\hat{L}||\hat{U}|\|u_l$ is sufficiently less than* 1*, then the algorithm produces an iterate $\hat{x}$ satisfying Equation 2.8.*

[54, p. 372]

2.3. Analyzing Speedups

As mentioned earlier, different algorithms can be used with different precisions. For instance, a matrix-vector product can be computed in both *single* and *double* precision. Surely, these are not the only possible options theoretically. However, numerous hardware only supports a limited amount of Floating Point (FP) formats. Following this, it is often useful to calculate a factor between the time needed to use an algorithm with *double* precision and the same algorithm used with *single* precision. Some factors are reoccurring and are therefore defined in the following equations.

$$F_{GEMV} = \frac{SGEMV}{DGEMV} \tag{2.11}$$

$$F_{GEMM} = \frac{SGEMV}{DGEMM} \tag{2.12}$$

$$F_{GETRF} = \frac{SGETRF}{DGETRF} \tag{2.13}$$

$$F_{IR_H} = \frac{DGESV}{IR_{(u_H,u_D,u_D)}} \tag{2.14}$$

$$F_{IR_S} = \frac{DGESV}{IR_{(u_S,u_D,u_D)}} \tag{2.15}$$

The used abbreviations can be found in Table 2.2. Please note that the abbreviations in this context refer to the **runtime** of the referred algorithm. Similar definitions can be found in [65]. Please further note that in the context of this thesis, additionally, the abbreviations D is used for *double*, S is used for *single*, and H is used for *half*. Correspondingly, u_S, u_S, and u_D are the *unit roundoff*s of *half*, *single*, and *double*.

Table 2.2.: Used abbreviations with their description, where $f_1, f_2, f_3 \in \boldsymbol{F}$ are FP formats.

Abbreviation	Description
(S.D)GEMV	Matrix Vector Multiplication in (single/double) precision.
(S,D)GEMM	Matrix Matrix Multiplication in (single/double) precision.
(S,D)GETRF	LU factorization in (single/double) precision.
(S,D)GETRS	Forward and backward substitution in (single/double) precision.
(S,D)GESV	Solve linear system ((S,D)GETRF + (S,D)GETRS)
$IR_{(f_1,f_2,f_3)}$	Iterative Refinement with: $f_1 = u_l$, $f_2 = u$ and $f_3 = u_r$

3. Literature Review

Algorithms exploiting the advantages of *mixed precision* are getting increasingly important, even though some modern high-level programming languages like *Python* natively do not require the user to select any specific data type for variables. Especially nowadays, where the use of AI has skyrocketed, performing tasks like training a *neural network*s as efficiently as possible is significant. Amongst other things, this motivated *Intel* to propose the *bfloat16* format in 2018, which should be advantageous for *deep learning* [58]. Work has been done by [15, 25, 26, 32, 36, 77, 80, 100, 102, 103] and many more where it is shown that it is sufficient to use very low precision when employing a *neural network*. Such gains in efficiency do not only result in speedups but also in energy efficiency [22, 48].

3.1. Numerical Algorithms

When it comes to numerical algorithms, *mixed precision* approaches are often used in combination with *iterative refinement*, which is explained in Section 2.2.3. For short, using *iterative refinement*, one can calculate a less precise result and refine it afterward to achieve a higher precision. Very early in the year 1967, this procedure was proposed to mitigate roundoff errors of Floating Point Number (FPN). For instance, Björck [10] proposed such an approach for *linear least square* solutions and Moler [78] for linear systems. This work was later extended by Gulliksson [45] in 1994 for constrained and weighted *linear least square* methods. Later in 2009 Demmel et al. [31] discussed a practical implementation accompanied with numerical testing which showed that the algorithm achieved condition-independent accuracy under certain conditions for both the solution and the residual. But not only can the approach of *iterative refinement* be used to achieve more accurate results, but it is also potentially faster than other methods as, for instance, Meijerink and Van Der Vorst [75] found out in 1977 for linear systems where the *system matrix* is a symmetric *M-matrix*.

Nevertheless, *iterative refinement* is not limited by directly improving results computed in lower precisions but can also be used to increase numeric stability. Also, in the year 1977 Jankowski and Woźniakowski [62] published a paper proving that *iterative refinement* with an arbitrary linear solver is stable under certain conditions. Also, Skeel [86] published a paper that *iterative refinement* can be used to increase numeric stability for Gaussian Elimination [54]. This idea was further generalized by Higham in the year 1991 in [52] and in the year 1997 in [53].

By the year 1981, Kiełbasiński [63] investigated the idea to use continuously changing precisions for the refinement procedure to achieve an arbitrary precision in the end result. According to Higham and Mary [54], this idea was first introduced in the context of an exercise by Stewart [88] in 1973 and was later on further examined by Smoktunowicz and Sokolnicka [87] in 1984. Therefore, at least theoretically, *iterative refinement* was no longer bound by the FP precisions available on a computer but was also investigated in the sense of a much broader *multiprecision* stage. Also, around the same time in 1982 Zlatev [109] investigated the use of *iterative refinement* in the context of a linear system with a sparse *system matrix*. Further, the method of *iterative refinement* has also been explored in more specific contexts like the work of Govaerts and Pryce [42] in 1990 where they investigated the topic in the context of a black box solver for *bordered linear system*. In this context, a *bordered linear system* means that the *system matrix* has added rows and columns which hold additional information.

In the year 2001, Newton's Method was investigated by Tisseur [98] in the context of the generalized eigenvalue problem. Again, it was found that *iterative refinement* is useful to improve the method's stability, and in this case, it could be shown that both backward and forward errors of the computed *eigenpair* are improved. Building on this also Davies, Higham and Tisseur [28] showed that Newton's method could be used to produce *eigenpair* with small backward errors. According to Higham and Mary [54], Newton's method is well suited to find a small number of *eigenpair*s but is not feasible to solve a whole eigensystem.

Also, Ogita and Aishima [81] propose an algorithm in 2018 for solving eigenvalue problems using *iterative refinement* and got good results if a modestly good initial guess is provided. Their solution even works with systems not limited by nearly multiple eigenvalues, which means that multiple eigenvectors have the same eigenvalue [90]. The problem with the need for a good initial guess was addressed in the follow-up paper in the year 2019 [82]. Later in 2022 Tsai, Luszczek and Dongarra reinvestigated Newton's method in the context of eigensystem and achieved mentionable speedups.

After 2000, new hardware emerged, and operations done in single precision were much faster than those done in double precision [54, 64]. According to Higham and Mary [54], this motivated Langou et al. [65] to propose a new way to solve linear systems by the usage of *iterative refinement*. A solver was depicted, which is very similar to Algorithm 4 but uses only two precision instead of three, namely one lower precision (*single*) for the *LU factorization*, *forward substitution* and *backward substitution* and one higher precision (*double*) for the computation of the residual. With this method for different architectures, a speedup between 1-2 could be achieved concerning an algorithm that only uses double precision. Also, Buttari et al. [13] analyzed the performance between *single* and *double* precision in 2007 for dense matrices followed by an analysis for sparse matrices by Buttari et al. [14] in 2008.

Later in the year 2018 Haidar et al. [46] made a new analysis of the performance of *iterative refinement*. They did not only use an *iterative refinement* variant that utilized *double* and *single* precision but also employed one that operates with *double* and *half* precision. In addition, they tried to harness GPU tensor cores. With all this, they achieved a speedup of up to 4 compared to a solver, which only uses *double* precision. The same year, Haidar et al. [48] released another paper with a similar analysis specifically emphasizing on energy efficiency. They could achieve an energy reduction of 50% on an Intel Knights Landing (KNL) architecture and an even higher reduction of 80% when using Nvidia V100 PCIe GPUs. Also Lee et al. [69] analyzed the energy efficiency when using lower precision FP numbers and achieved a speedup of between 2.0 and 2.6 and energy savings between a factor of 1.8 and 2.4. Further, in the year 2020 Haidar et al. [47] did another performance analysis where they analyzed many more sparse matrices and got roughly similar results.

In 2020 Iwashita, Suzuki and Fukaya [60] propose an *iterative refinement* not limited to FP formats but introduce the use of integer arithmetic for *GMRES*. They find that this version performs comparably to the standard version, which uses solely FP arithmetic. A year later, in 2021 Amestoy et al. [3] also wrote a paper concerning the *GMRES* and propose a version that uses five different precisions through the algorithmic routine. In 2022 Oktay and Carson [83] depicted an algorithms which combines different versions of *GMRES*. It starts with the least expensive version and switches to a more expensive but reliable version when the convergence slows.

The same year in 2022 Zounon et al. [110] released a paper comparing the performance of *LU factorization* using *single* and *double* on different sparse matrices. It was found that only large matrices reach the theoretical speedup of 2×. The same year, a mixed precision version for the s-step Lanczos and the conjugate gradient method was introduced by Carson, Gergelits and Yamazaki [16]. Furthermore, Tsai, Luszczek and Dongarra [99] depicted a mixed-precision algorithm for finding *eigensolver* for symmetric and Hermitian matrices and achieved a speedup of up to 3.6×. One year later, in 2023 Amestoy et al. [2] released a paper combining *sparse approximate factorizations* with mixed-precision *iterative refinement* and find that they achieve a reduction in both time and space complexity. It is also important to note that overflows can happen more frequently when the number of available bits of the exponent is reduced. This problem arises because, in that case, the exponent part of the FPN cannot represent exponents large enough. Therefore, for instance, Higham, Pranesh and Zounon [56] proposed a method in 2019 to artificially shrink the values of a matrix to be able to be stored using fewer bits for the exponent. This is achieved by introducing a scalar, such that elementwise multiplication of this scalar with the shrunken matrix restores the original matrix.

Lastly, two surveys, one from Higham and Mary [54] and another one from Abdelfattah et al. [1] should be mentioned, because both provide an exceptional introduction to the topic and current developments. Also, Higham wrote a book on the accuracy and stability

of numerical algorithms [51], which does not directly touch the topic of *mixed precision* but is mentioned for the sake of completeness.

3.2. Machine Learning

The use of low precision FP formats is also widely used in *Machine Learning*. For this use case, *bfloat16* [58] was developed since it was found that this data type is sufficient for *neural network*s. The format is preferable because lower precision formats are computationally faster, and the drawbacks are irrelevant in the case of a *neural network* [36]. Also Courbariaux, Bengio and David [25] find in the year 2015 that very low precisions are sufficient to train a *neural network*. For instance, they found that training certain *neural network*s is possible with only a 10-bit multiplication. Furthermore, in the year 2018 Wang et al. [103] published a paper where they showed that they were able to train a *neural network* using only 8-bit data and computation precision and also reduced the arithmetic precision from 32-bit to 16-bit.

Also, in 2018 Das et al. [26] implemented a *neural network* using *mixed precision* integer operations and reported an increase in throughput of 1.8× and still maintaining the target accuracy. The same year, Micikevicius et al. [77] showed that model accuracy could also be maintained by using *half* and *single* precision. Furthermore, Nandakumar et al. [80] proposed a *mixed precision* combination of two memory units where the first one is storing the weights using lower precision, and the second storing the accumulated weight updates. Amongst others, a similar approach is also proposed by Micikevicius et al. [77]. Besides that, *mixed precision* can also be used to quantize a model, which is to reduce the size of the network in order to improve inference speed. Concerning this, two papers were published in 2019 regarding finding good choices for the precision to use for each layer. The first one by Dong et al. [32] proposed an approach using each layer's Hessian spectrum, and the second one from Wang et al. [102], on the other hand, tries to find a method based on the individual hardware specifications and achieving a reduction in latency by a factor of 1.4× to 1.95× and an energy reduction of 1.9×. Also, Cai and Vasconcelos [15] were concerned about the optimal quantization of a network. They proposed a search algorithm with a differentiable search architecture to find an optimal solution and achieved good results by outperforming uniform counterparts. Lastly, also Uhlich et al. [100] published a paper concerned with choosing the right parameters for *mixed precision* DNNs and accomplished state-of-the-art performance.

However, the introduction of lower precision FP formats not only has benefits on the performance but also on the resilience against *soft-error* as Li et al. [71] pointed out in the year 2017. This better behavior is because most severe faults induced by *soft-error* are caused by a bit-flip in the more significant bits of the exponent [84]. Furthermore, the errors are more severe if a *soft-error* flips a bit from 0 to 1 than vice versa [84]. The first is logical since the deviation from the correct value is larger if a bit with higher

significance flips its value. Likewise, the latter seems logical since if a bit flips from 1 to 0, the magnitude of the deviation is less than or equal to the absolute value of the correct value. Because this is not the case when a 0 flips to a 1, the relative error might be much larger.

If more bits are used for operating the system, it seems clear that, therefore, more bits are vulnerable for *soft-error*s. For this case, for instance, Chen [21] designed an algorithm which finds the most endangered bits in $\mathcal{O}(\log n)$ *time complexity*. This information can subsequently be used to additionally secure vulnerable bits and use a less secure procedure for less vulnerable ones. However, since the more vulnerable bits are typically the more significant, one could also think of reducing the precision and omitting most of the non-significant bits.

3.3. Mixed-Precision in other Sciences

Surely *mixed precision* is also used in other fields of science, like, for instance, metrology. In 2014 Düben, McNamara and Palmer [34] proposed a *mixed precision* approach to improve weather and climate predictions. Also, Thornes, Düben and Palmer [96] and Hatfield et al. [50] were using *half* precision in order to improve their models. In the year 2019 Chantry et al. [19] also proposed the uses of less precise data types for weather modeling and stated that larger scales need higher precision than shorter ones regarding weather forecasting time scales. They also find that the expected global mean quantities *half* precision is sufficient for the simulation. Further, also in 2019 Hatfield et al. [49] published a paper where they investigated the usage of *half*, *single* and *double* precision on Nvidia Tensor Cores for weather forecasting. It was found that it is possible to use *half* and *single* precision almost throughout the simulation without affecting the forecast when compared to the usage of *double* precision. Additionally, in 2019 Tintó Prims et al. [97] uses *mixed precision* for ocean simulations and got promising results.

However, *mixed precision* approaches can not only be used for meteorological forecasting but also for *Finite Element Method*s like was done by Göddeke, Strzodka and Turek [41] in the year 2007 using *FPGA*s and GPUs. Nevertheless, at least for GPUs, this approach is slightly slower when compared to solving with *iterative refinement*. Similarly, Das et al. [27] employed a *mixed precision Finite Element Method* and achieved a performance that was unprecedented at the time. Furthermore, in the year 2020 Wang et al. [104] utilized *mixed precision* for a *Finite Element Method* and accomplished a 10% increase in performance. Besides that, parallelization also achieved a speedup of 25×.

Different FP formats can also be used in chemistry and physics. For instance Clark et al. [23] used the approach for Molecular-Dynamics, Chen, Chacón and Barnes [20] for a fully implicit particle-in-cell method and Le Grand, Götz and Walker [66] to solve a lattice QCD system. All their results showed a good performance compared to the

standard double version. Further, Bini and Robol [9] used a *multiprecision* approach to solve secular and polynomial equations and obtained results so good that they even described them as "dramatic". Additionally, McCormick, Benzaken and Tamstorf [74] and Tamstorf, Benzaken and McCormick [92] investigated *mixed precision* in the context of multigrid solvers.

Lastly, *mixed precision* can also be useful in finance. Both Chow et al. [22] and Brugger et al. [12] considered the usage of different precisions for Monte Carlo methods to price *Asian option*s using *FPGA*s. Chow et al. achieve a speedup of a factor or 4.6× and a 5.5× increase in energy efficiency. Brugger et al., on the other hand, even achieved a speedup of between 3× and 9×.

3.4. Simulating Multiprecision Arithmetics

When a *multiprecision* approach is used for any algorithm, the question of which exact precision should be used arises quickly. In that domain, simulation is certainly one option and there are already simulators for arbitrary precision FP arithmetic in existence like *mpmath* [95] for *Python* and *MPFR* [39, 70] for C. *SymPy* [76] also has the ability to vary the accuracy of its FP datatypes, yet uses the *mpmath* library for it. However, for instance, *MPFR* [39] was originally designed to simulate higher precision FP numbers [54], which was necessary because some simulations need a certain accuracy to acquire well enough results. A similar fact is true for *mpmath*. Furthermore, there is also *MPFUN2020* an arbitrary FP simulator form Bailey [6] for C++. Bailey also provides additional software for simulating FPNs on his website [7]. However, besides *MPFUN2020* those are only for simulating high-precision fixed format (*double-quad*, *quad precision*, *double-double*) FPNs. Additionally, there is the *MPLAPACK* library, which however also uses *MPFR* arithmetic to simulate arbitrary precision FP formats, along with some fixed precision data types [79].

Those simulators are made for high-precision simulations, not especially for low-precision ones. Although this does not mean that lower precisions cannot be simulated with these simulators, one would not get the desired speedups because of their implementation, which is optimized for high precision. For example, *MPFR* is using arrays of so-called *limbs* to represent FP data [39]. One limb is, for instance, a 64-bit unsigned integer Lefevre and Zimmermann, and with the stacking of such limbs, it is possible to represent a *mantissa* length of more than 64 bits. However, when calculating a sum of two FPN with a *mantissa* length of less than 64 still, the whole limb is used for the *addition*, which is certainly not optimal. Such a condition is, of course, a problem when analyzing the time efficiency of an algorithm like *iterative refinement*.

Other simulators and simulation techniques for low precision FP arithmetic exist, like [30, 55], but these also do not show performance differences between arbitrary FP formats. So, for instance, computing an addition for two FPNs with $\boldsymbol{F}(20, 9)$ may take the same time as computing the same by using $\boldsymbol{F}(21, 9)$. On the contrary, some of the other

available software solutions only emphasize typical FP implementations like *half*, *single*, and *double*. The latter is the case for software published in 2013 by Rubio-González et al. [85], where they proposed a solution which examines the FP formats inside a program and finds the local minimum where the analyzed program performs best. Unfortunately, the search space only covers *float*, *double*, and *long double*, and therefore, this solution cannot analyze arbitrary FP formats.

Later in 2017 Dawson and Düben [30] describe a reduced-precision emulator written in Fortran. Good results were accomplished, but the program first computes all results in higher precision and later reduces the precision as desired, which, of course, does not maintain any relationship between the performance of different FP formats. They state that this provides a good approximation and avoids the necessity of implementing every function from scratch [30]. Furthermore, Dawson and Düben [30] stated that the simulator is focused on verifying results but not simulating the expected performance gains. An interesting side note is that they mentioned that the Software could also be adapted to simulate bit-flips. A similar approach is used by Higham and Pranesh [55] in the year 2019, where they present a function named *chop* to simulate low precision FP arithmetic and achieved positive results. A similar approach is also provided by the library *CPFloat* [37], but as before, such methods do not maintain the performance relationship between different FP formats.

There are also approaches to implementing lower precision formats in hardware and analyzing their performance from that perspective. However, since this only touches the topic of this thesis, the analysis will be held shortly. One early approach was made by Tan, Danysh and Liebelt [93] in the year 2003, where they implemented a *multiprecision* Vector Multiply-Accumulator for fixed point arithmetic using shared segmentation. Later in the year, 2012 Liang, Lee and Peterson [73] proposed an ALU design for arbitrary fixed- and floating-point precisions but focused on fixed-point adders and multipliers. Three years later, in 2015 Liang published his PhD thesis expending the work by implementing a fully *multiprecision* FP ALU. This ALU, when implemented on an *FPGA*, performs almost as well as a normal ALU but apparently occupies twice as much space.

Another paper to mention was published in the year 2008 by Sun, Peterson and Storaasli [91] where they implemented *iterative refinement* on an *FPGA*. They showed that the *mixed precision* approach significantly increased the performance when solving a linear system. In the year 2011 Lee and Peterson [67] proposed an *iterative refinement* variant with arbitrary precisions on an *FPGA* and showed that they can satisfactorily achieve results in *single* and *double* precision. Later in the year 2020 Lee et al. [68] proposed an improvement with their new *AIR* algorithm. With this, they also extend the work of Lee et al. [69] by using arbitrary precisions.

3.5. Performance of mixed-precision algorithms

Since it is later important for evaluating the proposed *multiprecision* simulator in this section, papers will be summarised, examining the performance of algorithms using different precision formats. Please note that some abbreviations used in this section are defined in chapter 2.3. The first to mention is the work form Langou et al. [65] from the year 2006. In their research, they evaluated the performance of algorithms using *single* and *double* precision or, in the case of *iterative refinement*, a combination of both. The results of their experiments, for which dense matrices were used, are depicted in Table 3.1. As can be seen, the speedups for *matrix-matrix multiplication* are between 2.12× and 0.99×. However, most are above 1.5×. In the case of *LU factorization*, the speedups range between 2.24× and 1.08×, but as seen before, most are above 1.5×. In the case of *iterative refinement*, the speedups are slightly less, ranging from 1.92× to 0.91×. This time, not most values are above 1.5, but exactly half of them.

Table 3.1.: The speedup factors between using *single* and *double* for different algorithms on different architectures as taken from [65].

Architecture	n	F_{GEMM}	F_{GETRF}	F_{IR_S}
Intel Pentium III Coppermine (Goto)	3,500	2.10	2.24	1.92
Intel Pentium III Katmai (Goto)	3,000	2.12	2.11	1.79
Sun UltraSPARC IIe (Sunperf)	3,000	1.45	1.79	1.58
Intel Pentium IV Prescott (Goto)	4,000	2.00	1.86	1.57
Intel Pentium IV-M Northwood (Goto)	4,000	2.02	1.98	1.54
AMD Opteron (Goto)	4,000	1.98	1.93	1.53
Cray X1 (libsci)	4,000	1.68	1.54	1.38
IBM Power PC G5 (VecLib)	5,000	2.29	2.05	1.24
Compaq Alpha EV6 (CXML)	3,000	0.99	1.08	1.01
IBM SP Power3 (ESSL)	3,000	1.03	1.13	1.00
SGI Octane (ATLAS)	2,000	1.08	1.13	0.91

Also, [14] depicts speedups of *matrix-matrix multiplication* in addition to factors for *matrix-vector multiplication*. Their results, which look quite similar to the results of [65], are shown in Table 3.2. This time, the range for matrix multiplication is a bit shorter, and all results lie between 1.44× and 2.21×, but as before, most values are above 1.5×. Furthermore, [14] examined the speedups for *matrix-vector multiplication* with values ranging from 1.64× to 2.21×.

Table 3.2.: The speedup factors between using *single* and *double* for standard matrix operations on different architectures as taken from [14].

Architecture	n	F_{GEMV}	n	F_{GEMM}
AMD Opteron 246	3,000	2.00	5000	1.70
Sun UltraSparc-IIe	3,000	1.64	5000	1.66
Intel PIII Copp.	3,000	2.03	5000	2.09
PowerPC 970	3,000	2.04	5000	1.44
Intel Woodcrest	3,000	1.81	5000	2.18
Intel XEON	3,000	2.04	5000	1.82
Intel Centrino Duo	3,000	2.71	5000	2.21

Another paper to mention was published in 2018 by Haidar et al. [46] in which they examined the computation times needed to solve matrices taken from the Sparse Matrix Collection of the University of Florida [29] using difference configurations of *iterative refinement* on Nvidia V100 GPUs. They only state the computation times, but the corresponding factors can be easily deduced and are shown in Table 3.3. It can be seen that all values range between a speedup of 1.59× and 1.95× when using *iterative refinement* with *single* precision in the refinement step. Such values seem to be in line with the results that were previously examined. The values tend to increase when switching from *single* to *half* precision. The only exception is the computation for the matrix *em192*. However, in the original table in [46], it can be seen that in that case, considerably more iterations are needed in the refinement step, which probably means that problems concerning the convergence were present in that case.

Table 3.3.: The speedup factors between using *iterative refinement* compared to solving it directly by using *PLU factorization*. These results are taken from [46].

Matrix Name	n	F_{IR_S}	F_{IR_H}
em192	26,896	1.83	1.09
appu	14,000	1.59	1.79
ns3Da	20,414	1.62	2.07
nd6k	18,000	1.80	2.25
nd12k	36,000	1.95	3.05

Haidar et al. [47] did another paper in 2020 where again the performance of *iterative refinement* was evaluated, where this time, Nvidia Quadro GV100 (Volta) GPUs were used. When comparing the results with those obtained in 2018, they are slightly lower but still quite similar. Most of the results are above 1.5×. However, none of the results are above a factor of 2×. The results are shown in Table 3.4. Again, the Sparse Matrix Collection of the University of Florida [29] was used. Furthermore, many more matrices were examined, but no speedups were given for *iterative refinement*, which uses *half*

precision for the refinement *LU factorization* without the utilization of tensor cores.

Table 3.4.: The speedup factors between using *iterative refinement* with *single* and *double* precision and solving it directly using *PLU factorization* with *double* precision. The results shown are taken from [47].

matrix name	n	sp	matrix name	n	sp
crystk03	24,696	1.68	nd12k	36,000	1.83
crystm03	24,696	1.68	epb1	14,734	1.47
TEM27623	27,623	1.74	appu	14,000	1.42
mixtank new	29,957	1.71	bcsstk25	15,439	1.51
e40r0100	17,281	1.52	bcsstk37	25,503	1.69
sme3Db	29,067	1.59	raefsky3	21,200	1.62
Goodwin 054	32,510	1.78	thread	29,736	1.78
ns3Da	20,414	1.59	mult dcop 01	25,187	1.64
Thermal1	17,880	1.54	ramage02	16,830	1.31
Zhao1	33,861	1.83	Poisson	32,000	1.81
nd6k	18,000	1.56	Vlasov	22,000	1.78

Lastly, also Zounon et al. [110] did a large analysis of the performance of *iterative refinement* using different precision for *LU factorization* in the year 2022. The obtained results roughly resemble the results that were previously discussed. Because of this and the truly massive amount of generated results, their findings will not be shown inside a table like the others. It is interesting to note that Zounon et al. states that the speed when performing *LU factorization* on dense matrices one should experience a speedup of about 2×. For sparse matrices, they find that only very large matrices experience a speedup close to that value. Also, they found that often, the performance of *LU factorization* in *single* precision is reduced compared to *double* precision because more subnormal numbers are created. They overcame this problem by flushing all values, which would be subnormal to zero, and subsequently got the expected speedups.

3.6. Positioning of this Thesis

As shown above, *mixed precision* approaches are very useful for numerous numerical problems like solving a linear system with *iterative refinement* or for implementing novel *eigensolver*s [1, 54]. Furthermore, *mixed precision* is a very important topic in *Machine Learning* [103] and other sciences like metrology [49, 97]. Therefore, it seems reasonable to claim that research in this field is of great importance. Besides developing novel algorithms that utilize the existence of different precisions, it is also very essential to choose optimal precisions [15, 32, 100, 102]. A general approach is still missing but is already being addressed in fields like *Machine Learning*. A solution could be accomplished by simulating *multiprecision* numbers. There are already papers proposing ways to simulate

multiprecision data types [6, 30, 55, 70, 85] and even libraries like *MPFR* [39] and *mpmath* [95]. Still, they do not maintain the relative performance benefits from various data types, as was argued earlier.

The fact that the relative speed is not maintained is a problem because, for instance, in the context of *iterative refinement*, an algorithm operating fully in *double* precision will normally converge in several steps less than or equal to the number of steps needed by an algorithm using lower precisions. However, the latter could, in practice, still outperform the first since the computational cost for performing operations with lower precision is smaller. It seems important to close this gap, and therefore, in this thesis, a simulator is proposed to simulate *multiprecision* numbers and maintain the relative performance properties of various FP formats using basic C++. To the best of the author's knowledge, such an approach has not been done yet. However, with such a simulator, it would be easy to evaluate the performance of arbitrary algorithms using various FPN formats on the fly.

4. Implementation Approach

After establishing the necessary theoretical background, the general methodology and design of the simulator shall now be discussed. This thesis aims to investigate the possibilities of a simulator for arbitrary precision FPNs that tries to reflect the different time complexities of different FP formats. To accomplish this, implementing a prototype that falls into the realm of qualitative methods was chosen. However, quantitative methods will be applied to evaluate its performance after implementation. Throughout this chapter, a thorough explanation of the simulator's design will be given, followed by a brief discussion of how to guarantee correct behavior according to *IEEE 754 standard* [57].

4.1. Design of the Simulator

When it comes to simulating lower precision FP formats, the idea of other authors like Higham and Pranesh [55] is first to calculate the result in an available common higher precision followed by chopping off less significant bits thereby making the result less precise. It is not hard to see that this approach does not reflect the different time complexities of various FP formats but will need similar times regardless of the wanted format. To overcome this problem, the proposed simulator internally represents an FP number using a vector of boolean values. Since the simulator is written in C++, this was accomplished using an `std::vector<bool>` object. To be even more precise, each value is represented by not only one vector but actually three objects. These are a boolean value representing the sign of the number and two vectors consisting of boolean values, one for the *mantissa* and one for the *exponent*. This splitting of the bit array was done to simplify both the code and the process of writing it. To represent an FP using the bit array, the *IEEE 754 standard* [57] was used. Further, note that the simulator does not support *subnormal numbers* since that would have drastically increased its complexity and seemed unnecessary.

After two arbitrary numbers are presented using the simulator, computations can be made using only the bit arrays. By doing this, it is assumed that representations using longer bit arrays will need a longer time to compute, thereby making the simulator reflect the individual time dependencies. Nevertheless, one obvious downside of this approach is that it makes computation much slower compared to data types provided by the system. Therefore, it is expected that this strategy will be able to evaluate the efficiency of different FP formats but will only apply to small problem sizes. The difficulties with performance time were the main reason C++ was chosen as a programming language since it is considerably faster than, for instance, *Python*. Further, parallelism cannot

accelerate the simulator efficiently since the computation time of one single operation, like multiplication, is still incredibly fast.

Before continuing with a more detailed explanation of the simulator's workings, it should be emphasized that the program could easily be implemented in numerous different ways. However, it was tried to find the most efficient way to implement a routine to have a consistent simulator. Of course, replicating a specific CPU architecture could also be attempted, but this approach seemed far too sophisticated for a first prototype. It might also negatively influence the generality of the outcome since the result might be biased towards that one CPU architecture. Besides this, two other general design choices were made. The first one is that the bit arrays containing the FP information are treated as constant references when using them within operator calculations. This is done because using the keyword `const` should help the compiler optimize the code [40]. However, this choice also increased the complexity of the code since sophisticated indexing needed to be implemented to avoid unnecessary copying. The second choice is only to allow calculations using FP numbers if they have exactly matching FP formats, which was done because it reduces the complexity of the code. This introduces additional overhead in *multiprecision* algorithms because of casting, which could otherwise be overcome with clever indexing. However, it appears that this added complexity is not needed in the first prototype. Furthermore, it also seems to replicate the behaviors of a computer better because, for instance, in C++, one must use similar data types as well. After all, if types do not match, the compiler will cast implicitly [5, 8].

In the next chapters, the implementation of the most important parts of the simulator will be laid out so that the reader can get a feel for how the simulator works. Nevertheless, not everything can be discussed in detail, but please find the code for the whole simulator in the Appendix (A.1) for further reference.

4.1.1. Addition and Subtraction

They will be discussed together because addition and subtraction have decently similar implementations. The heart of both algorithms is a full adder. The main difference between the two routines is that in the case of a *subtraction*, the *subtrahend* must be inverted, followed by adding one to it before getting added to the *minued*. It is also important to note that an *addition* where the two involved numbers have opposite signs is actually a *subtraction* – a case that needs to be handled using a conditional statement. The same is true for *subtraction*, with the addition that a further condition must also be checked. Assuming that both numbers have the same sign, the result will cross the zero point if the magnitude of the *subtrahend* is larger than that of the *minued*. In that case, the *subtrahend* and the *minued* need to change places before performing the *subtraction*, followed by switching the *sign* of the result. How it is determined which one of the two numbers is larger can be found in Section 4.1.3.

Before discussing the case of *adding* and *subtracting* FP numbers, first, the more general case of *adding* or *subtracting* binary numbers will be considered. This will be mentioned first since *addition* can be implemented more easily than *subtraction*. The used code consists of a full adder as depicted in Algorithm 5. Please note that if the carrier bit c is set at the end of the last iteration, an overflow will happen, which must be accounted for to produce correct results. Nevertheless, to not unnecessarily increase the complexity of this chapter, such cases would not be discussed but can be studied when reading the full code provided in the Appendix A.1.

Data: $a, b \in \mathbb{B}^n$
Result: the result $r \in \mathbb{B}^n = a + b$
c = false
for $i = n-1, n-2, ..., 0$ **do**
  $r_i = a_i \wedge b_i \wedge c$
  $c = ((a_i \wedge b_i) \vee (a_i \wedge c)) \vee (b_i \wedge c)$
end

Algorithm 5: Binary addition using a full adder.

In the case of subtraction, the algorithm only needs to be changed a little bit, as shown in Algorithm 6. Please note that the function `binaryAddition()` is Algorithm 5.

Data: $a, b \in \mathbb{B}^n$ where $a \geq b$
Result: the result $r \in \mathbb{B}^n = a - b$
for $i = 0, 1, ..., n-1$ **do**
  $b_i = \neg b_i$
end
$b = b + 1$
r = binaryAddition(a, b)

Algorithm 6: Binary subtraction using a full adder.

After having the appropriate algorithms for general adding and subtracting of binary numbers, one more algorithm is needed before continuing with the *addition* and *subtraction* of FP numbers, namely an algorithm for matching the *mantissa* of one FP number with the *mantissa* of another. The resulting implementation can be seen in Algorithm 7. In the code shown, 0*s* are appended to the *mantissa*s, which is certainly a correct implementation; however, indexing was used to avoid unnecessary copying in the final implementation. For reasons of simplicity, this is not shown in this section but can again be found in the Appendix A.1.

Data: $o \in \mathbb{R}$ and $m_a, m_b \in \mathbb{B}^n$
Result: the adapted mantissas $r_a, r_b \in \mathbb{B}^n$
$r_a = m_a$
$r_b = m_b$
for $i = 0, 1, ..., o-1$ **do**
  r_a.appendBack(0)
  r_b.appendFront(0)
end

Algorithm 7: Matching the *mantissa*s of two FP numbers by an offset o.

Finally, it can be continued with the respective implementations for FPNs. The code for the used routine for *addition* can be found in Algorithm 8. One of the first things that are important to mention when studying the algorithm is that the initial separation of the bit array into *sign*, *mantissa*, and *exponent* does not need to be done inside the simulator since it saves the FP information in three separate parts already. A similar statement is true when combining the three parts back together, as indicated with the function `combine()`, in the last line of the algorithm. The function `round()` rounds according to the *IEEE 754 standard* [57]. Please note that the function `round()` also alters the exponent if a *carry* bit was present in the last step of `binaryAddition()`. The algorithm `round()` will not be discussed but can be found in the code provided in Appendix A.1. Further, the algorithm shown does not handle special cases like over- or underflow for simplicity. However, the simulator, of course, does account for this.

As before, *subtraction* is implemented in a very similar style to *addition*, as can be seen when studying Algorithm 9. The most important difference is that if the *subtrahend* is larger than the *minued*, the two numbers need to switch place inside of the *subtraction* because the result will be on the other side of the number line. Further, if that happens also, the *sign* needs to be changed accordingly. As before, the separation of the FPNs needs not be done inside the simulator as they are stored in that format in the first place. Furthermore, also this time, the `round()` function must be able to change the exponent to be able to store the numbers according to *IEEE 754 standard* [57]. As was the case before, not all special cases are depicted in Algorithm 9 but can be found in Appendix A.1.

4.1.2. Multiplication and Division

As was the case for *addition* and *subtraction*, *multiplication* follows a similar pattern to what is taught in elementary school when *multiplication* is trained with pen and paper. Nevertheless, for instance, some improvements can be made for *multiplication* when utilizing Booth's famous algorithm [11]. Algorithm 9 shows a top-level representation of the *multiplication* algorithm as implemented in the simulator. It can be seen that the complexity increases compared to the *addition* and *subtraction* as much more indexing is needed. In the first step, the components of the FP numbers are separated, which

Data: $a, b \in \boldsymbol{F}$
Result: the result $r \in \boldsymbol{F} = a + b$

```
// s...sign, m...mantissa, e...exponent
s_a, m_a, e_a = separate(a)                    // separate the bit array of a
s_b, m_b, e_b = separate(b)                    // separate the bit array of b
size = m_a.size()

m_a.insertFront(1)                             // add hidden 1 at the beginning
m_b.insertFront(1)                             // add hidden 1 at the beginning

if e_a > e_b then
    difference = binarySubtraction(e_a, e_b)
    m_a, m_b = matchMantissa(difference, m_a, m_b)
    e_r = e_a
else if e_b > e_a then
    difference = binarySubtraction(e_b, e_a)
    m_a, m_b = matchMantissa(difference, m_b, m_a)
    e_r = e_b
else
    e_r = e_a
end

m_r = binaryAddition(m_a, m_b)
r = combine(s_a, m_r, e_r)
r = round(r, size)
```

Algorithm 8: FP addition

again does not need to be done inside the simulator for the same reasons as above. The next step is determining the *sign* and calculating the resulting *exponent*. Note that this *exponent* is not necessarily the result's final *exponent* but might be changed due to the normalization process. In the case when both FPNs have an *exponent* larger than 0, basically the *exponent* of the result is the sum of the *exponent*s of the two initial numbers. If that is not the case, the basic operation to calculate the *exponent* is still an addition; however, it must be accounted for the special representation of the *exponent* inside an FPN.

Next comes the *multiplication* of the two *mantissa*s. For this, the two vectors S and m_r need to be set up according to the needs of the algorithms, thereby respecting the special FP format. This is mainly accounting for the "invisible" leading zero at the beginning of the *mantissa*. Afterward, the main routine of Booth's algorithm starts. The function `binarySummation` is very similar to the function `binaryAddition` as depicted in Algorithm 5. The main difference is that it was optimized for the use inside of *mul-*

Data: $a, b \in \boldsymbol{F}$
Result: the result $r \in \boldsymbol{F} = a + b$

if $b > a$ **then**
  switch(a, b)
  $s_r = \neg$ sign(a)
else
  s_r = sign(a)
end

// s...sign, m...mantissa, e...exponent
s_a, m_a, e_a = separate(a) // separate the bit array of a
s_b, m_b, e_b = separate(b) // separate the bit array of b
size = m_a.size()

m_a.insertFront(1) // add hidden 1 at the beginning
m_b.insertFront(1) // add hidden 1 at the beginning

if $e_a > e_b$ **then**
  difference = binarySubtraction(e_a, e_b)
  m_a, m_b = matchMantissa(difference, m_a, m_b)
  $e_r = e_a$
else if $e_b > e_a$ **then**
  difference = binarySubtraction(e_b, e_a)
  m_a, m_b = matchMantissa(difference, m_b, m_a)
  $e_r = e_b$
else
  $e_r = e_a$
end

m_r = binaryAddition(m_a, m_b)
r = combine(s_r, m_r, e_r)
r = round(r, size)

Algorithm 9: FP subtraction

tiplication and *division* by utilizing indexing. By using such indexing, the speed of the simulator could be increased drastically.

The algorithm for *division* as represented in Algorithm 11 has a similar complexity as the one for *multiplication* as well as a similar basic structure. Again, in the first step, the different parts of the FPNs need to be extracted, followed by the calculation of the *exponent*. What happens is that the *exponent* form the *dividend* needs to be added

or subtracted by the one from the *divisor* depending on whether the *divisor* is smaller or larger than 1. The additional complexity comes mainly from the specialties of the FP format. Followed by the calculation of the *exponent*, the main routine performs a *division* using the two *mantissa*s. The routine used is straightforward and follows the same principle as a division of pen and paper.

The last step is similar for both *multiplication* and *division*. First, the different parts of the resulting FPN are set back together, followed by a normalization step and proper rounding. The exact routine for normalization and rounding will not be discussed but is expected to work according to *IEEE 754 standard* [57]. For both algorithms, it is also true that the handling of special cases or similar is not considered. This is because it would unnecessarily lengthen the depicted algorithms. However, the full implementation can be found in Appendix A.1.

4.1.3. Comparators

To use the simulator properly, the most common comparators $c \in \{<, >, \leq, \geq, ==, !=\}$ were implemented. The last two ($==$ and $!=$) are pretty straightforward. They both compare all the bits of an FPN, where the first one ($==$) returns *true* if all bits match, whereas the latter ($!=$) returns *true* if at least one bite does not match. The other four are also pretty simplistic and follow the same general pattern. As an example, the code for $<$ is given in Algorithm 12. The implementation for the other three is omitted since it can easily be deduced when studying either Algorithm 12 or the code provided in Appendix A.1. As can be seen, the implementation is pretty straightforward. It is iterated over each bit in order of significance, and as soon as a difference occurs, a decision about the outcome can be made.

4.1.4. Special Values

For a properly working simulator, it must be able to deal with the most common special values, which are `0`, `NaN`, `Inf`, and `-Inf`. All of those have special bit patterns, which must be checked before each operation since, in such a case, a special behavior is triggered. The exact listing of what will happen if a special value is encountered will not be listed here but can easily be found by either consulting the *IEEE 754 standard* [57] or the program code in the Appendix (A.1). Nevertheless, one special case will be mentioned. The proposed simulator does not differentiate between `+0` and `-0`. However, some machines do make a difference. Nevertheless, this decision should not make any significant difference during computation, and it was chosen because of the slightly greater simplicity.

4.1.5. General Workflow

Before continuing to the next section, it shall briefly be discussed how the general workflow of the experiments was implemented. First, the simulator was developed test-driven using a singular C++ class. On top of that, a second class was introduced using the objects of the simulator to implement all needed (numerical) algorithms. This class was also thoroughly tested using unit tests. Having all the necessary basic code, the test-driven development method was loosened due to time requirements and the fact that the number of written tests was already at $1,076$. A third C++ class was written to automate the experiments. This class then was important to *Python* using pybind11 [61] for easy handling. To structure the compilation process, *CMake* [24] was used.

4.2. Validation of the Code

As is the case for all coding projects, the correctness of the produced code must be validated to guarantee proper results. Because of this, a method of test-driven development was chosen. Therefore, hundreds of unit tests were written for the simulator alone to ensure the accuracy of the code. Additionally, after implementing the operators, randomized tests were added. This means for each of the four operators $op \in \{+, -, *, /\}$, two random FPNs were chosen, followed by a check for correctness of the calculation. To evaluate such correctness, the result is first computed using data types implemented natively inside the system and then compared bit by bit with the result of the simulator. If the result when using system data types is a subnormal number, it would have needed to be excluded since the simulator does not support them. However, during the experiment, no calculation yielded such a result. The experiment was run for a total of $10,000,000$ each time, resulting in correct calculations.

To make a statement of the correctness of the code, the Wilson score interval is used [35, 101, 108]. At a level of significance of 99%, the test predicts a confidence interval of (0.999999, 1.000000) for every operator and both *single* and *double* precision. This suggests a high confidence that the simulator is working properly. Unfortunately, this is insufficient to ensure that the simulator is working properly since some important special cases are rare. For instance, when using *single* precision, the chances of randomly selecting the same number twice are $\frac{1}{254 \times 2^{23}}$. Therefore, such a combination will only occur with a probability of $\frac{10^7}{254 \times 2^{23}} \cong 0.469\%$ during all the 10^7 runs. Despite this not being a lot, such a case occurs frequently when using *PLU factorization* as depicted in Algorithm 1 since the reciprocal of a value plays an important role. The chances of encountering such a combination for *double* precision are even lower with $\frac{10^7}{2046 \times 2^{52}} \cong 10^{-11}\%$, which means it is practically impossible to hit such a case.

To address this problem, numerous unit tests were written to account for as many special cases as possible. Furthermore, another kind of randomized test was conducted where the results of *GESV* were calculated using the simulator and compared against

the results when using native system data types. Again, no errors occurred. All of this strongly indicates that the simulator works according to *IEEE 754 standard* [57]. Nevertheless, faults can never be ruled out completely, as with all written computer code.

4.3. Generating Sparse Matrices

For one of the presented experiments (see Section 5.2), it is necessary to generate sparse matrices. Sparse matrices have a very high percentage of *zero* values [94]. In this thesis, they are produced by using a *sparsity rate*, which defines the percentage of *zero* values of a matrix. So, for instance, if a *sparsity rate* of 0.8 is given, this means that 80% of the values of a matrix are *zeros*. However, in the actual implementation, it is not exactly 80% since it is written in such a way that, in this case, every value only has an 80% chance of being *zero*, which introduces some randomness into the system. Nevertheless, the larger the matrix is, the closer the real value will be to the defined one. This variant was chosen to keep the implementation rather simplistic.

Unfortunately, just setting every entry with a certain percentage of a matrix to *zero* will not provide the expected results since it does not guarantee *invertibility*. Therefore, before setting all other elements of the matrix, one randomly selected entry of each matrix row is first set to a *nonzero* value, thereby ensuring that the matrix is indeed *invertible*. Disappointingly, now setting all the other elements to *zero* according to a previously defined *sparsity rate* would result in a wrong rate since a certain number of entries are already *nonzero*. Therefore, by using the original *sparsity rate* r, a new *adapted sparsity rate* r_a must be calculated. The way this adjustment is made is explained in the following lemma. Please note that this lemma would also work for a more complex implementation where the percentage of *zeros* exactly coincides with the given *sparsity rate*. Nevertheless, it is also applicable to the more simplistic approach chosen in this thesis.

Lemma 4.1. *Given a sparsity rate r, which is the expected percentage of zero elements inside a matrix $M \in \mathbb{R}^{n \times n}$. If the matrix M already contains n nonzero elements, all the other $n(n-1)$ elements must be set according to a new adapted sparsity rate r_a, which is defined as*

$$r_a = \frac{r \cdot n}{n-1} \tag{4.1}$$

where n is the size of the matrix.

Proof. First, the total number of *zero* elements in M is $r_a \cdot n(n-1)$ since the other complementary n elements do not contain any *zero* by definition. After dividing by the total number of elements (n^2), the following formula for the *sparsity rate* r is obtained:

$$r = \frac{r_a \cdot n(n-1)}{n^2} \tag{4.2}$$

which can easily be transformed into Equation 4.1

□

4.4. Model-Based Transformation

As will be shown when examining the results in Chapter 5, *addition* and *subtraction* run much faster than *multiplication* and *division*. This fact can also be deduced by looking at the respective algorithms in Section 4.1.1 and 4.1.2 where it is visible that the algorithms for *addition* (Algorithm 8) and *subtraction* (Algorithm 9) should experience linear *time complexity* compared to quadratic *time complexity* when doing a *multiplication* (Algorithm 10) or a *division* (Algorithm 11). This is not a problem per se, but when wanting to have results similar to a CPU, this must be addressed in some way. Therefore, a mathematical solution will be introduced in this section to mitigate this issue.

The idea behind the method is to replicate the behavior of a CPU where the time for *multiplication* and *division* is not quadratic, which is due to sophisticated optimizations in the CPU like parallelism [89]. The mathematical approach is necessary since optimizations like this cannot be efficiently replicated using C++ [107]. The goal of this mathematical approach is to artificially transform the quadratic *time complexity* into a near linear one. To be able to do this, the idea is that by taking a square root, quadratic behavior appears linear. This strategy is, of course, very simplistic and has by no means the expectation of replicating the behavior of a CPU exactly. It is mainly meant to sketch one probable solution to the problem and explore the possibilities of such a naïve approach.

To understand the method more deeply, first note that the *time complexity* $\mathbf{T}$ of an arbitrary algorithm that uses data types provided by the simulator can approximately be described as follows:

$$\mathbf{T} = \mathcal{O}(m_s) * n_s + \mathcal{O}({m_p}^2) * n_p \tag{4.3}$$

where :

m_s = the *mantissa* length for operations in the algorithm that form sums.
m_p = The *mantissa* length for operations in the algorithm that form products.
n_s = the number of operations in the algorithm that form a sum.
n_p = the number of operations in the algorithm that form a product.

Because *addition* and *subtraction* have a similar *time complexity*, they are grouped together and are denoted using a subscript s to indicate that these operators compute a sum. Similarly, *multiplication* and *division* are put into another group, denoted using subscript p to point out that these operators compute a product.

The presence of the quadratic term in $\mathcal{O}({m_p}^2)$ poses an issue when wanting to mirror a CPU since it is typically not present on a CPU, as will be shown in chapter 5. Because of that, it would be nice if it could be interchanged with $\mathcal{O}(m_p)$ instead, which would result in the following *wanted time complexity*:

$$\mathbf{T}_{wanted} = \mathcal{O}(m_s) * n_s + \mathcal{O}(m_p) * n_p \tag{4.4}$$

It should be clear that the simulator regrettably does not achieve this relation. Nevertheless, by dividing by $\mathcal{O}(m_p)$, a very similar formula can be achieved:

$$\mathbf{T}_{adapted} = \frac{\mathcal{O}(m_s) * n_s + \mathcal{O}({m_p}^2) * n_p}{\mathcal{O}(m_p)} \tag{4.5}$$

The expression above can be simplified to:

$$\mathbf{T}_{adapted} = \frac{\mathcal{O}(m_s)}{\mathcal{O}(m_p)} * n_s + \mathcal{O}\left(m_p\right) * n_p \tag{4.6}$$

The resulting estimation already looks decently close, but unfortunately, $\mathcal{O}(m_p)$ is not known. Nevertheless, it can be approximated by:

$$\alpha(m_p) := \sqrt{\mathcal{O}({m_p}^2)} \cong \mathcal{O}(m_p) \tag{4.7}$$

By using α Formula 4.5 can conveniently be rewritten written as:

$$\mathbf{T}_{adapted} = \frac{\mathbf{T}}{\alpha(m_p)} \cong \frac{t_{simulator}}{\alpha(m_p)} \tag{4.8}$$

In this case, $t_{simulator}$ describes the actual time measurement when running an algorithm using the data types of the simulator. Utilizing α also has the benefit that speedups can be calculated easily using:

$$speedup = \frac{\frac{t_1}{\alpha_1}}{\frac{t_2}{\alpha_2}} = \frac{t_1 * \alpha_2}{t_2 * \alpha_1} \tag{4.9}$$

Here, t_1 describes the execution time of the algorithm using a *mantissa* length m_1. Respectively, t_2 denotes the execution time of the same algorithm using a *mantissa* length of m_2. The constants α_1 and α_2 are the corresponding factors for m_1 and m_2

4.4.1. Error Analysis

At least for *multiplication* and *division*, the adapted model appears to be pretty near to the *wanted* solution. Unfortunately, the correction affects the first term as well. Compared to the *wanted* solution (Equation 4.4) the following absolute error e_a will be made when using the *adapted* model:

$$e_a = |\mathbf{T}_{wanted} - \mathbf{T}_{adapted}| \tag{4.10}$$

which is equivalent to:

$$e_a = \left|\mathcal{O}(m_s) - \frac{\mathcal{O}(m_s)}{\mathcal{O}(m_p)}\right| * n_s \tag{4.11}$$

It is immediately imminent that the adapted result will underestimate the correct runtime if $\mathcal{O}(m_p) > 1$ and $\mathcal{O}(m_s) > 0$, which will be the case in all of the discussed results. Due to this underestimation and the fact that the runtime cannot be negative, e_a is bounded as follows:

$$e_a \leq \mathcal{O}(m_s) * n_s \tag{4.12}$$

This means that in the worst case, the results appear to ignore *addition* and *subraction* completely.

On the contrary when comparing the original model $\mathbf{T}$ with the wanted solution $\mathbf{T}_{wanted}$ the following absolute error e_o is obtained:

$$e_o = |\mathbf{T}_{wanted} - \mathbf{T}| \tag{4.13}$$

which is equivalent to:

$$e_o = \left|\mathcal{O}(m_p) - \mathcal{O}(m_p^2)\right| * n_p \tag{4.14}$$

Compared to e_a, the error e_o cannot be bound as well as before. However, by ignoring the term $\mathcal{O}(m_p)$ the following upper bound can be achieved:

$$e_o \leq \mathcal{O}(m_p^2) * n_p \tag{4.15}$$

From Equation 4.14 and 4.15, it is obvious that e_o will grow quadratically as m_p grows linearly. Therefore, the error e_o becomes more pronounced for larger *mantissa* sizes. Thus, the proposed model ($\mathbf{T}_{adapted}$) seems preferable if the goal is to achieve results similar to those of a modern CPU. Nevertheless, it should be mentioned again that this is only a very simplistic view of the underlying workings of a modern CPU.

It is also important to note that the proposed adaptation affects not only *addition* and *subtraction* but also other components of the algorithm, such as comparators. Although their impact is assumed to be negligible and therefore omitted from this analysis, their presence should still be considered. It is also worth mentioning that this analysis only holds for one singular *mantissa* size for operations that form a sum and one for operations that form a product. Therefore, it needs to be adapted for more complex settings.

Data: $a, b \in \boldsymbol{F}$
Result: the result $r \in \boldsymbol{F} = a * b$

```
// s...sign, m...mantissa, e...exponent
s_a, m_a, e_a = separate(a)                    // separate the bit array of a
s_b, m_b, e_b = separate(b)                    // separate the bit array of b
s_r = s_a ⊕ s_b

// both exponents are positive
if e_a[0] ∧ e_b[0] then
    s = e_b; s[0] = ¬s[0]; s += 1
    e_r = binaryAddition(e_a, s)
else
    s = ones(e_b.size()-1)                                  // s = [1,1,...,1]
    s.insertFront(0)
    if e_a[0] ∧ ¬e_b[0] then
        s = binarySubtraction(s, e_a)
        e_r = binarySubtraction(e_b, s)
    else
        s = binarySubtraction(s, e_b)
        e_r = binarySubtraction(e_a, s)
    end
end

S = m_a.invert() + 1
S.insertFront(1)

m_r.zeros(m_a.size() + 2)                              // m_r = [0,0,...,0]
m_r.insertBack([0, 1, m_b, 0])

for i = 0, 1, ..., m_b.size() + 2 do
    if ¬P[−2] ∧ P[−1] then
        binarySummation(m_r, m_a)
    else
        binarySummation(m_r, S)
    end
end

r = combine(s_r, m_r, e_r)
normalize(r)
round(r, m_a.size())
```

Algorithm 10: FP multiplication (Booth's algorithm)

Data: $a, b \in \boldsymbol{F}$
Result: the result $r \in \boldsymbol{F} = a/b$

```
// s...sign, m...mantissa, e...exponent
s_a, m_a, e_a = separate(a)                    // separate the bit array of a
s_b, m_b, e_b = separate(b)                    // separate the bit array of b
s_r = s_a ⊕ s_b

if e_b[0] then
    s = e_b; s[0] = ¬s[0]; s += 1
    e_r = binarySubtraction(e_a, s)
else
    s = ones(e_b.size()-1)                              // s = [1,1,...,1]
    s.insertFront(0)
    s = binarySubtraction(s, e_b)
    e_r = binaryAddition(e_a, s)
end

R = m_a
R.insertFront([0, 0, 1])
S = m_b.invert() + 1

for i = 0, 1, ..., m_a.size() do
    shiftLeft(R)
    if R ≥ m_b then
        binarySummation(R, S)
    end
end

r = combine(s_r, m_r, e_r)
normalize(r)
round(r, m_a.size())
```

Algorithm 11: FP division

Data: $a, b \in \boldsymbol{F}$
Result: the result $r \in \mathbb{B}$ of $a < b$

```
// s...sign, m...mantissa, e...exponent
```
s_a, m_a, e_a = separate(a) `// separate the bit array of` a
s_b, m_b, e_b = separate(b) `// separate the bit array of` b
$s_r = s_a \oplus s_b$

if $\neg s_a \wedge s_b$ **then**
  return true
end

for $i = 0, 1, ..., e_a.size() - 1$ **do**
  if $\neg e_a[i] \wedge e_b[i]$ **then**
    return true
  end
end

for $i = 0, 1, ..., m_a.size() - 1$ **do**
  if $\neg m_a[i] \wedge m_b[i]$ **then**
    return true
  end
end

return false

Algorithm 12: Algorithm to evaluate $a < b$ where $a, b \in \boldsymbol{F}$.

5. Experimental Results

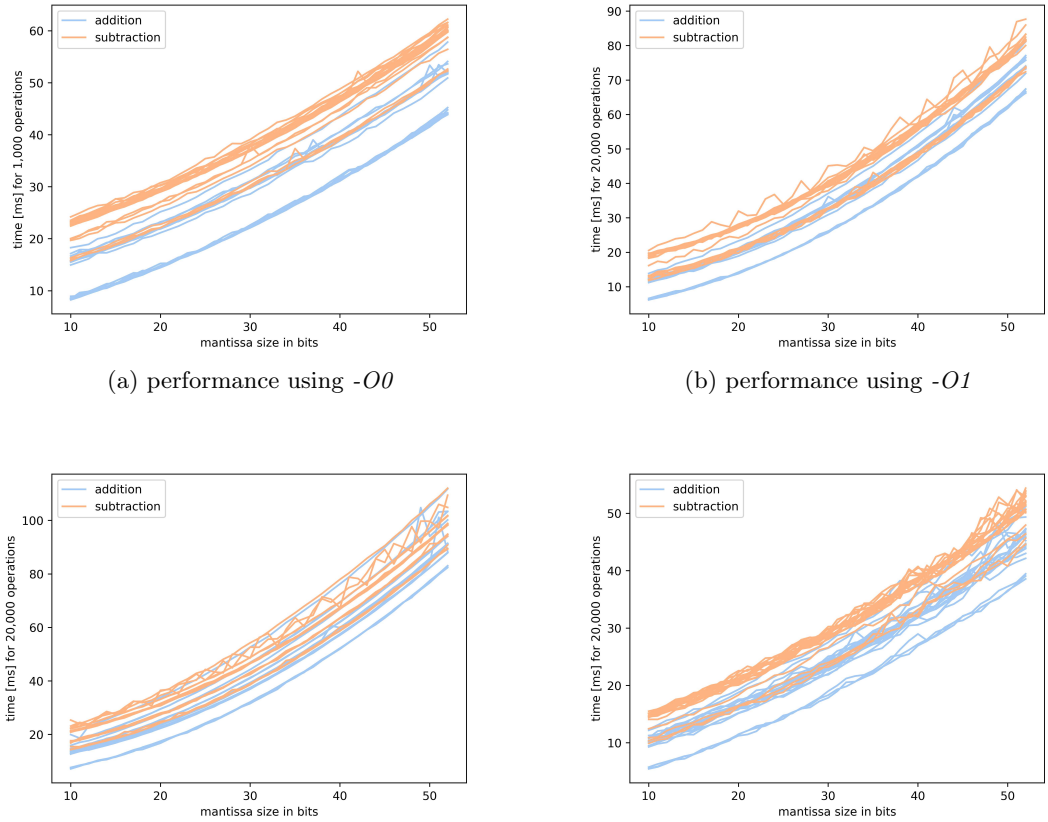
In this chapter, experimental results shall be presented. Most of these experiments are conducted using a server with 256 CPUs distributed on four *AMD EPYC 9534 64-core processors* with a maximum frequency of about 3.7 GHz. The problem will be addressed that comparisons between *single* and *double* precision are troublesome since, on modern CPUs, the computation time is often similar for both precisions. This effect appears because present-day processors are often optimized so that both operations only need one clock cycle. Besides experiments on the previously mentioned server, some further experiments were conducted on a *2020 M1 MacBook Air* to showcase behavior on a different platform.

As done in the chapter above, the simulator is sometimes referred to as Multi Precision Simulator or, for short, MPS. Further, *natively* implemented operators, for instance, when using *float* while programming, will be referred to as *system* operators. The evaluation of the simulator will start by studying the performance of the *basic* MPS operators $op \in \{+, -, *, /\}$. Because the amount of optimization by compiling can be chosen freely, this will also include the usage of different optimization levels. Following this, the performance of the MPS operators will be compared to the performance of when using system operators, which will be done by applying them inside simple algorithms. In contrast to the previous experiments, this part will not only be done on the server but also using a *2020 M1 MacBook Air*. Lastly, a small case study will be depicted on the server to illustrate potential use cases of the simulator.

5.1. Operators

5.1.1. Addition and Subtraction

The first operators analyzed are *addition* and *subtraction*. The whole simulator was compiled four times using a different optimization option to show the effects of different optimization levels. The results are depicted in Figure 5.1, where each subplot represents a different optimization option. Each line in Figure 5.1, was generated by choosing two positive numbers from $\{x \in \mathrm{F} \mid double.min() \leq x \leq 1,000,000\}$ followed by an addition or subtraction using different *mantissa* sizes. Please note that for *-O0*, each operation was only repeated 1,000 times before making a time measurement in contrast to the other three, where 20,000 repeats were made. Furthermore, each plot shows 20 different runs for each operator. However, 20 more were performed for each operator as a warmup. It is also important to mention that the *exponent* is fixed to 11 bits for all experiments.



(a) performance using *-O0*

(b) performance using *-O1*

(c) performance using *-O2*

(d) performance using *-O3*

Figure 5.1.: The performance of *addition* and *subtraction* using MPS with different optimization levels. Each line represents one operation for different *mantissa* sizes.

Therefore, the differences are only influenced by the size of the mantissa.

When studying Figure 5.1, it becomes clear that when using no optimization, the simulator is about 20× slower than when using at least some optimization. So, there appears to be a strong incentive to compile the simulator with at least some optimization options. Nevertheless, regardless of the optimization option, all curves appear decently linear, even though some exponential behavior is visible. This outcome is as wanted because a linear relation is to be expected. The exponential behavior, however, is mostly visible when compiling using *-O1* and *-O2*, and it is also interesting to note that apparently, the performance is better when using *-O1* compared to *-O2*. When using *-O3*, the curves appear again a bit more linear, as was the case when using no optimization, but the results appear a bit more unstable. It is evident that throughout all used optimizations

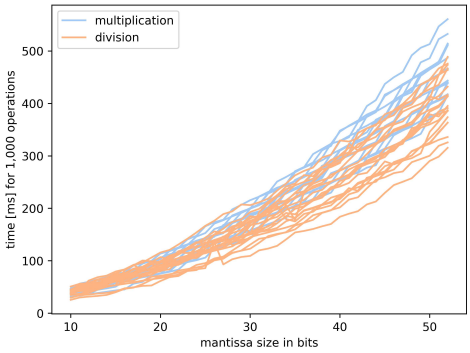
levels, *subtraction* needs longer than *addition*, which is also as expected as the bit array of the *subtrahend* must be inverted. A process that need not be done when performing an *addition*. Furthermore, in all figures, some lines seem to be shifted by a constant, which seems odd at first glance but is also as anticipated. It can be explained by the fact that when two numbers have the same *exponent*, they can be added or subtracted instantly. On the other hand, if this is not the case, the two numbers first need to be shifted according to their *exponents'* differences. Such an effect is not visible when performing system operations, but it is necessary to ensure correct computation in the simulator. In the end, it can be deduced that compiling with *-O3* seems preferable – at least on the used server.

5.1.2. Multiplication and Division

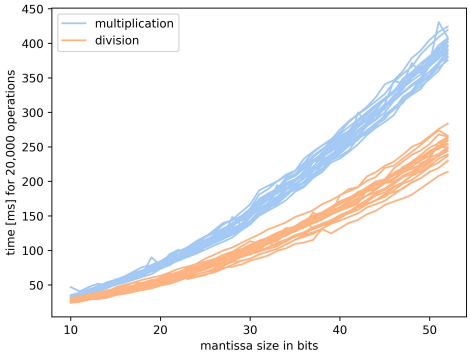
The next operators analyzed are *multiplication* and *division*. Again, the whole simulator was compiled four times using different optimization options, and the results are shown employing a subplot, each representing one optimization option. The experiment setup is very similar to the one for *addition* and *subtraction*. First, two positive numbers are chosen for each line from the set $\{x \in \mathrm{F} \mid double.min() \leq x \leq 1,000,000\}$, followed by *multiplication* or *division*. As done previously, 20 different lines were generated for each operator, preceded by the same amount as a warmup. Similar to the findings above, an immense performance benefit could be seen when using any optimization option. Because of that, 1,000 runs were executed for the case of no optimization, but 20,000 could be made when at least some optimization was present. Results are shown in Figure 5.2 and as was the case in the section above, the length of the *exponent* was fixed to 11 bits during the experiment.

The first thing that can be seen from Figure 5.2 is that all operations appear to be quadratic. This behavior is certainly not as wanted when comparing it to the expected behavior on FP arithmetic as implemented on a CPU. However, it is certainly as expected since the algorithms used for *multiplication* and *division* have a *time complexity* of $\mathcal{O}(n^2)$. Modern CPUs can achieve this linear behavior likely because of parallelization inside the Floating Point Unit (FPU). Such parallel designs are essential components in modern designs [89] and were impossible to replicate in the simulator used in this thesis. As can be seen in the next section, when using natively implemented data types of the system, *multiplication* and *division* do not even take longer than *addition* and *subtraction*.

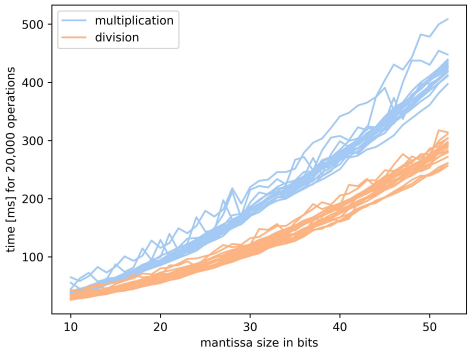
Nevertheless, the exponential part, as also seen for *addition* and *subtraction*, appears not to be extremely dominant. Still, on the whole, much more time is needed when performing a *multiplication* or *division*. With this being the case, all computations using the simulator will be predominately governed by the performance of *multiplications* and *divisions*. This condition is most definitely not something to be wanted for a simulator. At least not when results from a CPU should be mirrored, because with this being the case, the simulator's performance for *addition* and *subtraction* is indirectly neglected since it vanishes in comparison.



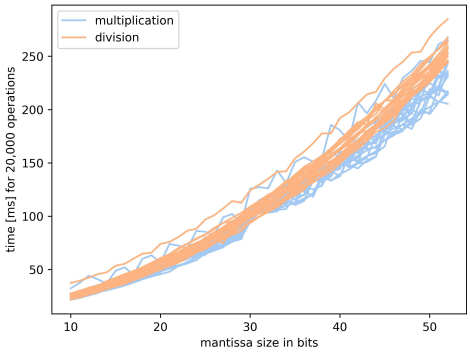
(a) performance using -*O0*



(b) performance using -*O1*



(c) performance using -*O2*



(d) performance using -*O3*

Figure 5.2.: The performance of *multiplication* and *division* using MPS with different optimization levels. Each line represents one operation for different *mantissa* sizes.

Furthermore, because the simulator's performance is much better when using optimization, it seems destined to be used. When -*O1* or -*O2* is used, *multiplication* and *division* seem to perform differently, which appears undesirable because it is not so in the unoptimized version. Matching performances are again visible when using -*O3*. Due to this and the better performance, complying with -*O3* seems to be the best option, as was the case for *addition* and *subtraction*.

5.1.3. Comparison of Exponent

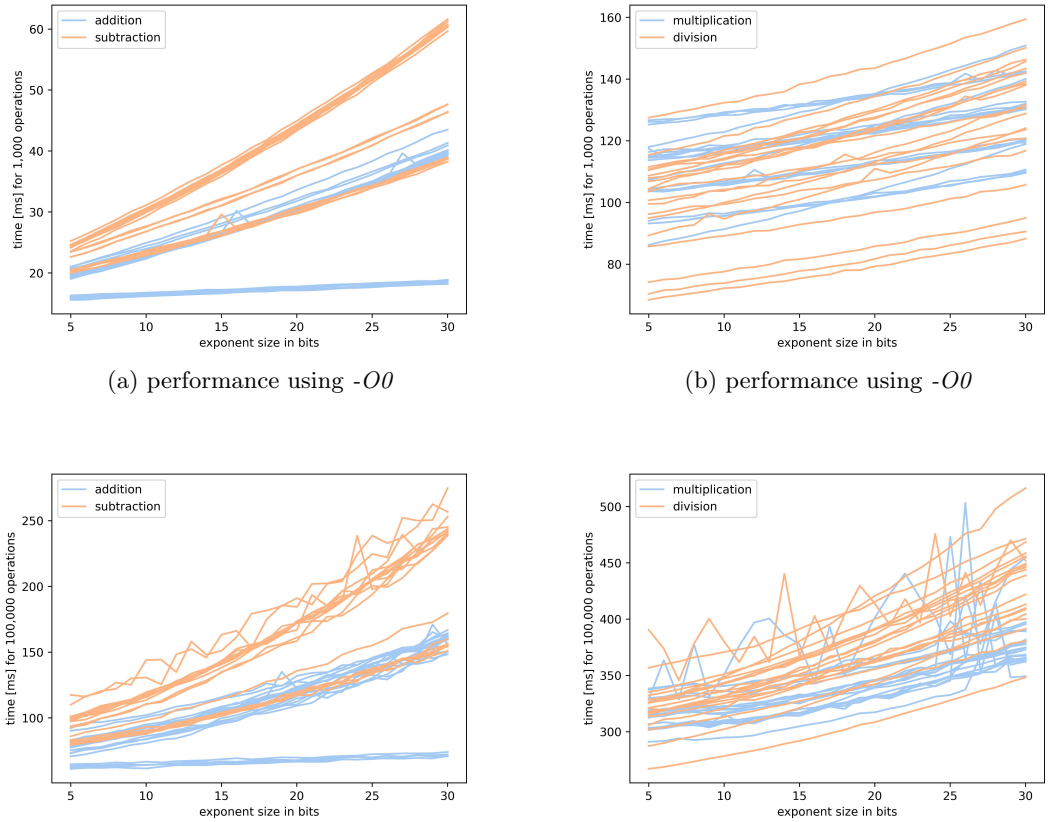
Unlike the preceding two sections, the focus shifts from analyzing the *mantissa* to examining the *exponent*. For this purpose, the size of the *mantissa* was fixed to 23 bits, as is the

case for *single* precision. The results will only be shown when using optimization levels *-O0* and *-O3* to save space. Experiments for the other optimization levels were conducted as well but did not provide further insights. What can be seen when studying Figure 5.3 is that when using optimization, again, the performance of the simulator is drastically increased. When it comes to performance, it is visible that increasing the *exponent* has linear or slightly quadratic *time complexity*, which seems as expected. Interestingly, for some numbers, expanding the length of the *exponent* has almost no effect on the runtime, which can be explained by the fact that if the *exponent*s match, no extra calculations regarding the *exponent* need to be carried out. At first glance, it also looks puzzling that the lines in Figure 5.3b and Figure 5.3d are very spread out. However, this is also the case when looking at Figure 5.2, only it is less noticeable due to the lines being far more compressed in the latter. Generally speaking, it can be said that the *exponent's* length influences computation time. But it seems fair to state that the effects can be neglected since the feasible range of the *exponent* is probably between 5 and 15 – a range where the differences appear not overly pronounced.

5.1.4. System Operators

Above, it was shown how operations using the MPS simulator perform. Now, those results are compared to those implemented natively on the server. The general setup of the experiment was conducted, as seen above. First, two positive numbers for each operation for the set $\{x \in \mathrm{F} \mid double.min() \leq x \leq 1,000,000\}$ were chosen, followed by the performing operation in question repeatedly for 1,000,000,000 times. The last number alone reveals how much better natively implemented data types perform than those from the simulator. Each subexperiment was repeated 100 times to ensure valid results, each time choosing a new combination of values. To accelerate this process, 20 treads were used for parallelization.

When plotting Figure 5.4, no outliers appear to be visible. Unfortunately, this is not true because many outliers were present in the case of *addition*. However, because this would have made the plot less readable without achieving valuable new insights, it was decided not to display them. Since the addition was calculated initially, the presence of the outliers might be explained due to an insufficient warmup. To address this problem, the number of warmup runs was elevated to 400 operations preceding the experiment, resulting in minor benefits compared to lower warmup times. Furthermore, the experiment needed to be carried out using *-O0* as an optimization setting. If choosing a higher optimization level, the operations that should be evaluated are deleted by the compiler because the result is not used besides taking time measurements. It would have been possible to overcome this problem with special programming tricks. However, except for small changes in the division, no substantially different outcomes could be seen, so they are not discussed further. Also, as mentioned, the results for the *addition* appear more spread out, possibly because of hardware-specific reasons. Findings like this will also not be addressed further since they are irrelevant for further experiments.



(a) performance using -*O0*

(b) performance using -*O0*

(c) performance using -*O3*

(d) performance using -*O3*

Figure 5.3.: The performance of *addition subtraction multiplication* and *division* using MPS with different optimization levels. The mantissa size is fixed, and each line represents one operation for different *exponent* sizes.

When looking at the results from Figure 5.4, it is clear that all operators need almost the same time, regardless of which precision format was chosen. As mentioned above, this behavior is likely caused by the parallel optimization inside CPUs, which can reduce the cycle time [89]. For instance, Booth's algorithm can be parallelized inside the FPU. But, later on, when using different data types inside algorithms, contrasts between the data types will become visible, which is likely because of the added complexity. Nevertheless, it should be evident that because of this behavior, it is surely not to be expected that the simulator performs similarly to the CPU.

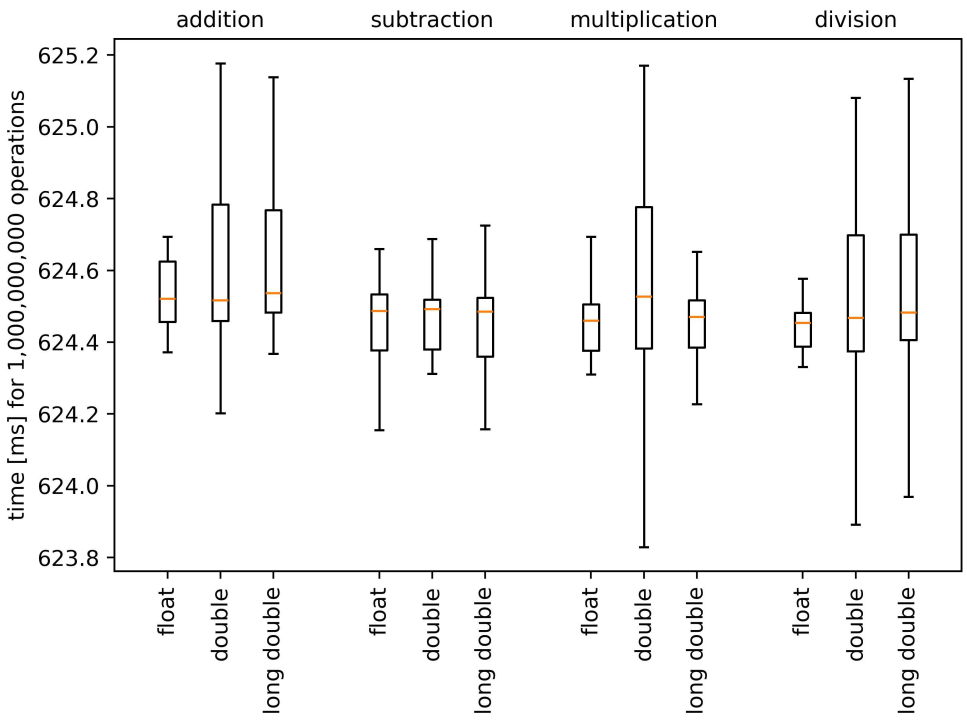
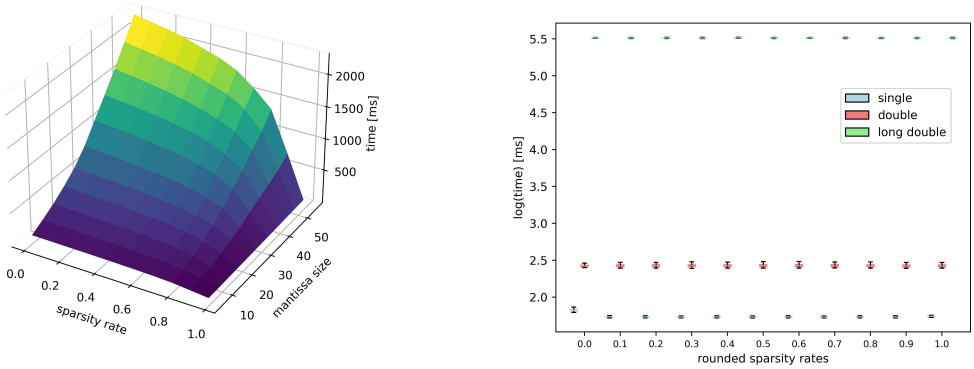


Figure 5.4.: The performance of different operations with natively implemented data types on the used server.

5.2. Comparison of Sparse Matrices

On the one hand, some authors mentioned in Section 3.5 do their analysis using dense matrices. On the other hand, some also conduct them using sparse matrices. Therefore, this small passage will test the simulator's performance on different matrices with different sparsity levels. The matrices are generated as described in Section 4.3. To evaluate the simulator's behavior, the computation time of *iterative refinement* was tested using system matrices with different sparsity rates. The used system matrices all have a dimension of $n = 50$, and if the value of an entry is not set to 0, it is chosen randomly from the set $\{x \in \mathrm{F} \mid -100 \leq x \leq 100\}$. As can be seen in Figure 5.5a, the sparsity of the matrice does make a difference in the performance in contrast when doing the same experiment with system data types as depicted in Figure 5.5b. Nevertheless, this effect appears to be relatively similar regardless of the size of the *mantissa*, so it should not affect the analysis of the following chapters since only speedup factors are compared, where such effects cancel each other out. Because the computation time is higher for dense matrices, such matrices will be used in the coming experiments to circumvent the possible negative effects of when handling small values. Further, note that the sparsity rates in Figure 5.5b are rounded. So, the right-most entry does not have a sparsity rate of 1 – which would be



(a) sparsity comparison mps

(b) sparsity comparison system

Figure 5.5.: Comparison of the behavior of matrices with different sparsity.

impossible if the matrix should also be invertible – but one of about 0.998. Also, note that the computation time for *long double* is far greater than for the other two data types. On the used system, *double* has a size of 8 bytes compared to a size of 16 bytes when using *long double*. Therefore, a doubling in time seems reasonable. Nevertheless, it also shows that *single* is apparently not implemented significantly differently from *double* since computations do not take half the time despite *single* only having half the size of *double*.

5.3. Comparison of Algorithms

This section will compare the simulator's performance to data types implemented natively in the systems used. At first, the results will be shown when performing them on the previously mentioned server, but later on, the same experiments will also be conducted on a *2020 M1 MacBook Air*. The general setup is similar for all experiments. Initially, time measurements were taken for all experiments when computing the result using double precision, followed by casting all inputs to *single* precision and evaluating the same algorithm again. Using both results makes it possible to calculate a speedup factor, which can be used for comparisons. A total of four algorithms were evaluated:

- General Matrix Vector Multiplication (GEMV)
- General Matrix Matrix Multiplication (GEMM)
- General Triangular Factorization (GETRF)
- General Solve (GESV)

Compared will be the vanilla results from the simulator denoted as **mps**, the results transformed according to the proposed mathematical solution from Section 4.4 denoted

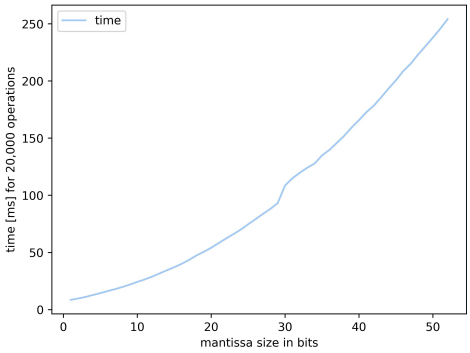
as **mps_factor**, and the results as obtained when using data types as provided by the used system denoted as **system**.

5.3.1. Comparisons: Server

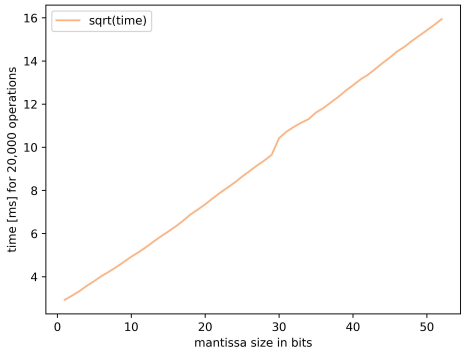
Each of the following experiments was conducted multiple times to get robust measurements. The so-generated experiments are reduced using the median, which is more robust than the average. Since this is not a statistical analysis, the median is not as important here. The average seems more appropriate, as the median would merely represent a combination of individual sub-results rather than a combination of all results. Additionally, parallelization was used to accelerate this process. Unfortunately, the simulator per se cannot be accelerated through multithreading, but the experiments as a whole can be. Despite having more cores available, only 20 of them were used. This is because it needs to be avoided that not enough cores are available at some point, which would falsify the results as the computer would then start to simulate parallelization [107]. The so lost increase in lost performance compared to when using mode threads is not a problem since it is needless to achieve decent results. All experiments have in common that the number of experiments done as a warmup is set to be equal to the number of used threads because, in that case, each thread performs one whole experiment as a warmup, which appeared adequate. The input values necessary for each experiment were chosen randomly from the set $\{x \in \mathrm{F} \mid -100 \leq x \leq 100\}$

Further note that, for instance, calculating a MVM takes much more time when applying it on larger matrices. Therefore, all experiments in this and the following section are implemented so that the repetitions for each matrix size are calculated individually to achieve similar runtimes for every matrix size. For example, when it is stated that 10,000 repetitions were made, it actually means that 10,000 repetitions were applied for the largest matrix but were possibly repeated much more often for smaller sizes to achieve approximately the same runtime. An implementation like this was chosen since more runtime also has a larger averaging effect because, in total, more computations are performed, and it was attempted to accomplish a similar averaging effect throughout all matrix sizes.

Next, before getting to the actual results, a function must be derived to calculate α for a given *mantissa* length m as depicted in Section 4.4. Since *multiplication* and *division* appear to have the same time complexity, this will only be done for *multiplication* but is expected to hold for both *multiplication* and *division*. To approximate the function $\alpha(m)$, first, 400 random multiplications were performed and later reduced using the median. Each operation was repeated 20,000 times before taking time measurements to attain results sufficiently large. Further, in contrast to all the other experiments in this section, the input values are chosen from the set $\{x \in \mathrm{F} \mid double.min() \leq x \leq 1,000,000\}$. The resulting curve is plotted in Figure 5.6a and accompanied by the square root of the function to the right in Figure 5.6b. According to Equation 4.7, the latter is the wanted approximation for the function $\alpha(m)$. Please note that the function in Figure 5.6b should



(a) computation time of *multiplication*



(b) square root of the computation time of *multiplication*, which will be used as $\alpha(m)$

Figure 5.6.: Functions in order to calculate $\alpha(m) = \sqrt{\mathcal{O}(m^2)}$.

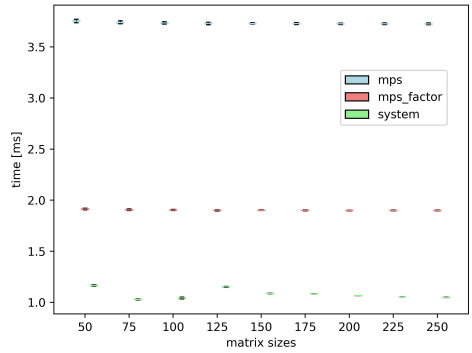
be scaled by a factor of $\frac{1}{20{,}000}$. However, since the only interest in this section is to calculate factors, not doing this makes no difference and is omitted to increase numeric stability.

Matrix Vector Multiplication

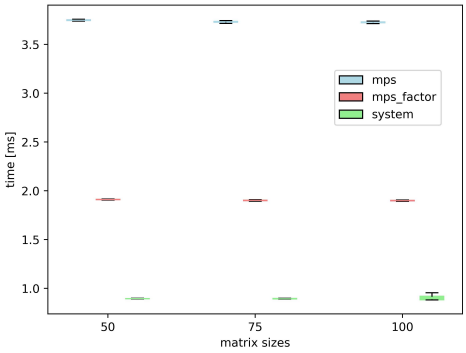
The first algorithm to be compared is Matrix Vector Multiplication (MVM). Since MVM runs way faster using natively implemented data types, 10,000 repetitions are needed to get satisfyingly large time measurements, compared to only one repetition for the simulator. Figure 5.7a shows the achieved speedup factors when comparing *single* and *double*. Unfortunately, what can be seen by studying Figure 5.7a is that there is almost no speedup when using the data types provided by the system. Also, the simulator's vanilla results are way too high compared to the results from Section 3.5. Nevertheless, when scaling the simulator's results by α, a speedup reasonably close to the ones depicted in Section 3.5 is visible. Further, note that the boxplots are very narrow, which indicates stable results.

Matrix Matrix Multiplication

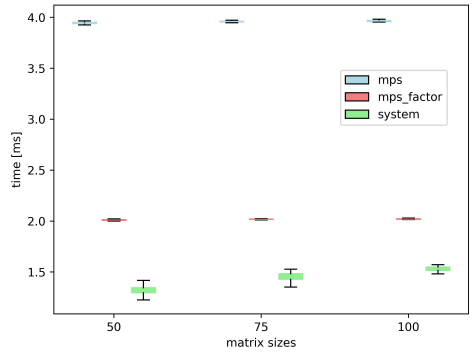
The next algorithm which is compared is Matrix Matrix Multiplication (MMM). Since MMM needs much more computation time for a given matrix size n than MVM, for this experiment, only 200 repetitions were performed when using system data types. The repetitions for the simulator cannot be decreased further, so the number and size of the evaluated matrices also needed to be reduced. Results are shown in Figure 5.7b. Again, as seen for the MVM, the system does not show any speedup. It almost shows a reduction in performance when switching from *double* to *float*, which seems odd and cannot be explained. It might be due to a programming error, but rest assured that the



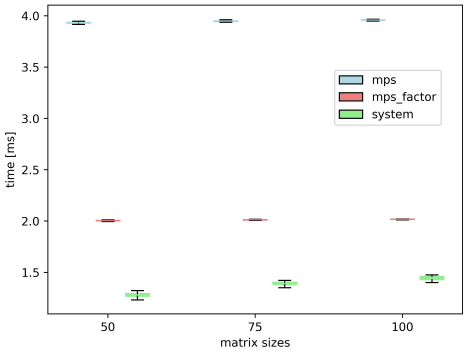
(a) Speedup factors for MVM. $n = 40$



(b) Speedup factors for MMM. $n = 40$



(c) Speedup factors for *PLU factorization*. $n = 40$



(d) Speedup factors for GESV using *PLU factorization*. $n = 40$

Figure 5.7.: Comparison of speedup factors for different algorithms.

implementation was more than triple-checked. Furthermore, as also seen for the MVM, the vanilla results most likely overestimate the speedup by about 3.5×. Conversely, when adjusted by α, the result shows a speedup of about 2×, which is far closer to what ought to be expected when reading through the findings presented in Section 3.5.

PLU Factorization

In contrast to the already discussed algorithms of MVM and MMM, when looking at Figure 5.7c *PLU factorization*, shows a decent speedup of about 1.5× when using FP formats as provided by the system. This result is probably still a bit low but certainly in an acceptable range. The fact that *PLU factorization* shows a speedup in contrast to MVM and MMM could be explained by it being more complex and the compiler or FPU has problems optimizing it. But this cannot be said with certainty and is more a guess than anything else. In contrast, the speedup provided by the simulator is about 4×,

which is a bit higher than before. The speedup given when adjusting by α also indicates that the simulator takes longer this time. In that case, the speedup is a bit higher than $2\times$, which is close to the findings in Section 3.5 but still probably a bit too high.

General Solve

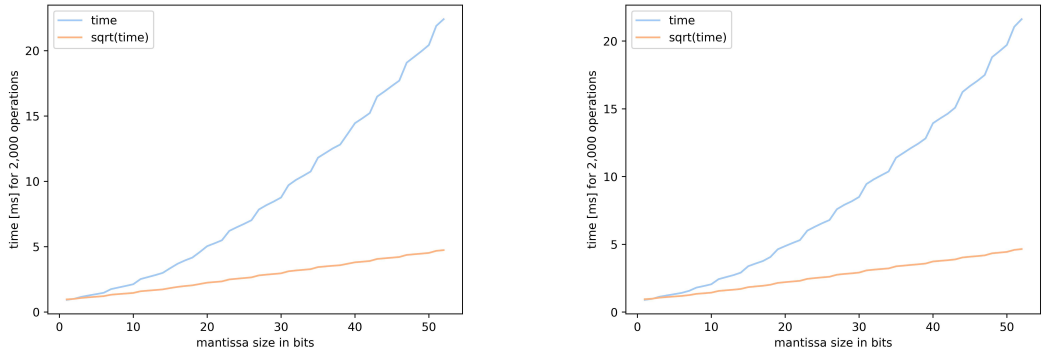
Some authors mentioned in Section 3.5 compare the results of the general solver using *double* precision with the results of *iterative refinement* using both *single* and *double* precision. This approach will not be addressed in this thesis since the MPS simulator is far too slow to be able to analyze large matrices, and the benefit of using *iterative refinement* is only visible when using it on large matrices. The reason for that is that $\mathcal{O}(n^3)$ is not that different from $\mathcal{O}(n^2)$ when working with small values of n. To still be able to perform an experiment that is at least similar, General Solve (GESV) using *single* precision will be compared to GESV using *double* precision. It is important to mention that this experiment is probably satisfactorily close to the original. They are the same except for the missing refinement steps when using the lower precision, which should not take long when working with large matrices. The assumption that there are not many refinement steps needed is supported by looking at the data provided by [46] and [47], where it can be found that normally only a very small number of steps are needed for convergence.

Nevertheless, the results for GESV look quite similar to the ones for *PLU factorization* and can be seen in Figure 5.7d. This does not come unexpectedly since most of the computational work done when using GESV is computing the *PLU factorization*. The only added complexity comes from *forward substitution* and *backward substitution*.

5.3.2. Comparisons: 2020 M1 MacBook Air

After examining the server's behavior, results will be studied when performing the experiments on a 2020 M1 MacBook Air. The basic setup is very similar to the one before. Nevertheless, the computational power of the MacBook used is certainly not as good as that of the server. Further multithreading will still be a part of the experiment, but only two threads can be used safely this time. When using more threads than that, observations have shown that the CPU sometimes defaults to simulate multithreading instead of actually performing it, which is certainly not as wanted. This means that the number of warmup runs was also decreased to two so that they again match the number of threads. Also, the number of repetitions had to be decreased, which means the quality of the results is certainly lower than before. Still, the results are interesting because the MacBook uses a different CPU than the server, making a valuable comparison.

At the beginning of the analysis, similar tests were performed to decide which compiler optimization level to choose. Interestingly, all optimization options seem valid this time, so all experiments were made for all levels. These investigations showed that there are indeed differences between them, and to show some of those, results are provided for *-O1* and *-O3* as depicted in Figure 5.9 and Figure 5.10. But before the results are going to be



(a) factor function graphs -*O1*

(b) factor function graphs -*O3*

Figure 5.8.: Graphs relevant for the factor functions used for the 2020 M1 MacBook Air.

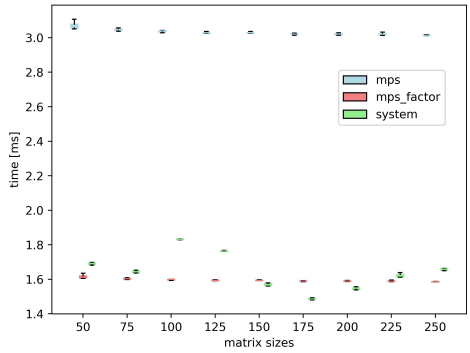
analyzed, as done before, the function $\alpha(m)$ needs to be derived for each optimization level presented. Figure 5.8 shows the two derived functions which are needed. To generate the data to derive the functions $\alpha(m)$, this time, a total of 204 *multiplication* runs were made, from which the first four were cut off as a warmup. The numbers used in the experiments were chosen randomly in the same fashion as in Section 5.3.1. Despite the differences in magnitude, the newly generated functions seem to be similar compared to the one seen for the server in Figure 5.6.

Matrix Vector Multiplication

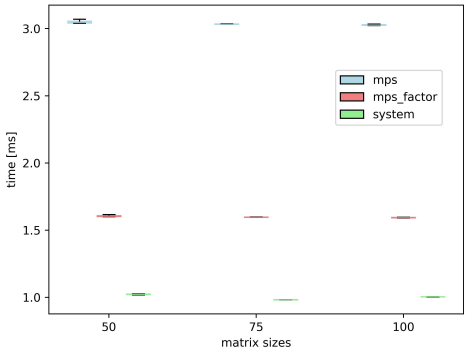
The results for MVM do not seem overly dependent on the optimization level, as can be seen when looking at Figure 5.9a and Figure 5.10a. When comparing it to the results from the server depicted in Figure 5.7a, differences appear visible. First, in contrast to when using the server on the MacBook, a speedup is visible when using the data types provided by the system. Also, the simulator appears to perform differently depending on the platform since the speedup is lower than the one observed before. Further, when adjusting the result using α, the results seem to match decently well with the numbers resulting when using system data types. A good correlation can also be ascertained with the findings depicted in Section 3.5.

Matrix Matrix Multiplication

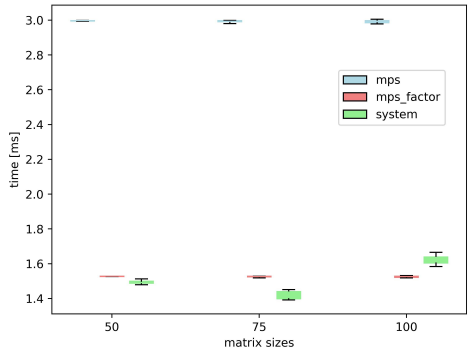
MMM shows interesting differences when comparing the plots. In contrast to the findings from the server (Figure 5.7b) or the MacBook, when using -*O1* (Figure 5.9b) Figure 5.10b shows a visible speedup of about 1.25×, at least for the one matrix with size $n = 50$. Since it seemed odd that the speedup was only noticeable for $n = 50$, the experiment was redone, but similar results were achieved. Nevertheless, now that at least some differences



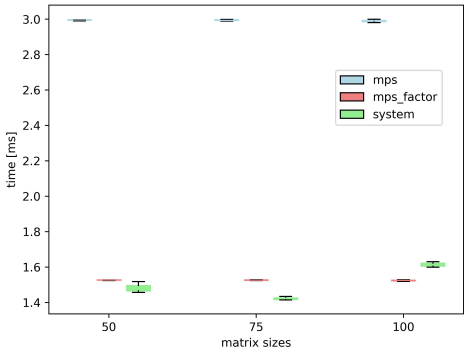
(a) Speedup factors for MVM. $n = 10$



(b) Speedup factors for MMM. $n = 10$



(c) Speedup factors for *PLU factorization*. $n = 20$



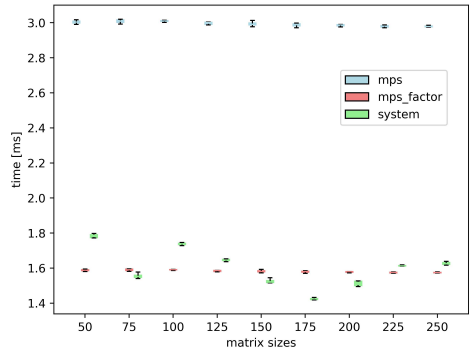
(d) Speedup factors for GESV using *PLU factorization*. $n = 20$

Figure 5.9.: Comparison of speedup factors for different algorithms on a 2020 M1 MacbookAir using *-O1*.

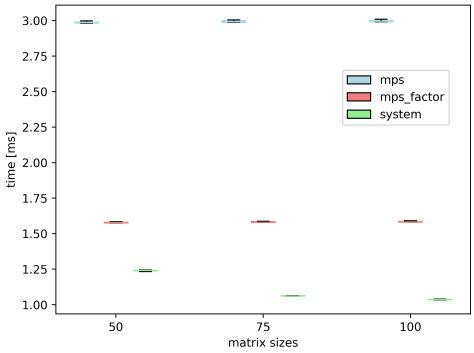
between *single* and *double* are visible, it is less likely that the absence of any speedup results from a programming mistake. Nevertheless, it could not yet be explained why MMM and MVM differ so much since the algorithm is quite similar. However, signs of this similarity can be seen when comparing the results from the simulator since the results from MMM and MVM are fairly close. When comparing the speedup factors for MMM to the ones obtained from the server, they seem to be a bit lower than when using the MacBook, as was already observed for MVM.

PLU Factorization and General Solve

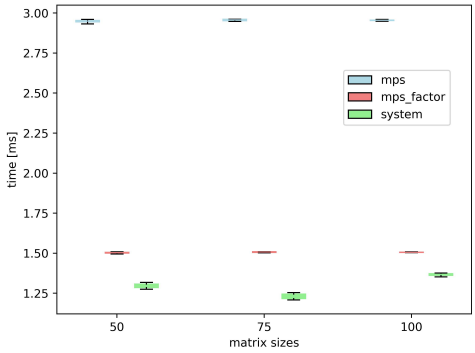
Due to the fact that the results for the two are so similar, they will be handled together. The speedup factors of the two when using *-O1* appear to be far more pronounced than



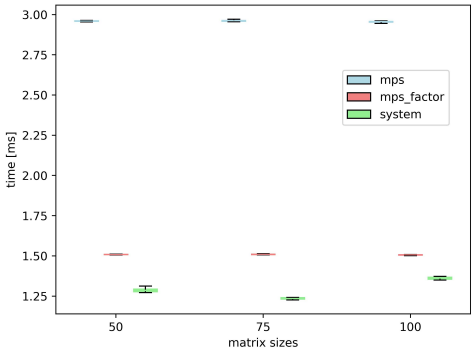
(a) Speedup factors for MVM. $n = 10$



(b) Speedup factors for MMM. $n = 10$



(c) Speedup factors for *PLU factorization*. $n = 20$



(d) Speedup factors for GESV using *PLU factorization*. $n = 20$

Figure 5.10.: Comparison of speedup factors for different algorithms on a 2020 M1 MacbookAir using *-O3*.

when using *-O3*. This could be explained because the compiler uses more optimization when using *-O3*, dampening the effects of *multiprecision*. Interestingly, the server still behaves like the MacBook when using *-O1* despite actually using *-O3*. The different CPUs might explain this since different compilers need to be used, which likely have different optimization options. Nevertheless, the speedup factors of the simulator again match up satisfactorily with the results presented in 3.5.

5.4. Application Example of the Simulator

In the last sections, the simulator's performance was compared to results obtained from the literature and using the same code but with data types provided by the system. Now,

this chapter shall briefly show how the simulator can be applied. Since the server is now being used again, it is possible to use 20 threads as before. In total, 60 runs were made for each experiment, including 20 runs as a warmup. The resulting 40 runs were reduced by applying the average. As an example, *iterative refinement* in three precisions will be used to demonstrate the simulator. Two different approaches were chosen to do this. For the first one, the number of refinement iterations will be fixed. After each refinement step, the execution is halted to make measurements, at which the current relative error is saved. Ultimately, all those sub-results are summed up, giving an approximate integral that can measure how well the current algorithm converges to the correct solution. Until now, all this could be accomplished by using a simplistic *chop* function as depicted in [55] as well. But of course, just summing up the errors alone will not give good insights since using a higher precision should always converge faster than when using a lower precision. However, the simulator described in this thesis does provide performance differences between data types, which shall be used in this analysis. The previously described sum is multiplied by the algorithm's runtime to integrate this into the analysis. Therefore, if an algorithm converges quickly by utilizing FP formats with a large precision, it is also penalized because of the longer time requirement. The resulting analysis can be found in Figure 5.11.

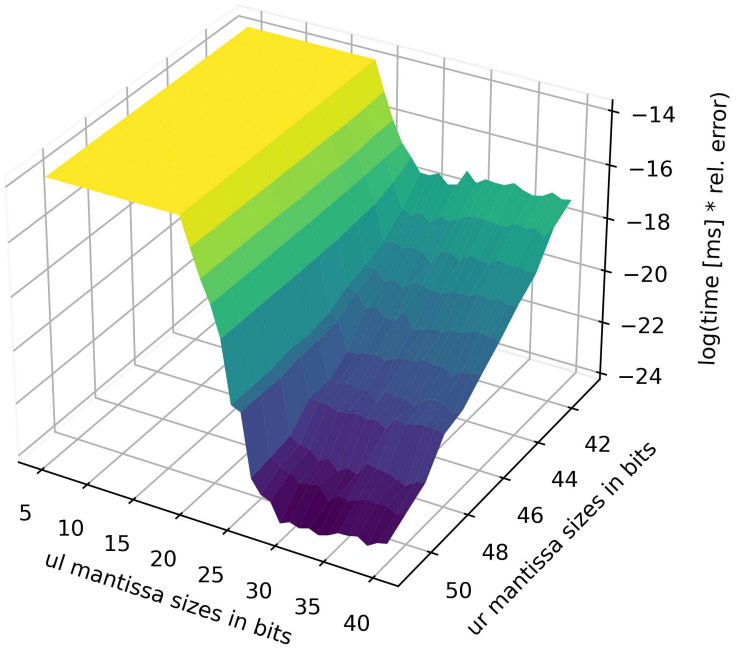
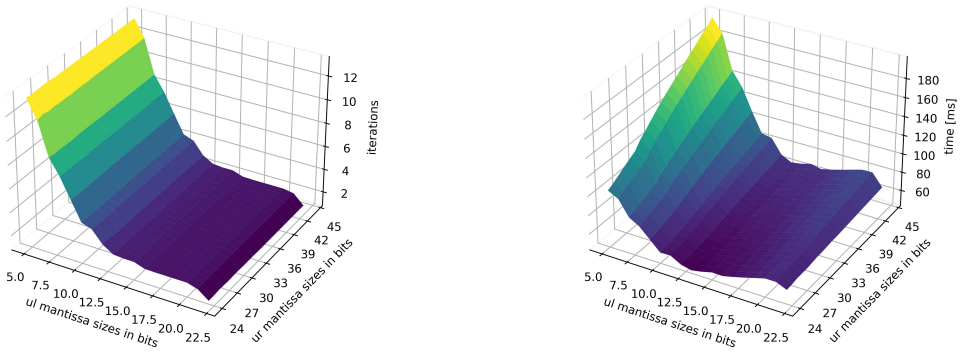


Figure 5.11.: Performance analysis of *iterative refinement* using MPS and the relative error.



(a) Analysis using the number of needed iterations.

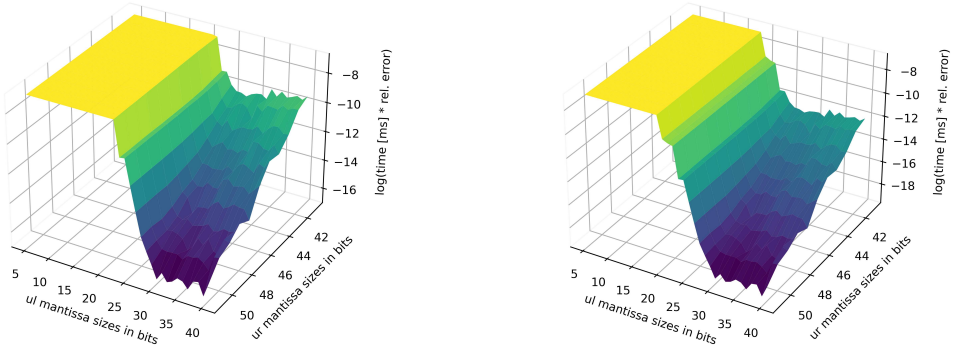
(b) Analysis using the runtime.

Figure 5.12.: Performance analysis of *iterative refinement* using MPS and a wanted relative error.

To generate the plane shown in Figure 5.11 the working precision u was fixed to $\mathrm{F}(41, 11)$, while $u_l \in \{x \in \mathrm{F}(m, 11) \mid 5 \leq m \leq 40\}$ and $u_l \in \{x \in \mathrm{F}(m, 11) \mid 41 \leq m \leq 51\}$. Please be aware that data above a certain maximum value was set to that maximum, resulting in the visible yellow plane on the left side of the axis representing u_l. Further, note that the values on the z-axis are negative because of the application of the *logarithm*. When looking at the graph of Figure 5.11, the use case of the simulator becomes imminent. Without much work, it is possible to see that increasing the precision of u_r definitely impacts performance. Moreover, it can be observed that the optimal *mantissa* size for u_l is probably around 30. It might not be evident at first glance, but the z-value does increase again for *mantissa* sizes above 30. These are valuable results since, despite the inability of the simulator to mirror real computers completely, the behavior of different data types can be deduced.

The second approach for analyzing *iterative refinement* is defining a wanted maximal relative error and stopping the refinement process after the current relative error of x falls below this maximal error. The results are depicted in Figure 5.12, and one can see quickly where the benefits of the simulator lie. In Figure 5.12a, the number of iterations is taken as a performance measurement. Again, this could be achieved using a function as *chop* as described in [55]. However, when using the capabilities of the simulator, as was done in Figure 5.12b, one sees immediately that a higher precision in u_r might not add certain benefits.

But wait, does this not contradict the findings from Figure 5.11? Well, yes and no. On the one hand, these results are contradictory, but please be aware that the wanted error, which is set beforehand, introduces some hard limit. The current error is either above this line or below. Unfortunately, in this particular analysis, the impact of the precision of u_r is not high enough to decrease the number of iterations needed until this barrier is reached.



(a) without transformation

(b) with transformation

Figure 5.13.: Performance analysis using multiple time measurements with and without the transformation using MPS.

Keep in mind that Figure 5.11 is also transformed by the *logarithm*, and therefore, the effect of varying u_r is unnoticeable in real live applications. So, as shown in Figure 5.11, it is true that the precision of u_r does influence convergence, but it is probably not relevant enough to have an impact in other experimental settings as shown in Figure 5.12b. Also, for generating Figure 5.12, slightly different settings were chosen. For instance the working precision u was fixed to $\mathrm{F}(23, 11)$, while $u_l \in \{x \in \mathrm{F}(m, 11) \mid 5 \leq m \leq 22\}$ and $u_l \in \{x \in \mathrm{F}(m, 11) \mid 24 \leq m \leq 44\}$, so this might also have an influence. Nevertheless, both approaches show the benefits of a simulator as designed in this thesis when wanting to analyze *multiprecision* algorithms

The results above are not altered by applying the factor α as mentioned above. Unfortunately, in this setting, it is not as easy as before, and it is not feasible to multiply all results by one factor because now, not only one precision format is applied, but three different ones. So, for instance, multiplying by one scalar would correct the findings for one data type but simultaneously falsify the effects of the others. Therefore, three different scalars are needed, but with that, a problem arises: with only a single time measurement, all three factors cannot be applied suitably. To address this problem, the experiment is slightly altered so that three different time measurements will be present at the end, namely one for u_l, u, and u_r, which could be accomplished by taking separate time measurements for each step followed by a summation. The information of which step of the algorithm corresponds to which precision can be found in the depiction of Algorithm 4. Now, with three distinct time measurements, applying the different factors is easy. The results when adapting the first approach are shown in Figure 5.13.

Before interpreting the results, note that this approach needs more computation time. The reason for this is because the many individual time measurements introduce a certain

overhead. Also, when comparing Figure 5.13b to Figure 5.13a the plane does not look overly different. The major distinction is that the values have a larger magnitude than in Figure 5.13a, which is easily explained by the usage of the logarithm in combination with the transformation which is lowering the values of the results. The fact that the plots are similar potentially indicates that the transformation may not even be needed if the method is used in certain settings, such as optimization problems. This statement is further supported when comparing Figure 5.11, Figure 5.13a and Figure 5.13b with one another, because in all of them an optimal *mantissa* length of about 30 bits is predicted.

6. Discussion

The first thing that shall be mentioned is the often-occurring inability of the used hardware to show computational improvements when using lower-precision formats. As can be seen from Figure 5.4, the computation time does not change when using different formats, which is probably because the hardware is optimized in a way such that all of these instructions have similar time requirements. Problems like this can be seen when looking at the results from Matrix Matrix Multiplication, where almost all results show no speedup when switching from *double* to *single* precision. On the other hand, differences appear to be visible for algorithms like Matrix Vector Multiplication. Perhaps these findings can be explained by the fact a modern computer not only performs both *single* and *double* precision in one clock circle but multiple of such instructions due to parallelization [59, 89]. Nevertheless, such simultaneous execution sometimes also depends on the mixture of instructions, which might explain certain differences. However, interestingly, Matrix Vector Multiplication and Matrix Matrix Multiplication reveal different speedups despite being relatively similar, so the actual reason is most likely different.

The fact that modern-day CPUs are optimized imposes a problem when wanting to replicate their behavior. For instance, Booth's algorithm [11], as it is implemented in the simulator, has a *time complexity* of $\mathcal{O}(n^2)$. Still on a CPU, *multiplication* appears to be realized in linear or even constant *time complexity* by employing parallelization [89]. Because of this, the speedup found when using the simulator is naturally far greater than when running the same experiment directly on hardware. To overcome this problem, a model-based correction was introduced, as depicted in Section 4.4, which resulted in noticeable improvements. However, the mathematical analysis indicates that the error introduced by this approach can be as significant as the combined time required for all *additions* and *subtractions*. The reason for this is that after transforming according to the model, on the one hand, *multiplication* and *division* now might appear as having linear *time complexity*, but on the other hand, the process simultaneously impacts *addition* and *subtraction*, which already exhibit nearly linear *time complexity*. It is also important to mention that this mathematical approach is very simplistic and by no means aims to emulate an actual CPU. The purpose is merely to show that mathematical transformations could have the ability to replicate results similar to a modern CPU.

Nevertheless, the results can also be interpreted from another viewpoint. Certainly, when focusing on the CPU, all operations should have about linear *time complexity*. But, for instance, Booth's Algorithm does, in fact, have $\mathcal{O}(n^2)$ *time complexity*, and the implementations on CPUs do not fundamentally change that but make improvements by utilizing parallelization. For instance, *multiplication* can be enhanced by connecting

multiple adders in series [89]. Such an approach obviously reduces the computation time but also increases the space needed on the CPU. Space that otherwise could have been used to accelerate other logic. So, in the end, it might be beneficial to examine the unaltered results from the simulator, as they could provide more authentic insights. Insights that could potentially be used to design optimized CPUs.

After all this, the simulator's practical applications come into focus. While the unaltered result from the simulator can help to decide which data types to implement on a CPU, using the simulator with transformed results could help find optimal data types for algorithms on specified hardware. For both of the described causes, the results from the simulator might not be the most accurate. Still, they definitely serve as a good first impression of how different data types affect the runtime of an arbitrary algorithm.

6.1. Further Work

A multitude of avenues exist to change or optimize the simulator. For instance, as of now, the simulator can only perform an operation like *addition* when both inputs have exactly the same FP format. Such constraints made sense for a prototype since it, on the one hand, resembles the mechanics of the real world [44] and, on the other hand, also reduces the complexity of the simulator. However, this restriction could be loosened so that the operation can handle inputs of different sizes, likely accelerating the simulator.

Another probable improvement would be to extend the simulator, possibly for integer formats. This is because when analyzing algorithms using the simulator, integer data types must be implemented as provided by the system since the proposed simulator does not yet support classical integer formats. Because of that, they run much faster than they should compared to the data types provided by the simulator. With more simulators available for other formats, analyses could be extended, which would lead to more insights.

Furthermore, the approach used in this thesis was to write a simulator using a classic programming language like C++. Therefore, all algorithms were implemented in a relatively basic way to have a high chance of closely replicating a large variety of hardware. However, it could also be tried to replicate the architecture of one specific CPU, which could improve the results – at least for the CPU architecture in question. Nevertheless, if such an approach is possible, it would be far more complex than the current implementation.

Lastly, although the simulator is already fairly simple to use, improvements to its usability can always be made. This would further ease the usability for new users and maybe even make the simulator a good teaching tool. For the latter, it would also not be of big concern that the simulator does not behave like a normal CPU since it still shows the differences between data types.

7. Conclusion

At the beginning of this thesis, numerous papers were referenced showcasing the importance of *multiprecision* algorithms and the common problems faced in this field. One of them is that a common CPU only provides a very limited number of different data types. In the case of FPN, this typically only includes *single* and *double* precision. Techniques were developed to circumvent this by using software for the simulation. Despite such simulators working reasonably well, they lack the capability to mirror the individual performance differences between different precision formats. To overcome this problem, this thesis proposed an *IEEE 754 standard* conform and efficient simulator that does account for such performance differences and is implemented in C++.

The basic idea was to internally represent all FP numbers using an `std::vector<bool>` object and implement all necessary operations for *IEEE 754 standard* [57] conform FP calculations on top of it. The simulator's conformity to the *IEEE 754 standard* was verified through statistical methods. However, using a vector of boolean values for the implementation inevitably has a strong influence on the simulator's performance, which necessitates a focus on optimization. In the end, according to Figure 5.2d, the simulator was able to perform a bit less than $100,000$ *multiplications* in one second on the used server when compiling with *-O3*. This already showcases one shortcoming of the simulator since it is certainly not feasible for calculations involving larger matrices.

Another downside of the algorithm might be that it does not exactly replicate the behavior of a typical CPU since the latter utilizes optimizations that cannot be replicated efficiently in code. The main problem comes with *multiplication* and *division* since they are implemented having a *time complexity* of $\mathcal{O}(n^2)$ inside the simulator. This fact is false for a normal CPU due to optimizations [89]. However, using parallelism, which reduces the *time complexity*, simultaneously increases the area needed on the CPU. Because this space could otherwise be used for other logic, the simulator can potentially provide a more unbiased analysis.

Despite this, the ability to reproduce results similar to that of a CPU is still very important. Therefore, a straightforward mathematical solution was introduced, which tried to correct the outcomes in favor of results closer to those of a CPU. This transformation mainly involves the *square root* to counteract the quadratic behavior of *multiplication* and *division*. Despite this also affecting other parts of the simulator like *addition* and *subtraction*, results near to that of a CPU were obtained. This means that the simulator can potentially be used to analyze FP behavior on existing CPUs. However, it by no means aims to replicate the exact behavior of an actual modern CPU.

Bibliography

[1] A. Abdelfattah et al. 'A Survey of Numerical Linear Algebra Methods Utilizing Mixed-Precision Arithmetic'. In: *The International Journal of High Performance Computing Applications* 35.4 (2021), pp. 344–369.

[2] P. Amestoy et al. 'Combining Sparse Approximate Factorizations with Mixed-precision Iterative Refinement'. In: *ACM Transactions on Mathematical Software* 49.1 (2023), 4:1–4:29.

[3] P. Amestoy et al. 'Five-Precision GMRES-Based Iterative Refinement'. In: *SIAM Journal on Matrix Analysis and Applications* (2021), pp. 1–24.

[4] M. Andrysco et al. 'On Subnormal Floating Point and Abnormal Timing'. In: 2015 IEEE Symposium on Security and Privacy. 2015, pp. 623–639.

[5] F. Attarzadeh and E. Nagler. 'Understanding and Using Cast Operations in C and C++'. In: *ASEE Computers in Education Journal* (2020), pp. 6–22.

[6] D. H. Bailey. *MPFUN2020: A Thread-Safe Arbitrary Precision Package with Special Functions (Full Documentation)*. 2024. URL: `https://www.davidhbailey.com/dhbpapers/mpfun2020.pdf` (visited on 21/06/2024).

[7] D. H. Bailey. *Software Directory*. URL: `https://www.davidhbailey.com/dhbsoftware/` (visited on 13/07/2024).

[8] Á. Baráth and Z. Porkoláb. 'Life without Implicit Casts: Safe Type System in C++'. In: Proceedings of the 7th Balkan Conference on Informatics Conference. BCI '15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 1–4.

[9] D. A. Bini and L. Robol. 'Solving Secular and Polynomial Equations: A Multiprecision Algorithm'. In: *Journal of Computational and Applied Mathematics* 272 (2014), pp. 276–292.

[10] Å. Björck. 'Iterative Refinement of Linear Least Squares Solutions I'. In: *BIT Numerical Mathematics* 7.4 (1967), pp. 257–278.

[11] A. D. Booth. 'A Signed Binary Multiplication Technique'. In: *Quarterly Journal of Mechanics and Applied Mathematics* 4 (1951), pp. 236–240.

[12] C. Brugger et al. 'Mixed Precision Multilevel Monte Carlo on Hybrid Computing Systems'. In: 2014 IEEE Conference on Computational Intelligence for Financial Engineering & Economics (CIFEr). 2014, pp. 215–222.

[13] A. Buttari et al. 'Mixed Precision Iterative Refinement Techniques for the Solution of Dense Linear Systems'. In: *International Journal of High Performance Computing Applications* 21 (2007), pp. 1–16.

[14] A. Buttari et al. 'Using Mixed Precision for Sparse Matrix Computations to Enhance the Performance while Achieving 64-bit Accuracy'. In: *ACM Transactions on Mathematical Software* 34.4 (2008), 17:1–17:22.

[15] Z. Cai and N. Vasconcelos. 'Rethinking Differentiable Search for Mixed-Precision Neural Networks'. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2020, pp. 2349–2358.

[16] E. Carson, T. Gergelits and I. Yamazaki. 'Mixed Precision S-Step Lanczos and Conjugate Gradient Algorithms'. In: *Numerical Linear Algebra with Applications* 29.3 (2022), pp. 1–34.

[17] E. Carson and N. J. Higham. 'A New Analysis of Iterative Refinement and Its Application to Accurate Solution of Ill-Conditioned Sparse Linear Systems'. In: *SIAM Journal on Scientific Computing* 39.6 (2017), A2834–A2856.

[18] E. Carson and N. J. Higham. 'Accelerating the Solution of Linear Systems by Iterative Refinement in Three Precisions'. In: *SIAM Journal on Scientific Computing* 40.2 (2018), A817–A847.

[19] M. Chantry et al. 'Scale-Selective Precision for Weather and Climate Forecasting'. In: *Monthly Weather Review* 147.2 (2019), pp. 645–655.

[20] G. Chen, L. Chacón and D. C. Barnes. 'An Efficient Mixed-Precision, Hybrid CPU–GPU Implementation of a Nonlinearly Implicit One-Dimensional Particle-In-Cell Algorithm'. In: *Journal of Computational Physics* 231.16 (2012), pp. 5374–5388.

[21] Z. Chen. 'Algorithm-Based Recovery for Iterative Methods without Checkpointing'. In: Proceedings of the 20th International Symposium on High Performance Distributed Computing. HPDC '11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 73–84.

[22] G. C. T. Chow et al. 'A Mixed Precision Monte Carlo Methodology for Reconfigurable Accelerator Systems'. In: Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays. FPGA '12. New York, NY, USA: Association for Computing Machinery, 2012, pp. 57–66.

[23] M. A. Clark et al. 'Solving Lattice QCD Systems of Equations Using Mixed Precision Solvers on GPUs'. In: *Computer Physics Communications* 181.9 (2010), pp. 1517–1528.

[24] *CMake - Upgrade Your Software Build System*. URL: `https://cmake.org/` (visited on 10/06/2024).

[25] M. Courbariaux, Y. Bengio and J.-P. David. 'Training Deep Neural Networks with Low Precision Multiplications'. In: *arXiv preprint* (2015), pp. 1–10. URL: `http://arxiv.org/abs/1412.7024` (visited on 31/01/2024).

[26] D. Das et al. 'Mixed Precision Training of Convolutional Neural Networks using Integer Operations'. In: *arXiv preprint* (2018), pp. 1–11. URL: http://arxiv.org/abs/1802.00930 (visited on 25/04/2024).

[27] S. Das et al. 'Fast, Scalable and Accurate Finite-Element Based Ab Initio Calculations Using Mixed Precision Computing: 46 PFLOPS Simulation of a Metallic Dislocation System'. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 1–11.

[28] P. I. Davies, N. J. Higham and F. Tisseur. 'Analysis of the Cholesky Method with Iterative Refinement for Solving the Symmetric Definite Generalized Eigenproblem'. In: *SIAM Journal on Matrix Analysis and Applications* 23.2 (2001), pp. 472–493.

[29] T. A. Davis and Y. Hu. 'The University of Florida Sparse Matrix Collection'. In: *ACM Transactions on Mathematical Software* 38.1 (2011), 1:1–1:25.

[30] A. Dawson and P. D. Düben. 'rpe v5: An Emulator for Reduced Floating-Point Precision in Large Numerical Simulations'. In: *Geoscientific Model Development* 10.6 (2017), pp. 2221–2230.

[31] J. Demmel et al. 'Extra-Precise Iterative Refinement for Overdetermined Least Squares Problems'. In: *ACM Transactions on Mathematical Software* 35.4 (2009), 28:1–28:32.

[32] Z. Dong et al. 'HAWQ: Hessian AWare Quantization of Neural Networks With Mixed-Precision'. In: Proceedings of the IEEE/CVF International Conference on Computer Vision. 2019, pp. 293–302.

[33] I. Dooley, L. Kalé and N. Goodwin. 'Quantifying the Interference Caused by Subnormal Floating-Point Values'. In: *Semantic Scholar proceeding* (2006), pp. 1–7.

[34] P. D. Düben, H. McNamara and T. N. Palmer. 'The Use of Imprecise Processing to Improve Accuracy in Weather & Climate Prediction'. In: *Journal of Computational Physics*. Frontiers in Computational Physics 271 (2014), pp. 2–18.

[35] K. Dunnigan and S. Consulting. 'Confidence Interval Calculation for Binomial Proportions'. In: *MidWest SAS Users Group* (2008), pp. 1–12.

[36] B. Fang et al. 'MPGemmFI: A Fault Injection Technique for Mixed Precision GEMM in ML Applications'. In: *arXiv preprint* (2023), pp. 1–13. URL: http://arxiv.org/abs/2311.05782 (visited on 17/11/2023).

[37] M. Fasi and M. Mikaitis. 'CPFloat: A C Library for Simulating Low-precision Arithmetic'. In: *ACM Transactions on Mathematical Software* 49.2 (2023), 18:1–18:32.

[38] G. Forsythe, M. Malcolm and C. Moler. *Computer Methods for Mathematical Computations*. 1977.

[39] L. Fousse et al. 'MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding'. In: *ACM Transactions on Mathematical Software* 33.2 (2007), pp. 1–15.

[40] M. Godbolt. 'Optimizations in C++ Compilers'. In: *Communications of the ACM* 63.2 (2020), pp. 41–49.

[41] D. Göddeke, R. Strzodka and S. Turek. 'Performance and Accuracy of Hardware-oriented Native-, Emulated- and Mixed-precision Solvers in FEM Simulations'. In: *International Journal of Parallel, Emergent and Distributed Systems* 22.4 (2007), pp. 221–256.

[42] W. Govaerts and J. D. Pryce. 'Block Elimination with one Refinement Solves Bordered Linear Systems Accurately'. In: *BIT Numerical Mathematics* 30.3 (1990), pp. 490–507.

[43] A. Greenbaum. 'Estimating the Attainable Accuracy of Recursively Computed Residual Methods'. In: *SIAM Journal on Matrix Analysis and Applications* 18.3 (1997), pp. 535–551.

[44] R. Grims. *Beginning C++ Programming*. Packt Publishing Ltd., 2017.

[45] M. Gulliksson. 'Iterative Refinement for Constrained and Weighted Linear Least Squares'. In: *BIT Numerical Mathematics* 34.2 (1994), pp. 239–253.

[46] A. Haidar et al. 'Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers'. In: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis. 2018, pp. 603–613.

[47] A. Haidar et al. 'Mixed-Precision Iterative Refinement Using Tensor Cores on GPUs to Accelerate Solution of Linear Systems'. In: *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 476.2243 (2020), pp. 1–30.

[48] A. Haidar et al. 'The Design of Fast and Energy-Efficient Linear Solvers: On the Potential of Half-Precision Arithmetic and Iterative Refinement Techniques'. In: Computational Science – ICCS 2018. Ed. by Y. Shi et al. Cham: Springer International Publishing, 2018, pp. 586–600.

[49] S. Hatfield et al. 'Accelerating High-Resolution Weather Models with Deep-Learning Hardware'. In: PASC '19: Platform for Advanced Scientific Computing Conference. Zurich Switzerland: ACM, 2019, pp. 1–11.

[50] S. Hatfield et al. 'Improving Weather Forecast Skill through Reduced-Precision Data Assimilation'. In: *Monthly Weather Review* 146.1 (2018), pp. 49–62.

[51] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics, 2002.

[52] N. J. Higham. 'Iterative Refinement Enhances the Stability of QR Factorization Methods for Solving Linear Equations'. In: *BIT Numerical Mathematics* 31.3 (1991), pp. 447–468.

[53] N. J. Higham. 'Iterative Refinement for Linear Systems and LAPACK'. In: *IMA Journal of Numerical Analysis* 17.4 (1997), pp. 495–509.

[54] N. J. Higham and T. Mary. 'Mixed Precision Algorithms in Numerical Linear Algebra'. In: *Acta Numerica* 31 (2022), pp. 347–414.

[55] N. J. Higham and S. Pranesh. 'Simulating Low Precision Floating-Point Arithmetic'. In: *SIAM Journal on Scientific Computing* 41.5 (2019), pp. C585–C602.

[56] N. J. Higham, S. Pranesh and M. Zounon. 'Squeezing a Matrix into Half Precision, with an Application to Solving Linear Systems'. In: *SIAM Journal on Scientific Computing* 41.4 (2019), A2536–A2551.

[57] 'IEEE Standard for Floating-Point Arithmetic'. In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), pp. 1–84.

[58] Intel Corporation. *bfloat16 - Hardware Numerics Definition*. White Paper. Document number 338302-001US. 2018. URL: `https://www.intel.com/content/www/us/en/content-details/671279/bfloat16-hardware-numerics-definition.html` (visited on 27/03/2024).

[59] Intel Corporation. *Intel®64 and IA-32 Architectures Optimization Reference Manual*. URL: `https://www.intel.com/content/www/us/en/content-details/671488/intel-64-and-ia-32-architectures-optimization-reference-manual-volume-1.html` (visited on 03/01/2024).

[60] T. Iwashita, K. Suzuki and T. Fukaya. 'An Integer Arithmetic-Based Sparse Linear Solver Using a GMRES Method and Iterative Refinement'. In: 2020 IEEE/ACM 11th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA). 2020, pp. 1–8.

[61] W. Jakob, J. Rhinelander and D. Moldovan. *pybind11 – Seamless Operability between C++11 and Python*. 2017. URL: `https://github.com/pybind/pybind11` (visited on 30/03/2023).

[62] M. Jankowski and H. Woźniakowski. 'Iterative Refinement Implies Numerical Stability'. In: *BIT Numerical Mathematics* 17.3 (1977), pp. 303–311.

[63] A. Kiełbasiński. 'Iterative Refinement for Linear Systems in Variable-Precision Arithmetic'. In: *BIT Numerical Mathematics* 21.1 (1981), pp. 97–103.

[64] J. Kurzak and J. Dongarra. 'Implementation of Mixed Precision in Solving Systems of Linear Equations on the Cell Processor'. In: *Concurrency and Computation: Practice and Experience* 19.10 (2007), pp. 1371–1385.

[65] J. Langou et al. 'Exploiting the Performance of 32 Bit Floating Point Arithmetic in Obtaining 64 Bit Accuracy (Revisiting Iterative Refinement for Linear Systems)'. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing. SC '06. New York, NY, USA: Association for Computing Machinery, 2006, pp. 1–17.

[66] S. Le Grand, A. W. Götz and R. C. Walker. 'SPFP: Speed without Compromise—A Mixed Precision Model for GPU Accelerated Molecular Dynamics Simulations'. In: *Computer Physics Communications* 184.2 (2013), pp. 374–380.

[67] J. K. Lee and G. D. Peterson. 'Iterative Refinement on FPGAs'. In: 2011 Symposium on Application Accelerators in High-Performance Computing. ISSN: 2166-515X. 2011, pp. 8–13.

[68] J. Lee et al. 'AIR: Iterative Refinement Acceleration using Arbitrary Dynamic Precision'. In: *Parallel Computing* 97 (2020), pp. 1–15.

[69] J. Lee et al. 'Energy-Efficient Iterative Refinement Using Dynamic Precision'. In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 8.4 (2018), pp. 722–735.

[70] V. Lefevre and P. Zimmermann. 'Optimized Binary64 and Binary128 Arithmetic with GNU MPFR'. In: 2017 IEEE 24th Symposium on Computer Arithmetic (ARITH). London, United Kingdom: IEEE, 2017, pp. 18–26.

[71] G. Li et al. 'Understanding Error Propagation in Deep Learning Neural Network (DNN) Accelerators and Applications'. In: SC '17: The International Conference for High Performance Computing, Networking, Storage and Analysis. Denver Colorado: ACM, 2017, pp. 1–12.

[72] G. Liang. 'Arithmetic Logic Unit Architectures With Dynamically Defined Precisions'. PhD thesis. University of Tennessee, 2015.

[73] G. Liang, J. Lee and G. D. Peterson. 'ALU Architecture with Dynamic Precision Support'. In: 2012 Symposium on Application Accelerators in High Performance Computing. 2012, pp. 26–33.

[74] S. F. McCormick, J. Benzaken and R. Tamstorf. 'Algebraic Error Analysis for Mixed-Precision Multigrid Solvers'. In: *SIAM Journal on Scientific Computing* 43.5 (2021), S392–S419.

[75] J. A. Meijerink and H. A. Van Der Vorst. 'An Iterative Solution Method for Linear Systems of which the Coefficient Matrix is a Symmetric M-Matrix'. In: *Mathematics of Computation* 31.137 (1977), pp. 148–162.

[76] A. Meurer et al. 'SymPy: Symbolic Computing in Python'. In: *PeerJ Computer Science* 3 (2017), pp. 1–27.

[77] P. Micikevicius et al. 'Mixed Precision Training'. In: *arXiv preprint* (2018), pp. 1–12. URL: `http://arxiv.org/abs/1710.03740` (visited on 25/04/2024).

[78] C. B. Moler. 'Iterative Refinement in Floating Point'. In: *Journal of the ACM* 14.2 (1967), pp. 316–321.

[79] M. Nakata. 'MPLAPACK Version 2.0.1 User Manual'. In: *arXiv preprint* (2022), pp. 1–136. URL: `http://arxiv.org/abs/2109.13406` (visited on 13/07/2024).

[80] S. R. Nandakumar et al. 'Mixed-Precision Architecture Based on Computational Memory for Training Deep Neural Networks'. In: 2018 IEEE International Symposium on Circuits and Systems (ISCAS). 2018, pp. 1–5.

[81] T. Ogita and K. Aishima. 'Iterative Refinement for Symmetric Eigenvalue Decomposition'. In: *Japan Journal of Industrial and Applied Mathematics* 35.3 (2018), pp. 1007–1035.

[82] T. Ogita and K. Aishima. 'Iterative Refinement for Symmetric Eigenvalue Decomposition II: Clustered Eigenvalues'. In: *Japan Journal of Industrial and Applied Mathematics* 36.2 (2019), pp. 435–459.

[83] E. Oktay and E. Carson. 'Multistage Mixed Precision Iterative Refinement'. In: *Numerical Linear Algebra with Applications* 29.4 (2022), pp. 1–30.

[84] L. Ping, J. Tan and K. Yan. 'SERN: Modeling and Analyzing the Soft Error Reliability of Convolutional Neural Networks'. In: Proceedings of the 2020 on Great Lakes Symposium on VLSI. GLSVLSI '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 445–450.

[85] C. Rubio-González et al. 'Precimonious: Tuning Assistant for Floating-Point Precision'. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. SC '13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 1–12.

[86] R. D. Skeel. 'Iterative Refinement Implies Numerical Stability for Gaussian Elimination'. In: *Mathematics of Computation* 35.151 (1980), pp. 817–832.

[87] A. Smoktunowicz and J. Sokolnicka. 'Binary Cascades Iterative Refinement in Doubled-Mantissa Arithmetics'. In: *BIT Numerical Mathematics* 24.1 (1984), pp. 123–127.

[88] G. W. Stewart. *Introduction to Matrix Computations*. Academic Press, 1973.

[89] J. E. Stine. *Digital Computer Arithmetic Datapath Design Using Verilog HDL*. Boston, MA: Springer US, 2004.

[90] G. Strang. *Introduction to Linear Algebra*. 6th ed. Wellesley-Cambridge Press. Wellesley-Cambridge Press, 2022.

[91] J. Sun, G. D. Peterson and O. O. Storaasli. 'High-Performance Mixed-Precision Linear Solver for FPGAs'. In: *IEEE Transactions on Computers* 57.12 (2008), pp. 1614–1623.

[92] R. Tamstorf, J. Benzaken and S. F. McCormick. 'Discretization-Error-Accurate Mixed-Precision Multigrid Solvers'. In: *SIAM Journal on Scientific Computing* 43.5 (2021), S420–S447.

[93] D. Tan, A. Danysh and M. Liebelt. 'Multiple-Precision Fixed-Point Vector Multiply-Accumulator Using Shared Segmentation'. In: Proceedings 2003 16th IEEE Symposium on Computer Arithmetic. 2003, pp. 12–19.

[94] R. P. Tewarson. *Sparse Matrices*. Academic Press, 1973.

[95] The mpmath development team. *mpmath: A Python library for Arbitrary-Precision Floating-Point Arithmetic (Version 1.3.0)*. 2023.

[96] T. Thornes, P. Düben and T. Palmer. 'On the Use of Scale-Dependent Precision in Earth System Modelling'. In: *Quarterly Journal of the Royal Meteorological Society* 143.703 (2017), pp. 897–908.

[97] O. Tintó Prims et al. 'How to use Mixed Precision in Ocean Models: Exploring a Potential Reduction of Numerical Precision in NEMO 4.0 and ROMS 3.6'. In: *Geoscientific Model Development* 12.7 (2019), pp. 3135–3148.

[98] F. Tisseur. 'Newton's Method in Floating Point Arithmetic and Iterative Refinement of Generalized Eigenvalue Problems'. In: *SIAM Journal on Matrix Analysis and Applications* 22 (2001), pp. 1038–1057.

[99] Y. M. Tsai, P. Luszczek and J. Dongarra. 'Mixed-Precision Algorithm for Finding Selected Eigenvalues and Eigenvectors of Symmetric and Hermitian Matrices1'. In: 2022 IEEE/ACM Workshop on Latest Advances in Scalable Algorithms for Large-Scale Heterogeneous Systems (ScalAH). 2022, pp. 43–50.

[100] S. Uhlich et al. 'Mixed Precision DNNs: All You Need is a Good Parametrization'. In: *arXiv preprint* (2020), pp. 1–21. URL: `http://arxiv.org/abs/1905.11452` (visited on 25/04/2024).

[101] S. Wallis. 'Accurate Confidence Intervals on Binomial Proportions, Functions of Proportions, Algebraic Formulae and Effect Sizes'. In: *Survey of English Usage, UCL* (2022), pp. 1–20.

[102] K. Wang et al. 'HAQ: Hardware-Aware Automated Quantization With Mixed Precision'. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2019, pp. 8612–8620.

[103] N. Wang et al. 'Training Deep Neural Networks with 8-bit Floating Point Numbers'. In: *arXiv preprint* (2018), pp. 1–11. URL: `http://arxiv.org/abs/1812.08011` (visited on 27/03/2024).

[104] S. Wang et al. 'A Novel Parallel Finite Element Procedure for Nonlinear Dynamic Problems using GPU and Mixed-Precision Algorithm'. In: *Engineering Computations* 37.6 (2020), pp. 2193–2211.

[105] X. Wang, J. Lin and Q. Liao. 'Characterizing Performance Impacts of Subnormal Numbers on Vector Instructions and Transcendental Functions'. In: 2023 IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS). 2023, pp. 799–806.

[106] 'Direct Methods for Solving Linear Systems'. In: *Numerical Linear Algebra: An Introduction*. Ed. by H. Wendland. Cambridge Texts in Applied Mathematics. Cambridge: Cambridge University Press, 2017, pp. 59–100.

[107] A. Williams. *C++ Concurrency in Action: Practical Multithreading*. Shelter Island, NY: Manning, 2012.

[108] E. B. Wilson. 'Probable Inference, the Law of Succession, and Statistical Inference'. In: *Journal of the American Statistical Association* 22.158 (1927), pp. 209–212.

[109] Z. Zlatev. 'Use of Iterative Refinement in the Solution of Sparse Linear Systems'. In: *SIAM Journal on Numerical Analysis* 19.2 (1982), pp. 381–399.

[110] M. Zounon et al. 'Performance Impact of Precision Reduction in Sparse Linear Systems Solvers'. In: *PeerJ Computer Science* 8 (2022), pp. 1–22.

Acronyms

AI Artificial Intelligence. 15, 78

FP Floating Point. ix, xiii, 1, 2, 5–7, 13, 14, 16–21, 25, 27–33, 39, 40, 45, 53, 58, 64, 65, 77–80

FPN Floating Point Number. iii, 1–3, 5–8, 15, 17, 20, 25, 27, 30–34, 65, 77–80

FPU Floating Point Unit. 45, 48, 53

GEMM General Matrix Matrix Multiplication. 50

GEMV General Matrix Vector Multiplication. 50

GESV General Solve. 50, 53, 54, 56, 57

GETRF General Triangular Factorization. 50

MMM Matrix Matrix Multiplication. 14, 52, 53, 55–57, 63

MPS Multi Precision Simulator. xi, 43, 44, 46–48, 54, 58–60

MVM Matrix Vector Multiplication. 3, 14, 51–53, 55–57, 63

NaN Not a Number. 6

Glossary

Asian option Asian options are financial derivative contracts whose payoff depends on the average price of the underlying asset over a specified period, rather than just its price at expiration. 20

CMake CMake can be used with various programming languages, including C, C++, Fortran, CUDA, and Objective-C, to manage the build process and generate native build environments across different platforms.. 34

FPGA FPGA stands for Field-Programmable Gate Array. It's a type of integrated circuit designed to be configured by a user after manufacturing. 1, 19–21

Finite Element Method Finite Element Method (FEM) is a numerical technique used for solving differential equations by dividing the domain of interest into smaller, simpler subdomains called finite elements. 19

GMRES A sophisticated algorithm which uses a Krylov subspace to solve a linear system $Ax = b$, where $A \in \mathbb{R}^{n \times n}$ and $x, b \in \mathbb{R}^n$. 12, 17

IEEE 754 standard A standard for FPNs defined in [57]. 1–3, 5, 6, 27, 30, 33, 35, 65

Intel One of the leading companies in semiconductor chip manufacturing. 15

LU factorization An algorithm to split a Matrix $A \in \mathbb{R}^{n \times n}$ into a *upper triangular matrix* and *upper triangular matrix* $L, U \in \mathbb{R}^{n \times n}$. It can be represented as $A = LU$ and the *time complexity* is about $\mathcal{O}(n^3)$. 9, 13, 16, 17, 22, 24

M-matrix A matrix $A = (A_{ij})$ is an M-matrix if $a_{ij} \leq 0$ for $i \neq j$, A is nonsingular and $A^{-1} \geq 0$ [75]. 15

MPFR Software to simulate numbers in arbitrary Floating Point formats in C [39]. 1, 20, 25

MPFUN2020 Software to simulate numbers in arbitrary Floating Point formats in C++ form Bailey [6]. 20

Machine Learning Machine learning involves the development of algorithms and models that enable computers to learn and make predictions based on data without being explicitly programmed for each task. 18, 24

PLU factorization An algorithm to split a Matrix $A \in \mathbb{R}^{n \times n}$ into a *upper triangular matrix* and *upper triangular matrix* $L, U \in \mathbb{R}^{n \times n}$ using pivoting which makes the algorithm numerically more stable. It can be represented as $PA = LU$ and the *time complexity*, where $A \in \mathbb{R}^{n \times n}$ is the *permutation matrix*. The *time complexity* is about $\mathcal{O}(n^3)$. ix, xiii, 9, 11–13, 23, 24, 34, 53, 54, 56, 57

Python An high-level interpreted programming language. 15, 20, 27, 34

backward substitution An efficient way to solve for $x \in \mathbb{R}^n$ in $Ux = b$, where $U \in \mathbb{R}^{n \times n}$ is a *upper triangular matrix* and $b \in \mathbb{R}^n$. The *time complexity* is about $\mathcal{O}(n^2)$. 8, 10–12, 16, 54

bfloat16 A FP format with one *sign* bit, 8 *exponent* bits and 7 bits for the *mantissa* [58]. 15, 18

bordered linear system A bordered linear system has a system *system matrix* with added rows and columns which hold additional information. The resulting matrix has the form:

$$\begin{pmatrix} A & b \\ c & d \end{pmatrix}$$

where $A \in \mathbb{R}^{n \times n}$ is the original *system matrix*, both $b, c \in \mathbb{R}^n$ and $d \in \mathbb{R}$. 16

deep learning A form of AI which uses a large *neural network*. 15

double A Floating Point format, which uses 64 bits of precision. One bit for the sign, 11 bits for the exponent, and 52 bits for the mantissa. ix, 1–3, 5, 6, 8, 13, 14, 16, 17, 19–25, 34, 43, 50, 52, 54, 56, 63, 65, 79

eigenpair The pair of an eigenvector and its corresponding eigenvalue. 16

eigensolver A algorithm to find the eigenvalues and eigenvectors of a system. 17, 24

exponent Represents the scale or magnitude of the number. It determines the position of the decimal point and indicates how many places the mantissa should be shifted to the left or right to represent the actual value of the number. In other words, the exponent of a FPN specifies the power of the base 2 by which the mantissa is multiplied to obtain the final value of the floating-point number. xi, 5, 6, 27, 30–33, 43, 45–48, 78

float A Floating Point format that is similar to *single*. This naming is often used in the context of C and C++. 21, 43, 52

forward substitution An efficient way to solve for $x \in \mathbb{R}^n$ in $Lx = b$, where $L \in \mathbb{R}^{n \times n}$ is a *lower triangular matrix* and $b \in \mathbb{R}^n$. The *time complexity* is about $\mathcal{O}(n^2)$. 8, 10–12, 16, 54

half A Floating Point format, which uses 16 bits of precision. One bit for the sign, 5 bits for the exponent, and 10 bits for the mantissa. 1, 2, 14, 17–19, 21, 23

invertibility Invertibility of a matrix refers to the condition where a square matrix has an inverse, meaning there exists another matrix that, when multiplied with the original matrix, results in the identity matrix.. 35

iterative refinement A algorithm which utilizes multiple precisions to solve a linear system $Ax = b$, where $A \in \mathbb{R}^{n \times n}$ and $x, b \in \mathbb{R}^n$. ix, xi, xiii, 2, 3, 11–13, 15–17, 19–25, 49, 54, 58, 59

linear least square A method to get an approximate solution for an over-determined linear system. 15

long double A Floating Point format that is used by C++ and has more precision as standard *double*. 21, 50

lower triangular matrix A matrix $L \in \mathbb{R}^{n \times n}$ where all entries above the diagonal are equal to zero. 8, 10, 78

mantissa Represents the significant digits or the fractional part of the FPN. It consists of the part of the number that contributes to the precision of the number. ix, xi, xiii, 5–7, 20, 27, 29–31, 33, 36–38, 43, 44, 46, 49, 51, 59, 61, 78

mixed precision A *mixed precision* algorithm uses a small number of available precisions typically implemented in hardware. [54]. 2, 7, 8, 11, 15, 18–21, 24

mpmath Software to simulate numbers in arbitrary Floating Point formats in C [95]. 1, 20, 25

multiprecision A *multiprecision* algorithm uses one or more precisions that are arbitrary and are typically provided by means of software. 2, 7, 8, 11, 16, 20–22, 24, 25, 28, 57, 60, 65

neural network A trainable model to predict data inspired by the brain of living organisms. 15, 18, 78

numerical stability Numeric stability refers to the property of an algorithm or calculation that ensures small changes in input lead to proportionally small changes in output, preventing amplification of errors.. 9

permutation matrix A matrix $P \in \mathbb{R}^{n \times n}$ that represents row permutations. Each row contains exactly one element equal to 1. All other elements are zero. 8, 9, 78

sign The sign indicates whether the FPN number is positive or negative. It is represented by a single bit, where 0 is used for a positive number and 1 for a negative number. ix, 5, 6, 28, 30, 31, 78

single A Floating Point format, which uses 32 bits of precision. One bit for the sign, 8 bits for the exponent, and 23 bits for the mantissa. ix, xi, 1–3, 5, 6, 13, 14, 16–19, 21–24, 34, 43, 47, 50, 52, 54, 56, 63, 65, 78

soft-error Soft-errors refer to transient faults or errors that occur in electronic systems due to external factors such as radiation, electrical noise, or cosmic rays. 18, 19

subnormal numbers Subnormal numbers are non-normalized Floating Point numbers with a magnitude smaller than the smallest normalized number in a particular floating-point format. 7, 27

system matrix The matrix $A \in \mathbb{R}^{n \times n}$ in a linear system $Ax = b$, where $x, b \in \mathbb{R}^n$. 8, 10, 15, 16, 78

time complexity A measure of the computational efficiency of an algorithm. iii, 3, 8, 11, 13, 19, 27, 36, 45, 47, 63, 65, 77, 78

time dependency A measure of the computational efficiency of an algorithm. 27

unit roundoff The *unit roundoff* or also called *machine epsilon* is a measurement of the possible error when working with FPNs. It is defined as the distance between 1.0 and the next largest representable FPN at a given accuracy [51]. ix, 5–7, 9, 11, 13, 14

upper triangular matrix A matrix $U \in \mathbb{R}^{n \times n}$ where all entries bellow the diagonal are equal to zero. 8, 10, 77, 78

working precision Indicates the wanted precision of a numerical solution. 12

A. Appendix

A.1. Complete Code of the Simulator

A.1.1. mps.h

```
1
2  #ifndef MPS_LIB_MPS_H
3  #define MPS_LIB_MPS_H
4
5  #include <vector>
6  #include <cmath>
7  #include <string>
8
9  using namespace std;
10
11 class mps {
12
13 private:
14
15     // properties of the bit array
16     //--------------------------------
17     unsigned long mantissa_length;    // The length of the mantissa of
       the bit array.
18     unsigned long exponent_length;    // The length of the exponent of
       the bit array.
19     //--------------------------------
20
21
22     // the actual bit array
23     //--------------------------------
24     bool sign;
25     vector<bool> exponent;
26     vector<bool> mantissa;
27     //--------------------------------
28
29
30 public:
31     // constructors and destructor
32     //--------------------------------
33     mps(unsigned long mantissa_length, unsigned long exponent_length,
       double value);
34     mps(unsigned long mantissa_length, unsigned long exponent_length);
35     mps();
36     ~mps();
37
```

```
38      // getter methods
39      //-------------------------------
40      [[nodiscard]] bool getSign() const;
41      [[nodiscard]] vector<bool> getMantissa() const;
42      [[nodiscard]] vector<bool> getExponent() const;
43      [[nodiscard]] vector<bool> getBitArray() const;
44
45      [[nodiscard]] unsigned long getMantisseLength() const;
46      [[nodiscard]] unsigned long getExponentLength() const;
47      [[nodiscard]] unsigned long getBitArrayLength() const;
48      [[nodiscard]] double getValue() const;
49
50      [[nodiscard]] bool isZero() const;
51      [[nodiscard]] bool isInf() const;
52      [[nodiscard]] bool isPositive() const;
53      [[nodiscard]] bool isNaN() const;
54
55      [[nodiscard]] std::string print() const;
56      [[nodiscard]] std::string toString(int precision = -1) const;
57
58      // precision
59      //-------------------------------
60      [[nodiscard]] bool checkPrecision(const mps& compare, unsigned long
        precision) const;
61      [[nodiscard]] long long int getPrecision(const mps& compare) const;
62      [[nodiscard]] mps getAbsoluteError(const mps& compare) const;
63      [[nodiscard]] mps getRelativeError(const mps& compare) const;
64      [[nodiscard]] double getAbsoluteError_double(const mps& compare)
        const;
65      [[nodiscard]] double getRelativeError_double(const mps& compare)
        const;
66      [[nodiscard]] double getAbsoluteError_double(const double& compare)
        const;
67      [[nodiscard]] double getRelativeError_double(const double& compare)
        const;
68
69      // setter methods
70      //-------------------------------
71      void setInf(bool negative = false);
72      void setZero(bool negative = false);
73      void setNaN(bool negative = false);
74      void setSign(bool negative);
75      void setMantissa(vector<bool>& new_mantissa);
76      void setExponent(vector<bool>& new_exponent);
77
78      // cast
79      //-------------------------------
80      void cast(unsigned long new_mantissa_size, unsigned long
        new_exponent_size);
81
82      // operators
83      //-------------------------------
84      mps& operator=(const mps& other);
```

```
85      mps& operator|=(const mps& other);
86      mps& operator=(double value);
87      mps operator+(const mps& other) const;
88      mps operator-(const mps& other) const;
89      mps operator*(const mps& other) const;
90      mps operator/(const mps& other) const;
91
92      // comparators
93      //-------------------------------
94      bool operator==(const mps& other) const;
95      bool operator!=(const mps& other) const;
96      bool operator>(const mps& other) const;
97      bool operator<(const mps& other) const;
98      bool operator>=(const mps& other) const;
99      bool operator<=(const mps& other) const;
100
101
102 private:
103
104     // helper for cast
105     //-------------------------------
106     void resize_mps_object(unsigned long new_mantissa_size, unsigned long
        new_exponent_size);
107
108     // helper for operators
109     //-------------------------------
110     void setValue(double value);
111     [[nodiscard]] static mps addition(const mps& one, const mps& two,
       bool set_sign) ;
112     [[nodiscard]] static mps subtraction(const mps& minued, const mps&
       subtrahend, bool set_sign) ;
113     [[nodiscard]] static mps multiplication(const mps& minued, const mps&
        subtrahend, bool set_sign) ;
114     [[nodiscard]] static mps division(const mps& dividend, const mps&
       divisor, bool set_sign) ;
115
116     // helper for comparators
117     //-------------------------------
118     [[nodiscard]] static bool equal(const mps& one, const mps& two) ;
119     [[nodiscard]] static bool notEqual(const mps& one, const mps& two) ;
120     [[nodiscard]] static bool larger(const mps& one, const mps& two) ;
121     [[nodiscard]] static bool smaller(const mps& one, const mps& two) ;
122     [[nodiscard]] static bool largerEqual(const mps& one, const mps& two)
        ;
123     [[nodiscard]] static bool smallerEqual(const mps& one, const mps& two
       ) ;
124
125     [[nodiscard]] static char compare(const mps& one, const mps& two) ;
126
127     // general helper functions
128     //-------------------------------
129     [[nodiscard]] static vector<bool> binaryAddition(const vector<bool>&
       one, const vector<bool>& two, bool* carrier_return = nullptr);
```

```
130     [[nodiscard]] static vector<bool> binarySubtraction(const vector<bool
        >& minuend, const vector<bool>& subtrahend);
131
132     static void binarySummation(vector<bool>* summand, const vector<bool
        >& addend, bool = false);
133     static vector<bool> binaryOffsetAddition(const vector<bool>& lp,
        const vector<bool>& rp, unsigned long off_set, bool c, const bool p
        [2], const bool hd[2],  bool* cr = nullptr);
134     static inline bool round(vector<bool>* mantissa, unsigned long
        mantissa_len);
135
136     [[nodiscard]] static unsigned long binaryToInt(vector<bool>
        bit_vector);
137     [[nodiscard]] static vector<bool> intToBinary(unsigned long value);
138
139     [[nodiscard]] long getBias() const;
140     [[nodiscard]] static char larger(const vector<bool>& a, const vector<
        bool>& b, bool division_case = false);
141     static void shiftLeft(vector<bool>* vec);
142     static bool addOneToBinary(vector<bool>* vector);
143     static bool subtractOneFromBinary(vector<bool>* vector);
144     [[nodiscard]] static vector<bool> invertAndAddOne(const vector<bool>
        &vec, bool *carrie = nullptr, bool division_case = false);
145     [[nodiscard]] static bool allTrue(const vector<bool>& vector);
146     [[nodiscard]] static bool allFalse(const vector<bool>& vector);
147 };
148
149
150 #endif //MPS_LIB_MPS_H
```

Listing A.1: mps.h

A.1.2. mps.cpp

```
1
2  #include "mps.h"
3  #include <iostream>
4  #include <algorithm>
5
6  #include <sstream>
7
8  // constructors and destructor
9  //-------------------------------
10
11 /**
12  * Constructor for a multiprecision simulator object using values.
13  *
14  * @param mantissa_length Mantisse of the floating point representation
15  * @param exponent_length Exponent of the floating point representation
16  * @param value Value of the floating point number
17  */
18 mps::mps(unsigned long mantissa_length, unsigned long exponent_length,
        double value) {
```

```
19
20      // check input values
21      //-------------------------------
22      if (mantissa_length <= 0) {
23          throw std::invalid_argument("ERROR: in constructor : mantissa
        size too small");
24      }
25      if (exponent_length <= 1) {
26          throw std::invalid_argument("ERROR: in constructor : exponent
        size too small");
27      }
28      //-------------------------------
29
30      this->mantissa_length = mantissa_length;
31      this->exponent_length = exponent_length;
32
33      this->exponent.resize(this->exponent_length);
34      this->mantissa.resize(this->mantissa_length);
35
36      this->sign = false;
37
38      this->setValue(value);
39  }
40
41  /**
42   * Constructor for a multiprecision simulator object. The value is set to
        NAN.
43   * Info: Having the value set to NAN is probably not necessary, but it
        may prevent errors because of false use.
44   *
45   * @param mantissa_length Mantisse of the floating point representation
46   * @param exponent_length Exponent of the floating point representation
47   */
48  mps::mps(unsigned long mantissa_length, unsigned long exponent_length) {
49
50      // check input values
51      //-------------------------------
52      if (mantissa_length <= 0) {
53          throw std::invalid_argument("ERROR: in constructor : mantissa
        size too small");
54      }
55      if (exponent_length <= 1) {
56          throw std::invalid_argument("ERROR: in constructor : exponent
        size too small");
57      }
58      //-------------------------------
59
60      this->mantissa_length = mantissa_length;
61      this->exponent_length = exponent_length;
62
63      this->exponent.resize(this->exponent_length);
64      this->mantissa.resize(this->mantissa_length);
65
```

```
66      this->sign = false;
67
68      this->setNaN();      // Set it to be safe.
69  }
70
71  /**
72   * Constructor for an empty multiprecision simulator object.
73   */
74  mps::mps(){
75      this->mantissa_length = 0;
76      this->exponent_length = 0;
77
78      this->sign = false;
79  }
80
81  /**
82   * Destructor for a multiprecision simulator object.
83   */
84  mps::~mps() = default;
85  //-------------------------------
86
87
88  // getter methods
89  //-------------------------------
90
91  /**
92   * Returns the sign.
93   *
94   * @return the sign
95   */
96  [[nodiscard]] bool mps::getSign() const{
97      return this->sign;
98  }
99
100 /**
101  * Returns the mantissa.
102  *
103  * @return the mantissa
104  */
105 [[nodiscard]] vector<bool> mps::getMantissa() const{
106     return this->mantissa;
107 }
108
109 [[nodiscard]] vector<bool> mps::getExponent() const{
110     return this->exponent;
111 }
112
113 /**
114  * Returns the length of the mantissa.
115  *
116  * @return length of the mantissa
117  */
118 unsigned long mps::getMantisseLength() const {
```

```
119     return this->mantissa_length;
120 }
121
122 /**
123  * Returns the length of the exponent.
124  *
125  * @return length of the exponent
126  */
127 unsigned long mps::getExponentLength() const {
128     return this->exponent_length;
129 }
130
131 /**
132  * Returns the length of the whole bit array.
133  * Info: bit array = sign + matisse + exponent
134  *
135  * @return length of the total bit array
136  */
137 unsigned long mps::getBitArrayLength() const {
138     return this->exponent.size() + this->mantissa.size() + 1;
139 }
140
141 /**
142  * Concatenates the sign bit with the exponent vector and the mantissa
        vector.
143  * The resulting vector represents the floating point number in memory.
144  * Info: bit array = sign + matisse + exponent
145  *
146  * @return vector representing the floating point number
147  */
148 vector<bool> mps::getBitArray() const {
149
150     vector<bool> ret;
151     ret.reserve(1 + this->exponent_length + this->mantissa_length);
152
153     ret.push_back(this->sign);
154     ret.insert(ret.end(), this->exponent.begin(), this->exponent.end());
155     ret.insert(ret.end(), this->mantissa.begin(), this->mantissa.end());
156
157     return ret;
158 }
159
160 /**
161  * Calculates the value of the floating point object as double.
162  *
163  * @return value as double
164  */
165 double mps::getValue() const {
166
167     // handle special cases
168     if(isZero()){
169         return 0;
170     } else if (isInf() && isPositive()){
```

```
171         return numeric_limits<double>::infinity();
172     } else if (isInf()){
173         return numeric_limits<double>::infinity() * -1;
174     } else if (isNaN()){
175         return numeric_limits<double>::quiet_NaN();
176     }
177 
178 
179     double ret;
180 
181     // positive or negative
182     if(this->sign){
183         ret = -1;
184     } else {
185         ret = 1;
186     }
187 
188     // handle mantissa
189     for(unsigned long i = 0; i < mantissa_length; i++){
190         if(mantissa[i]){
191             if(ret >= 0){
192                 ret += pow(0.5, i + 1);
193             } else {
194                 ret -= pow(0.5, i + 1);
195             }
196         }
197     }
198 
199     // get exponent
200     long exp = 0;
201     for(unsigned long i = 0; i < exponent_length; i++){
202         if(this->exponent[i]){
203             exp += (long) pow(2, exponent_length-i-1);
204         }
205     }
206 
207     // apply bias and apply exponent
208     exp -= getBias();
209 
210     return ret * pow(2, exp);
211 }
212 
213 
214 /**
215  * Returns true if the mps object is zero.
216  * The sign bit is irrelevant.
217  *
218  * @return true if zero
219  */
220 bool mps::isZero() const{
221 
222     for(unsigned long i = 0; i < this->exponent_length; i++){
223         if(this->exponent[i]){
```

```
224             return false;
225         }
226     }
227 
228     for(unsigned long i = 0; i < this->mantissa_length; i++){
229         if(this->mantissa[i]){
230             return false;
231         }
232     }
233 
234     return true;
235 }
236 
237 /**
238  * Returns true if the mps object is positive or negative infinity.
239  *
240  * @return true if pos. or neg. infinity
241  */
242 bool mps::isInf() const{
243 
244 
245     for(unsigned long i = 0; i < this->exponent_length; i++){
246         if(!this->exponent[i]){
247             return false;
248         }
249     }
250 
251     for(unsigned long i = 0; i < this->mantissa_length; i++){
252         if(this->mantissa[i]){
253             return false;
254         }
255     }
256 
257     return true;
258 }
259 
260 /**
261  * Returns true if the mps object is positive.
262  *
263  * @return true if positive
264  */
265 bool mps::isPositive() const{
266 
267     if(this->sign){
268         return false;
269     } else {
270         return true;
271     }
272 }
273 
274 /**
275  * Returns true if the mps object represents a NaN value.
276  * The sign bit is irrelevant since some implementations differ between -
```

```
    NaN and +NaN.
 *
 * @return true if NaN
 */
bool mps::isNaN() const{

    for(unsigned long i = 0; i < exponent_length; i++){

        if(!exponent[i]){
            return false;
        }
    }

    if(!mantissa[0]){
        return false;
    }

    for(unsigned long i = 1; i < mantissa_length; i++){

        if(mantissa[i]){
            return false;
        }
    }

    return true ;
}

/**
 * Returns a string that represents the floating point number as a bit
    array.
 * Info: bit array = sign + matisse + exponent
 *
 * @return the FPN as a string consisting of 1s and 0s
 */
std::string mps::print() const {

    std::string str;
    for(bool bit : this->getBitArray()){
        if(bit){
            str.append("1");
        } else {
            str.append("0");
        }
    }

    return str;
}

/**
 * Returns the (double) value of the mps object as a string instead of a
    double.
 *
 * @return the value as a string
```

```
327  */
328 std::string mps::toString(const int precision) const {
329
330     if(precision < 0){
331         return std::to_string(this->getValue());
332     } else {
333         std::ostringstream out;
334         out.precision(precision);
335         out << std::fixed << this->getValue();
336         return std::move(out).str();
337     }
338 }
339 //--------------------------------
340
341
342 // precision
343 //--------------------------------
344 /**
345  * Compares the mantissa of two mps objects up to a specified precision.
346  * So if the two mantissas are different of a magnitude smaller than the
        precision specified, the function returns true.
347  *
348  *
349  * The two mps objects must have the same mantissa length and exponent
        length.
350  *
351  * @param compare the mps object to which "this" mps object should be
        compared to
352  * @param precision up to which precision the mps objects should be
        compared
353  * @return true if the mps objects are the same up to the given accuracy
354  */
355 [[nodiscard]] bool mps::checkPrecision(const mps& compare, unsigned long
        precision) const {
356
357     vector<bool> max_error_exponent;
358     max_error_exponent.reserve(this->exponent_length);
359     for(auto i = precision; i > 0; i /= 2){
360         max_error_exponent.insert(max_error_exponent.begin(), i % 2);
361     }
362     for(auto i = max_error_exponent.size(); i < this->exponent_length; i
        ++){
363         max_error_exponent.insert(max_error_exponent.begin(), false);
364     }
365
366     addOneToBinary(&max_error_exponent);
367     max_error_exponent = binarySubtraction(compare.exponent,
        max_error_exponent);
368
369
370     mps max_error = mps(this->mantissa_length, this->exponent_length);
371     max_error.setZero(false);
372
```

```
373     if(not max_error_exponent[0]){ // if overflow happened
374         max_error.setExponent(max_error_exponent);
375     } else {
376         cout << "WARNING: checkPrecision: value to small to check
    precision properly!\n";
377         max_error_exponent.resize(this->exponent_length, false);
378         max_error_exponent.back() = true;
379     }
380 
381     auto diff = *this - compare;
382     diff.setSign(false);
383 
384     if(diff <= max_error){
385         return true;
386     } else {
387         return false;
388     }
389 }
390 
391 /**
392  * Gets the precision of two mps objects.
393  * A high level definition of the precision is the number of matching
    mantissa bits from left to right.
394  * However, this is not a completely correct definition.
395  *
396  * Prints a WARNING when NaNs or Infinities are involved except if both
    mps objects are the same.
397  *
398  * Throws Exception:    When the mantissas do not match.
399  *                      When the exponents do not match.
400  *
401  * @param compare the mps object which which this mps object should be
    compared
402  * @return the matching mantissa bits
403  */
404 [[nodiscard]] long long int mps::getPrecision(const mps& compare) const {
405 
406     if (this->exponent_length != compare.exponent_length) {
407         throw std::invalid_argument("ERROR: in getPrecision : Exponents
    do not match");
408     }
409     if (this->mantissa_length != compare.mantissa_length) {
410         throw std::invalid_argument("ERROR: in getPrecision : Mantissas
    do not match");
411     }
412 
413     if(this->isNaN() && compare.isNaN()){
414         return (long long) compare.mantissa_length;
415     } else if(this->isNaN() || compare.isNaN()){
416         cout << "WARNING: getPrecision: one value is NaN => returning max
     negative value\n";
417         return numeric_limits<long long>::max() * -1;
418     }
```

```
419
420     if(this->isInf() && compare.isInf() && (this->isPositive() == compare
    .isPositive())){
421         return (long long) compare.mantissa_length;
422     } else if(this->isInf() || compare.isInf()){
423         cout << "WARNING: getPrecision: one value is infinity =>
    returning max negative value\n";
424         return numeric_limits<long long>::max() * -1;
425     }
426
427     auto exponent_this = (long long) binaryToInt(this->exponent);
428     auto exponent_compare = (long long) binaryToInt(compare.exponent);
429
430     long long precision_error = 0;
431
432     if(exponent_this == exponent_compare){
433
434         auto mantissa_this = (long long) binaryToInt(this->mantissa);
435         auto mantissa_compare = (long long) binaryToInt(compare.mantissa)
    ;
436
437         if(mantissa_this == mantissa_compare){
438             return (long long) compare.mantissa_length;
439         }
440
441         precision_error = (long long) floor(log2(abs(mantissa_this -
    mantissa_compare))) + 1;
442
443
444     } else if (1 == exponent_this - exponent_compare){
445
446         auto copy_this = this->mantissa;
447         copy_this.insert(copy_this.begin(), true);
448
449         auto copy_compare = compare.mantissa;
450         copy_compare.insert(copy_compare.begin(), {false, true});
451         copy_compare.pop_back();
452
453         auto mantissa_compare = (long long) binaryToInt(copy_compare);
454         auto mantissa_this = (long long) binaryToInt(copy_this);
455
456         precision_error = (long long) floor(log2(abs(mantissa_this -
    mantissa_compare))) + 1;
457
458     } else if(1 == exponent_compare - exponent_this) {
459
460         auto copy_compare = compare.mantissa;
461         copy_compare.insert(copy_compare.begin(), true);
462
463         auto copy_this = this->mantissa;
464         copy_this.insert(copy_this.begin(), {false, true});
465         copy_this.pop_back();
466
```

```
467         auto mantissa_compare = (long long) binaryToInt(copy_compare);
468         auto mantissa_this = (long long) binaryToInt(copy_this);
469 
470         precision_error = (long long) floor(log2(abs(mantissa_this -
    mantissa_compare))) + 1;
471 
472     } else {
473 
474         if(exponent_compare > exponent_this){
475             return 0;
476         } else {
477 
478             auto difference = abs(exponent_compare - exponent_this);
479 
480             for(unsigned long i = 0; i < (unsigned long) difference; i++)
    {
481                 if(this->mantissa[i]){
482                     difference++;
483                     break;
484                 }
485             }
486 
487             return difference * -1;
488         }
489     }
490 
491     return ((long long) compare.mantissa_length) - precision_error;
492 }
493 
494 /**
495  * Calculates the absolute error between two mps objects using operators
    from the mps class.
496  *
497  * @param compare the mps object to which "this" mps object should be
    compared to
498  * @return the absolute error
499  */
500 [[nodiscard]] mps mps::getAbsoluteError(const mps& compare) const{
501 
502     auto ret = compare - *this;
503     ret.setSign(false);
504 
505     return ret;
506 }
507 
508 /**
509  * Calculates the relative error between two mps objects using operators
    from the mps class.
510  *
511  * @param compare the mps object to which "this" mps object should be
    compared to
512  * @return the relative error
513  */
```

```
514 [[nodiscard]] mps mps::getRelativeError(const mps& compare) const {
515
516     auto ret = this->getAbsoluteError(compare) / compare;
517     ret.setSign(false);
518
519     return ret;
520 }
521
522 /**
523  * Calculates the absolute error between two mps objects using the built-
        in operators for double.
524  *
525  * @param compare the mps object to which "this" mps object should be
        compared to
526  * @return the absolute error
527  */
528 [[nodiscard]] double mps::getAbsoluteError_double(const mps& compare)
        const {
529
530     auto is = this->getValue();
531     auto should = compare.getValue();
532
533     return abs(is-should);
534 }
535
536 /**
537  * Calculates the relative error between two mps objects using the built-
        in operators for double.
538  *
539  * @param compare the mps object to which "this" mps object should be
        compared to
540  * @return the relative error
541  */
542 [[nodiscard]] double mps::getRelativeError_double(const mps& compare)
        const {
543
544     auto is = this->getValue();
545     auto should = compare.getValue();
546
547     return abs(is-should) / abs(should);
548 }
549
550 /**
551  * Calculates the absolute error between one mps object and a double
        value using the built-in operators for double.
552  *
553  * @param compare the mps object to which "this" mps object should be
        compared to
554  * @return the absolute error
555  */
556 [[nodiscard]] double mps::getAbsoluteError_double(const double& compare)
        const {
557
```

```
558     auto is = this->getValue();
559     return abs(is-compare);
560 }
561 
562 /**
563  * Calculates the relative error between one mps object and a double
        value using the built-in operators for double.
564  *
565  * @param compare the mps object to which "this" mps object should be
        compared to
566  * @return the relative error
567  */
568 [[nodiscard]] double mps::getRelativeError_double(const double& compare)
        const {
569 
570     auto is = this->getValue();
571     return abs(is-compare) / abs(compare);
572 }
573 //--------------------------------
574 
575 
576 // setter methods
577 //--------------------------------
578 
579 /**
580  * Sets the floating point number to infinity.
581  *
582  * @param negative set to true for negative infinity (default false)
583  */
584 void mps::setInf(bool negative) {
585 
586     if( exponent.empty() || mantissa.empty()){
587         exponent.resize(exponent_length);
588         mantissa.resize(exponent_length);
589     }
590 
591     if(negative){
592         sign = true;
593     } else {
594         sign = false;
595     }
596 
597     for(unsigned long i = 0; i < exponent_length; i++){
598         exponent[i] = true;
599     }
600 
601     for(unsigned long i = 0; i < mantissa_length; i++){
602         mantissa[i] = false;
603     }
604 }
605 
606 /**
607  * Sets the floating point number to zero.
```

```
608  *
609  * @param negative set to true for negative zero (default false)
610  */
611 void mps::setZero(bool negative){
612
613     if( (exponent.size() != this->exponent_length) || (mantissa.size() !=
         this->mantissa_length)){
614         exponent.resize(exponent_length);
615         mantissa.resize(mantissa_length);
616     }
617
618     sign = negative;
619
620     for(unsigned long i = 0; i < exponent_length; i++){
621         exponent[i] = false;
622     }
623     for(unsigned long i = 0; i < mantissa_length; i++){
624         mantissa[i] = false;
625     }
626 }
627
628 /**
629  * Sets the floating point number to NaN (not a number)
630  *
631  * The sign bit can be set individually using the parameter.
632  *
633  * @param negative sets the sign bit of the NaN value
634  */
635 void mps::setNaN(bool negative){
636
637     if( exponent.empty() || mantissa.empty()){
638         exponent.resize(exponent_length);
639         mantissa.resize(exponent_length);
640     }
641
642     if(negative){
643         sign = true;
644     } else {
645         sign = false;
646     }
647
648     for(unsigned long i = 0; i < exponent_length; i++){
649         exponent[i] = true;
650     }
651
652     mantissa[0] = true;
653     for(unsigned long i = 1; i < mantissa_length; i++){
654         mantissa[i] = false;
655     }
656 }
657
658 /**
659  * Sets the sign to a new sign.
```

```
660  *
661  * @param negative the value to which the sign bit should be set (true
        for negative)
662  */
663 void mps::setSign(bool negative){
664     this->sign = negative;
665 }
666 
667 /**
668  * Sets the mantissa to a new mantissa.
669  *
670  * Throws Exception:    When mantissa sizes do not match.
671  *
672  * @param new_mantissa the vector to which the mantissa should be set
673  */
674 void mps::setMantissa(vector<bool>& new_mantissa){
675 
676     if(new_mantissa.size() != this->mantissa.size()){
677         throw std::invalid_argument("ERROR: in setMantissa: New mantissa
        has wrong size.");
678     }
679 
680     for(unsigned long i = 0; i < new_mantissa.size(); i++){
681         this->mantissa[i] = new_mantissa[i];
682     }
683 }
684 
685 /**
686  * Sets the exponent to a new exponent.
687  *
688  * Throws Exception:    When exponent sizes do not match.
689  *
690  * @param new_mantissa the vector to which the mantissa should be set
691  */
692 void mps::setExponent(vector<bool>& new_exponent){
693 
694     if(new_exponent.size() != this->exponent.size()){
695         throw std::invalid_argument("ERROR: in setExponent: New exponent
        has wrong size.");
696     }
697 
698     for(unsigned long i = 0; i < new_exponent.size(); i++){
699         this->exponent[i] = new_exponent[i];
700     }
701 }
702 //--------------------------------
703 
704 
705 // cast
706 //--------------------------------
707 /**
708  * Casts the mps object to a new size.
709  *
```

```
710  * If the size is larger than before, the value is not changed, and the
       new elements are just filled with the
711  * appropriate values. If the new size is smaller, the value is rounded
       to be accurate for this new size.
712  * If needed, the object will be set to a special value.
713  *
714  * Info: Does check for correct input and special values.
715  *
716  * @param new_mantissa_size the new size of the mantissa
717  * @param new_exponent_size the new size of the exponent
718  * @return the resulting mps object.
719  */
720 void mps::cast(const unsigned long new_mantissa_size, const unsigned long
        new_exponent_size){
721
722
723     // check input values
724     //-------------------------------
725     if (new_mantissa_size <= 0) {
726         throw std::invalid_argument("ERROR: in cast : new mantissa size
        too small");
727     }
728     if (new_exponent_size <= 1) {
729         throw std::invalid_argument("ERROR: in cast : new exponent size
        too small");
730     }
731     //-------------------------------
732
733
734     // handle special values
735     //-------------------------------
736     if(this->isNaN()){
737         this->resize_mps_object(new_mantissa_size, new_exponent_size);
738         this->setNaN();
739         return;
740     } else if(this->isInf()){
741         this->resize_mps_object(new_mantissa_size, new_exponent_size);
742         this->setInf(this->sign);
743         return;
744     }
745     //-------------------------------
746
747
748     // rounding of the exponent
749     //-------------------------------
750     if(new_exponent_size > this->exponent_length){
751
752         if((this->exponent)[0] || this->isZero()){    // positive
        exponent case
753
754             for(auto i = this->exponent_length; i < new_exponent_size; i
        ++){
755                 this->exponent.insert(this->exponent.begin()+1, false);
```

```
756             }
757
758         } else {    // negative exponent case
759
760             for(auto i = this->exponent_length; i < new_exponent_size; i
    ++){
761                 this->exponent.insert(this->exponent.begin()+1, true);
762             }
763
764         }
765
766         this->exponent_length = new_exponent_size;
767
768     } else if(new_exponent_size < this->exponent_length) {
769
770
771         // check if the exponent is too large or too small
772         // ------------------------------------------------------
773         if(!this->isZero()) {
774             // "normal case"
775             auto tmp = this->exponent_length - new_exponent_size + 1;
776             for (unsigned long i = 1; i < tmp; i++) {
777                 if ((this->exponent)[i] == (this->exponent)[0]) {
778                     this->resize_mps_object(new_mantissa_size,
    new_exponent_size);
779
780                     if((this->exponent)[0]){
781                         this->setInf();
782                     } else {
783                         this->setZero();
784                     }
785
786                     return;
787                 }
788             }
789
790             // case if new exponent would be all 1's.
791             if((this->exponent)[0]) {
792                 auto tmp_case = true;
793                 for (auto i = this->exponent_length-1; i > (this->
    exponent_length - new_exponent_size + 1); i--) {
794                     if (!(this->exponent)[i]) {
795                         tmp_case = false;
796                         continue;
797                     }
798                 }
799                 if (tmp_case) {
800                     this->resize_mps_object(new_mantissa_size,
    new_exponent_size);
801                     this->setInf();
802                     return;
803                 }
804             }
```

```
805         }
806         // -------------------------------------------------------
807 
808 
809         for(auto i = new_exponent_size; i < this->exponent_length; i++){
810             this->exponent.erase(this->exponent.begin()+1);
811         }
812 
813         this->exponent_length = new_exponent_size;
814         //--------------------------------
815     }
816 
817 
818     // rounding of the mantissa
819     //--------------------------------
820     if(new_mantissa_size > this->mantissa_length){
821 
822         for(unsigned long i = this->mantissa_length; i <
    new_mantissa_size; i++){
823             this->mantissa.push_back(false);
824         }
825         this->mantissa_length = new_mantissa_size;
826 
827     } else if(new_mantissa_size < this->mantissa_length) {
828 
829         round(&(this->mantissa), new_mantissa_size);
830         this->mantissa_length = new_mantissa_size;
831     }
832     //--------------------------------
833 }
834 
835 /**
836  * Resizes the mps object to the a new size. It does not care for the
     value of the object
837  * but sets the size of the vectors and the value of the variables
     containing the sizes.
838  * Info: Does not check for correct input.
839  *
840  * @param new_mantissa_size the new size of the mantissa
841  * @param new_exponent_size the new size of the exponent
842  */
843 void mps::resize_mps_object(const unsigned long new_mantissa_size, const
    unsigned long new_exponent_size){
844 
845     if(new_mantissa_size > this->mantissa_length){
846         for(auto i = this->mantissa_length; i < new_mantissa_size; i++){
847             this->mantissa.push_back(false);
848         }
849     } else if(new_mantissa_size < this->mantissa_length){
850         for(auto i = new_mantissa_size; i < this->mantissa_length; i++){
851             this->mantissa.pop_back();
852         }
853     }
```

```
854
855     if(new_exponent_size > this->exponent_length){
856         for(auto i = this->exponent_length; i < new_exponent_size; i++){
857             this->exponent.push_back(false);
858         }
859     } else if(new_exponent_size < this->exponent_length){
860         for(auto i = new_exponent_size; i < this->exponent_length; i++){
861             this->exponent.pop_back();
862         }
863     }
864
865     this->mantissa_length = new_mantissa_size;
866     this->exponent_length = new_exponent_size;
867 }
868 //--------------------------------
869
870
871 // operators
872 //--------------------------------
873
874 /**
875  * Sets the value of the mps object.
876  */
877 mps& mps::operator=(double value){
878
879     this->setValue(value);
880     return *this;
881 }
882
883 /**
884  * Sets one mps object equal to another mps object.
885  */
886 mps& mps::operator=(const mps& other) {
887
888     if (this == &other){
889         return *this;
890     }
891
892     if (this->exponent_length != other.exponent_length) {
893         throw std::invalid_argument("ERROR: in = : Exponents do not match
    ");
894     }
895     if (this->mantissa_length != other.mantissa_length) {
896         throw std::invalid_argument("ERROR: in = : Mantissas do not match
    ");
897     }
898
899     this->sign = other.sign;
900     for(unsigned long i = 0; i < this->mantissa_length; i++){
901         this->mantissa[i] = other.mantissa[i];
902     }
903     for(unsigned long i = 0; i < this->exponent_length; i++){
904         this->exponent[i] = other.exponent[i];
```

```
905     }
906
907     return *this;
908 }
909
910 /**
911  * Sets one mps object equal to another mps object.
912  *
913  * If the mantissas or the exponents do not match, this operator does not
        throw an exception
914  * but reformats the object instead.
915  *
916  * This operator should be used with caution since, in this way, implicit
        casts are possible.
917  */
918 mps& mps::operator|=(const mps& other) {
919
920     if (this == &other){
921         return *this;
922     }
923
924     if (this->exponent_length != other.exponent_length) {
925         this->exponent_length = other.exponent_length;
926         this->exponent.resize(this->exponent_length);
927     }
928     if (this->mantissa_length != other.mantissa_length) {
929         this->mantissa_length = other.mantissa_length;
930         this->mantissa.resize(this->mantissa_length);
931     }
932
933     this->sign = other.sign;
934     for(unsigned long i = 0; i < this->mantissa_length; i++){
935         this->mantissa[i] = other.mantissa[i];
936     }
937     for(unsigned long i = 0; i < this->exponent_length; i++){
938         this->exponent[i] = other.exponent[i];
939     }
940
941     return *this;
942 }
943
944 /**
945  * Performs an addition of two mps floating point values.
946  *
947  * It is responsible for special values and choosing which "calculation"
       to use.
948  *
949  * Throws Exception:    When the mantissas do not match.
950  *                      When the exponents do not match.
951  */
952 mps mps::operator+(const mps& other) const {
953
954     if (this->exponent_length != other.exponent_length) {
```

```
955         throw std::invalid_argument("ERROR: in + : Exponents do not match
    ");
956     }
957     if (this->mantissa_length != other.mantissa_length) {
958         throw std::invalid_argument("ERROR: in + : Mantissas do not match
    ");
959     }
960 
961 
962     if(this->isNaN() || other.isNaN()){
963 
964         mps ret(this->mantissa_length, this->exponent_length);
965         ret.setNaN();
966         return ret;
967 
968     } else if(this->isInf() && other.isInf()){
969         mps ret(this->mantissa_length, this->exponent_length);
970 
971         if(this->isPositive() == other.isPositive()){
972             ret.setInf(!this->isPositive());
973         } else {
974             ret.setNaN();
975         }
976 
977         return ret;
978     } else if(this->isInf()){
979 
980         mps ret(this->mantissa_length, this->exponent_length);
981         ret.setInf(!this->isPositive());
982         return ret;
983 
984     } else if(other.isInf()){
985 
986         mps ret(other.mantissa_length, other.exponent_length);
987         ret.setInf(!other.isPositive());
988         return ret;
989 
990     } else if(this->isZero()){
991         return other;
992     } else if(other.isZero()){
993         return *this;
994     }
995 
996 
997     if(this->isPositive() && other.isPositive()){
998         return addition(*this, other, false);
999     } else if(!this->isPositive() && !other.isPositive()){
1000         return addition(*this, other, true);
1001     } else if(!this->isPositive()){
1002         return subtraction(other, *this, false);
1003     } else{
1004         return subtraction(*this, other, false);
1005     }
```

```
1006
1007 }
1008
1009 /**
1010  * Performs a subtraction of two mps floating point values.
1011  *
1012  * It is responsible for special values and choosing which "calculation"
       to use.
1013  *
1014  * Throws Exception:    When the mantissas do not match.
1015  *                      When the exponents do not match.
1016  */
1017 mps mps::operator-(const mps& other) const {
1018
1019     if (this->exponent_length != other.exponent_length) {
1020         throw std::invalid_argument("ERROR: in - : Exponents do not match
      ");
1021     }
1022     if (this->mantissa_length != other.mantissa_length) {
1023         throw std::invalid_argument("ERROR: in - : Mantissas do not match
      ");
1024     }
1025
1026
1027     if(this->isNaN() || other.isNaN()){
1028
1029         mps ret(this->mantissa_length, this->exponent_length);
1030         ret.setNaN();
1031         return ret;
1032
1033     } else if(this->isInf() && other.isInf()){
1034         mps ret(this->mantissa_length, this->exponent_length);
1035
1036         if(this->isPositive() == other.isPositive()){
1037             ret.setNaN();
1038         } else {
1039             ret.setInf(!this->isPositive());
1040         }
1041
1042         return ret;
1043     } else if(this->isInf()){
1044
1045         mps ret(this->mantissa_length, this->exponent_length);
1046         ret.setInf(!this->isPositive());
1047         return ret;
1048
1049     } else if(other.isInf()){
1050
1051         mps ret(other.mantissa_length, other.exponent_length);
1052         ret.setInf(other.isPositive());
1053         return ret;
1054
1055     } else if(this->isZero()){
```

```
1056         auto ret = other;
1057         ret.sign = !ret.sign;
1058         return ret;
1059     } else if(other.isZero()){
1060         return *this;
1061     }
1062 
1063 
1064     if(this->isPositive() && other.isPositive()){
1065         return subtraction(*this, other, false);
1066     } else if(!this->isPositive() && !other.isPositive()){
1067         return subtraction(*this, other, true);
1068     } else if(this->isPositive()) {
1069         return addition(*this, other, false);
1070     } else {
1071         return addition(*this, other, true);
1072     }
1073 }
1074 
1075 /**
1076  * Performs a multiplication of two mps floating point values.
1077  *
1078  * It is responsible for special values and choosing which "calculation"
       to use.
1079  *
1080  * Throws Exception:    When the mantissas do not match.
1081  *                      When the exponents do not match.
1082  */
1083 mps mps::operator*(const mps& other) const {
1084 
1085     if (this->exponent_length != other.exponent_length) {
1086         throw std::invalid_argument("ERROR: in * : Exponents do not match
       ");
1087     }
1088     if (this->mantissa_length != other.mantissa_length) {
1089         throw std::invalid_argument("ERROR: in * : Mantissas do not match
       ");
1090     }
1091 
1092 
1093     if(this->isNaN() || other.isNaN()){
1094 
1095         mps ret(this->mantissa_length, this->exponent_length);
1096         ret.setNaN();
1097         return ret;
1098 
1099     } else if(this->isInf() && other.isInf()){
1100         mps ret(this->mantissa_length, this->exponent_length);
1101 
1102         if(this->isPositive() == other.isPositive()){
1103             ret.setNaN();
1104         } else {
1105             ret.setInf(!this->isPositive());
```

```
1106         }
1107
1108         return ret;
1109     } else if(this->isInf()){
1110
1111         mps ret(this->mantissa_length, this->exponent_length);
1112         ret.setInf(!this->isPositive());
1113         return ret;
1114
1115     } else if(other.isInf()){
1116
1117         mps ret(other.mantissa_length, other.exponent_length);
1118         ret.setInf(!other.isPositive());
1119         return ret;
1120
1121     } else if(this->isZero() || other.isZero()){
1122         mps ret(other.mantissa_length, other.exponent_length);
1123         ret.setZero();
1124         return ret;
1125     }
1126
1127
1128     return multiplication(*this, other, this->sign != other.sign);
1129 }
1130
1131 /**
1132  * Performs a division of two mps floating point values.
1133  *
1134  * It is responsible for special values and choosing which "calculation"
       to use.
1135  *
1136  * Throws Exception:    When the mantissas do not match.
1137  *                      When the exponents do not match.
1138  */
1139 mps mps::operator/(const mps& other) const {
1140
1141     if (this->exponent_length != other.exponent_length) {
1142         throw std::invalid_argument("ERROR: in / : Exponents do not match
       ");
1143     }
1144     if (this->mantissa_length != other.mantissa_length) {
1145         throw std::invalid_argument("ERROR: in / : Mantissas do not match
       ");
1146     }
1147
1148
1149     if(this->isNaN() || other.isNaN()){
1150
1151         mps ret(this->mantissa_length, this->exponent_length);
1152         ret.setNaN();
1153         return ret;
1154
1155     } else if(this->isInf() && other.isInf()){
```

```
1156         mps ret(this->mantissa_length, this->exponent_length);
1157 
1158         if(this->isPositive() == other.isPositive()){
1159             ret.setNaN();
1160         } else {
1161             ret.setInf(!this->isPositive());
1162         }
1163 
1164         return ret;
1165     } else if(this->isInf()){
1166 
1167         mps ret(this->mantissa_length, this->exponent_length);
1168         ret.setInf(!this->isPositive());
1169         return ret;
1170 
1171     } else if(other.isInf()){
1172 
1173         mps ret(other.mantissa_length, other.exponent_length);
1174         if(this->isPositive() == other.isPositive()){
1175             ret.setZero();
1176         } else {
1177             ret.setZero(true);
1178         }
1179         return ret;
1180 
1181     } else if(this->isZero()){
1182         mps ret(other.mantissa_length, other.exponent_length);
1183         if(this->isPositive() == other.isPositive()){
1184             ret.setZero();
1185         } else {
1186             ret.setZero(true);
1187         }
1188         return ret;
1189     } else if(other.isZero()){
1190         mps ret(other.mantissa_length, other.exponent_length);
1191         ret.setInf(!this->isPositive());
1192         return ret;
1193     }
1194 
1195 
1196     return division(*this, other, this->sign != other.sign);
1197 }
1198 
1199 
1200 /**
1201  * Calculates the binary floating point representation of the given value
        and saves it inside the mps object.
1202  *
1203  * @param value the value to which the bit array should be set
1204  */
1205 void mps::setValue(const double value) {
1206 
1207     // Handle special values.
```

```
1208     //-------------------------------
1209     if(0 == value){
1210         setZero(); return;
1211     } else if(numeric_limits<double>::infinity() == value) {
1212         setInf(); return;
1213     } else if (numeric_limits<double>::infinity() * -1 == value){
1214         setInf(true); return;
1215     } else if (isnan(value)){
1216         if(signbit(value)){
1217             setNaN(true);
1218         } else {
1219             setNaN();
1220         }
1221         return;
1222     }
1223     //-------------------------------
1224
1225     // prepare vectors
1226     //-------------------------------
1227     exponent.clear();
1228     mantissa.clear();
1229     //-------------------------------
1230
1231
1232     // set sign
1233     //-------------------------------
1234     this->sign = value <= 0;
1235     //-------------------------------
1236
1237
1238     // exponent
1239     //-------------------------------
1240     // calculate the exponent
1241     long mantissa_shift;
1242     if(abs(value) > numeric_limits<float>::max()){
1243         // When the value is too large, rounding problems occur using
     log2.
1244         // (At least I think that is the problem)
1245         double tmp_value = abs(value);
1246         long tmp_count = 0;
1247
1248         while(tmp_value >= 1){
1249             tmp_value /= 2;
1250             tmp_count++;
1251         }
1252         mantissa_shift = tmp_count-1;
1253     } else {
1254         mantissa_shift = (long) floor(log2(abs(value)));
1255     }
1256
1257     // Getting the exponent in binary format.
1258     // The sign is not relevant. Always returns a "positive" value.
1259     for(auto i = mantissa_shift + getBias(); i > 0; i /= 2){
```

```
1260            exponent.insert(exponent.begin(), i % 2);
1261        }
1262
1263        bool value_too_large = false;
1264        if(exponent.size() > exponent_length){
1265            value_too_large = true;
1266        } else if(exponent.size() == exponent_length){
1267            value_too_large = allTrue(this->exponent);
1268        }
1269
1270        if  (value_too_large){
1271
1272            exponent.resize(exponent_length);
1273            mantissa.resize(mantissa_length);
1274
1275            if(value > 0){
1276                this->setInf();
1277            } else {
1278                this->setInf(true);
1279            }
1280
1281            return;
1282
1283        } else {
1284            exponent.insert(exponent.begin(), exponent_length-exponent.size()
        , false);
1285        }
1286        //--------------------------------
1287
1288
1289        // mantissa
1290        //--------------------------------
1291        auto tmp_value = value / pow(2, mantissa_shift); // adjust the value
        so that everything is behind the decimal point.
1292
1293        // Convert the part after the decimal point to a bit string.
1294        // The sign is not relevant. Always returns a "positive" value.
1295        double reminder = abs(tmp_value) - floor(abs(tmp_value));
1296        for(unsigned long i = 0; i < mantissa_length; i++){
1297
1298            reminder *= 2;
1299            if(reminder >= 1){
1300                mantissa.push_back(true); reminder -= 1;
1301            } else{
1302                mantissa.push_back(false);
1303            }
1304        }
1305        //--------------------------------
1306
1307    }
1308
1309    /**
1310     * Performs an addition on two mps objects that have the same sign.
```

```
1311  *
1312  * @param one reference to the first addend
1313  * @param two reference to the second addend
1314  * @param set_sign the sign to which the final result should be set
1315  * @return the resulting mps object
1316  */
1317 [[nodiscard]] mps mps::addition(const mps &one, const mps &two, const
        bool set_sign) {
1318
1319     // Set up the return object.
1320     //--------------------------------
1321     mps ret;
1322     ret.exponent_length = one.exponent_length;
1323     ret.mantissa_length = one.mantissa_length;
1324     ret.sign = set_sign;
1325     //--------------------------------
1326
1327
1328     // addition of mantissas.
1329     //--------------------------------
1330     auto add = []
1331             (const vector<bool> &first, const vector<bool> &second,
        unsigned long off_set, bool *carrier) -> vector<bool>{
1332
1333         bool hd[2] = {true, true};
1334         bool p[2] = {false, false};
1335
1336         return binaryOffsetAddition(first, second, off_set, true, p, hd,
        carrier);
1337     };
1338     //--------------------------------
1339
1340     // set exponent and addition
1341     //--------------------------------
1342     bool carrier;
1343     unsigned long exponent_diff;
1344     char larger_tmp = larger(one.exponent, two.exponent);
1345     if(1 == larger_tmp){
1346
1347         exponent_diff = binaryToInt(binarySubtraction(one.exponent, two.
        exponent));
1348         ret.exponent = one.exponent;
1349
1350         if(exponent_diff <= one.mantissa.size()){
1351             ret.mantissa = add(two.mantissa, one.mantissa, exponent_diff,
         &carrier);
1352         } else if(one.mantissa.back() && (exponent_diff == one.mantissa.
        size()+1) && allFalse(two.mantissa)){
1353             // special case where the rounding distance is the same, and
        the rounded number, therefore, must be even.
1354             ret.mantissa = one.mantissa;
1355             if(addOneToBinary(&ret.mantissa)){
1356                 addOneToBinary(&ret.exponent);
```

```
1357                // Checking the carrier bit can be omitted since it (conveniently) does this by default.
1358            }
1359            return ret;
1360        } else {
1361            ret.mantissa = one.mantissa;
1362            return ret;
1363        }
1364
1365    } else if(-1 == larger_tmp){
1366
1367        exponent_diff = binaryToInt(binarySubtraction(two.exponent, one.exponent));
1368        ret.exponent = two.exponent;
1369
1370        if(exponent_diff <= one.mantissa.size()){
1371            ret.mantissa = add(one.mantissa, two.mantissa, exponent_diff, &carrier);
1372        } else if(two.mantissa.back() && (exponent_diff == two.mantissa.size()+1) && allFalse(one.mantissa)){
1373            // special case where the rounding distance is the same, and the rounded number, therefore, must be even.
1374            ret.mantissa = two.mantissa;
1375            if(addOneToBinary(&ret.mantissa)){
1376                addOneToBinary(&ret.exponent);
1377                // Checking the carrier bit can be omitted since it (conveniently) does this by default.
1378            }
1379            return ret;
1380        } else {
1381            ret.mantissa = two.mantissa;
1382            return ret;
1383        }
1384
1385    } else {
1386
1387        ret.exponent = two.exponent;
1388        ret.mantissa = add(one.mantissa, two.mantissa, 0, &carrier);
1389    }
1390    //--------------------------------
1391
1392
1393    // adjusting the mantissa
1394    //--------------------------------
1395    ret.mantissa.erase(ret.mantissa.begin(), ret.mantissa.begin()+1);
1396    if(round(&ret.mantissa, ret.mantissa_length)){
1397        addOneToBinary(&ret.exponent);
1398        // checking is not needed since it is (conveniently) correct by default
1399    }
1400    //--------------------------------
1401
1402    // adjusting the mantissa
```

```
1403     //-------------------------------
1404     if(carrier){
1405         addOneToBinary(&ret.exponent);
1406         if(allTrue(ret.exponent)){
1407             ret.setInf(set_sign);
1408             return ret;
1409         }
1410     }
1411     //-------------------------------
1412
1413
1414
1415     return ret;
1416 }
1417
1418 /**
1419  * Performs a subtraction on two mps objects that have the same sign.
1420  *
1421  * @param minued reference to the minued number
1422  * @param subtrahend reference to the subtracted number
1423  * @param set_sign the sign to which the final result should be set
1424  * @return the resulting mps object
1425  */
1426 [[nodiscard]] mps mps::subtraction(const mps &minued, const mps &
        subtrahend, bool set_sign) {
1427
1428     // Set up the return object.
1429     //-------------------------------
1430     mps ret;
1431     ret.exponent_length = minued.exponent_length;
1432     ret.mantissa_length = minued.mantissa_length;
1433     ret.sign = set_sign;
1434     //-------------------------------
1435
1436
1437     // subtraction of mantissas.
1438     //-------------------------------
1439     auto subtract = []
1440             (const vector<bool> &first, const vector<bool> &sub, unsigned
         long off_set) -> vector<bool>{
1441
1442             bool carrier;
1443
1444             auto iSub = invertAndAddOne(sub, &carrier);
1445             bool hd[2] = {carrier, true};
1446             bool p[2] = {true, false};
1447
1448             return binaryOffsetAddition(iSub, first, off_set, false, p,
        hd);
1449     };
1450     //-------------------------------
1451
1452
```

```
1453    // set exponent and does calculation.
1454    //------------------------------
1455    char larger_tmp = larger(minued.exponent, subtrahend.exponent);
1456    if(1 == larger_tmp){
1457        unsigned long exponent_diff = binaryToInt(binarySubtraction(
        minued.exponent, subtrahend.exponent));
1458        ret.exponent = minued.exponent;
1459
1460        if(exponent_diff <= minued.mantissa.size()){
1461            ret.mantissa = subtract(minued.mantissa, subtrahend.mantissa,
         exponent_diff);
1462        } else if (exponent_diff == minued.mantissa.size() + 1){
1463            ret.mantissa = minued.mantissa;
1464
1465            if(allFalse(ret.mantissa)){
1466                subtractOneFromBinary(&ret.exponent);
1467            }
1468            subtractOneFromBinary(&ret.mantissa);
1469
1470            return ret;
1471        } else {
1472            ret.mantissa = minued.mantissa;
1473            return ret;
1474        }
1475
1476    } else if(-1 == larger_tmp){
1477
1478        unsigned long exponent_diff = binaryToInt(binarySubtraction(
        subtrahend.exponent, minued.exponent));
1479        ret.exponent = subtrahend.exponent;
1480        ret.sign = !ret.sign; // flip sign
1481
1482        if(exponent_diff <= minued.mantissa.size()){
1483            ret.mantissa = subtract(subtrahend.mantissa, minued.mantissa,
         exponent_diff);
1484        } else if(exponent_diff == minued.mantissa.size() + 1){
1485            ret.mantissa = subtrahend.mantissa;
1486
1487            if(allFalse(ret.mantissa)){
1488                subtractOneFromBinary(&ret.exponent);
1489            }
1490            subtractOneFromBinary(&ret.mantissa);
1491
1492            return ret;
1493        } else {
1494            ret.mantissa = subtrahend.mantissa;
1495            return ret;
1496        }
1497
1498    } else {
1499
1500        ret.exponent = minued.exponent;
1501
```

```
1502         larger_tmp = larger(minued.mantissa, subtrahend.mantissa);
1503         if(1 == larger_tmp){
1504 
1505             ret.mantissa = subtract(minued.mantissa, subtrahend.mantissa,
     0);
1506 
1507         } else if(-1 == larger_tmp){
1508 
1509             ret.mantissa = subtract(subtrahend.mantissa, minued.mantissa,
     0);
1510             ret.sign = !ret.sign; // flip sign
1511 
1512         } else {
1513             ret.setZero();
1514             return ret;
1515         }
1516 
1517     }
1518     //--------------------------------
1519 
1520 
1521     // Adjust mantissa and exponent
1522     //--------------------------------
1523     unsigned long exponent_shift = 0;
1524     for(unsigned long i = 0; i < ret.mantissa.size(); i++){
1525         if(ret.mantissa[i]){
1526             exponent_shift = i;
1527             break;
1528         }
1529 
1530         // for every "right shift", pull one zero from the right.
1531         ret.mantissa.push_back(false);
1532     }
1533 
1534     ret.mantissa.erase(ret.mantissa.begin(), ret.mantissa.begin() + (long
    ) exponent_shift + 1);
1535 
1536     vector<bool> exponent_shift_binary = intToBinary(exponent_shift);
1537     exponent_shift_binary.insert(exponent_shift_binary.begin(), ret.
    exponent_length - exponent_shift_binary.size(), false);
1538     ret.exponent = binarySubtraction(ret.exponent, exponent_shift_binary)
    ;
1539     //--------------------------------
1540 
1541 
1542     if(round(&ret.mantissa, ret.mantissa_length)){
1543         addOneToBinary(&ret.exponent);
1544         // cannot overflow since it is a subtraction.
1545     }
1546 
1547     return ret;
1548 }
1549 
```

```
1550 /**
1551  * Performs a multiplication on two mps objects that have the same sign.
1552  *
1553  * @param one reference to the first multiplicand
1554  * @param two reference to the second multiplicand
1555  * @param set_sign the sign to which the final result should be set
1556  * @return the resulting mps object
1557  */
1558 [[nodiscard]] mps mps::multiplication(const mps& one, const mps& two,
      bool set_sign) {
1559
1560     // Set up the return object.
1561     //--------------------------------
1562     mps ret(one.mantissa_length, one.exponent_length);
1563     ret.sign = set_sign;
1564     //--------------------------------
1565
1566
1567     // Calculate the exponent
1568     //--------------------------------
1569     if(one.exponent[0] && two.exponent[0]){ // Both exponents are
      positive.
1570
1571         vector<bool> addend = two.exponent;
1572
1573         addend[0] = !addend[0];
1574         addOneToBinary(&addend);    // Carrier bit must not be checked
      because of if-statement.
1575
1576         bool tmp_carry;
1577         ret.exponent = binaryAddition(one.exponent, addend, &tmp_carry);
1578
1579         // Check if the number will be more than the maximal allowed
      value.
1580         if(tmp_carry && one.exponent[0] && !addend[0]){
1581             ret.setInf(ret.sign);
1582             return ret;
1583         }
1584
1585     } else {
1586
1587         vector<bool> subtrahend;
1588         subtrahend.reserve(two.exponent.size());
1589         subtrahend.push_back(false);
1590         for(unsigned long i = 1; i < two.exponent.size(); i++){
1591             subtrahend.push_back(true);
1592         }
1593
1594         if(one.exponent[0] && !two.exponent[0]){
1595             subtrahend = binarySubtraction(subtrahend, one.exponent);
1596             ret.exponent = binarySubtraction(two.exponent, subtrahend);
1597         } else {
1598             subtrahend = binarySubtraction(subtrahend, two.exponent);
```

```
1599
1600            if(!one.exponent[0] && !two.exponent[0] && 1 == larger(
    subtrahend, one.exponent)){
1601                ret.setZero();
1602                return ret;
1603            }
1604
1605            ret.exponent = binarySubtraction(one.exponent, subtrahend);
1606        }
1607    }
1608    //--------------------------------
1609
1610
1611    // Calculate the mantissa
1612    //--------------------------------
1613    //bool prefix[2] = {false, true};
1614
1615    vector<bool> S;
1616    bool carrier;
1617    S = invertAndAddOne(one.mantissa, &carrier);
1618    S.insert(S.begin(), carrier);
1619    S.insert(S.begin(), true);
1620
1621
1622    // set up P vector (product)
1623    vector<bool> &P = ret.mantissa;     // Use reference P in order to
    keep the naming.
1624    P.clear();
1625    P.reserve(one.mantissa.size() + two.mantissa_length + 5);
      // 3 => 2* (sign and "invisible 1") + 1
1626    for(unsigned long i = 0; i < one.mantissa_length + 2; i++){    // 2
    => sign and "invisible 1"
1627        P.push_back(false);
1628    }
1629    P.push_back(false);
1630    P.push_back(true);
1631    for(unsigned long i = 0; i < two.mantissa_length; i++){
1632        P.push_back(two.mantissa[i]);
1633    }
1634    P.push_back(false);
1635
1636    // main loop
1637    for(unsigned long i = 0; i < two.mantissa_length+2; i++){
1638
1639
1640        if(!P.end()[-2] && P.back()){
1641            binarySummation(&P, one.mantissa, true);
1642        } else if(P.end()[-2] && !P.back()) {
1643            binarySummation(&P, S);
1644        }
1645
1646        P.pop_back();
1647        P.insert(P.begin(), P[0]);
```

```
1648
1649        }
1650
1651        P.pop_back();
1652        P.erase (P.begin());         // delete "invisible 1"
1653
1654        // normalize
1655        int count = 0;
1656        for(unsigned long i = 0; i < P.size(); i++){
1657            if(P[0]){
1658                P.erase(P.begin());
1659                break;
1660            }
1661            P.erase(P.begin());
1662            count++;
1663        }
1664        //--------------------------------
1665
1666        // rounding
1667        //--------------------------------
1668        if(round(&P, ret.mantissa_length)){
1669            addOneToBinary(&ret.exponent);
1670            // Cannot overflow. Same reasoning as for addition.
1671        }
1672
1673        if(count <= 1){
1674            addOneToBinary(&ret.exponent);
1675        }
1676        //--------------------------------
1677
1678        return ret;
1679    }
1680
1681    /**
1682     * Performs a division on two mps objects that have the same sign.
1683     *
1684     * @param dividend reference to the dividend of the division
1685     * @param divisor reference to the divisor of the division
1686     * @param set_sign the sign to which the final result should be set
1687     * @return the resulting mps object
1688     */
1689    [[nodiscard]] mps mps::division(const mps& dividend, const mps& divisor,
        bool set_sign) {
1690
1691        // Set up the return object.
1692        //--------------------------------
1693        mps ret; //(dividend.mantissa_length, divisor.exponent_length);
1694        ret.mantissa_length = dividend.mantissa_length;
1695        ret.exponent_length = dividend.exponent_length;
1696        ret.sign = set_sign;
1697        //--------------------------------
1698
1699
```

```
1700     // Calculate the exponent
1701     //-------------------------------
1702     if(divisor.exponent[0]){ // Both exponents are positive.
1703 
1704         vector<bool> subtrahend = divisor.exponent;
1705 
1706         subtrahend[0] = !subtrahend[0];
1707         addOneToBinary(&subtrahend);    // Carrier bit must not be
    checked because of if-statement.
1708 
1709         ret.exponent = binarySubtraction(dividend.exponent, subtrahend);
1710 
1711         if(1 == larger(ret.exponent, dividend.exponent)){
1712             ret.mantissa.resize(ret.mantissa_length, false);
1713             ret.exponent.resize(ret.exponent_length, false);
1714             ret.setZero();
1715             return ret;
1716         }
1717 
1718 
1719     } else if(!divisor.exponent[0]){
1720 
1721         vector<bool> subtrahend;
1722         subtrahend.reserve(divisor.exponent.size());
1723         subtrahend.push_back(false);
1724         for(unsigned long i = 1; i < divisor.exponent.size(); i++){
1725             subtrahend.push_back(true);
1726         }
1727 
1728         subtrahend = binarySubtraction(subtrahend, divisor.exponent);
1729 
1730         bool carrier;
1731         ret.exponent = binaryAddition(dividend.exponent, subtrahend, &
    carrier);
1732 
1733         if(carrier || allTrue(ret.exponent)){
1734             ret.mantissa.resize(ret.mantissa_length, false);
1735             ret.exponent.resize(ret.exponent_length, false);
1736             ret.setInf(ret.sign);
1737             return ret;
1738         }
1739 
1740     }
1741     //-------------------------------
1742 
1743 
1744     // Division
1745     //-------------------------------
1746 
1747     // remainder
1748     vector<bool> R = dividend.mantissa;
1749     R.insert(R.begin(), true);
1750     R.insert(R.begin(), 2, false);
```

```
1751
1752    // quotient
1753    vector<bool> &Q = ret.mantissa;         // reference for better
        naming.
1754    Q.resize(ret.mantissa_length, false);
1755
1756    // subtrahend (used for binarySummation)
1757    auto S = invertAndAddOne(divisor.mantissa, nullptr, true);    //
        carry bit must not be checked because d must not be 0;
1758
1759
1760    // main loop
1761    for(auto && i : Q){
1762        shiftLeft(&R);
1763
1764        char tmp = larger(R, divisor.mantissa, true);
1765        if(1 == tmp || 0 == tmp){
1766            binarySummation(&R, S);
1767            i = true;
1768        }
1769    }
1770    //--------------------------------
1771
1772
1773    // normalisation
1774    //--------------------------------
1775    unsigned long count = 0;
1776    vector<bool> count_vec(divisor.exponent.size(), false);
1777
1778    for(unsigned long i = 0; i < Q.size(); i++){
1779        if(Q[0]){
1780            Q.erase(Q.begin());
1781            count++;
1782            //addOneToBinary(&count_vec);
1783            break;
1784        } else {
1785            Q.erase(Q.begin());
1786            count++;
1787            addOneToBinary(&count_vec);
1788        }
1789    }
1790
1791    // rerun the algorithm for every leading bit removed.
1792    for(unsigned long i = 0; i < count; i++){
1793        shiftLeft(&R);
1794
1795        char tmp = larger(R, divisor.mantissa, true);
1796        if(1 == tmp || 0 == tmp){
1797            // R = binarySubtraction(R, D);
1798            binarySummation(&R, S);
1799            Q.push_back(true);
1800        } else {
1801            Q.push_back(false);
```

```
1802         }
1803     }
1804 
1805     ret.exponent = binarySubtraction(ret.exponent, count_vec);
1806     //--------------------------------
1807 
1808 
1809     // rounding
1810     //--------------------------------
1811     shiftLeft(&R);
1812 
1813     char tmp = larger(R, divisor.mantissa, true);
1814     if(1 == tmp || 0 == tmp){
1815         addOneToBinary(&Q);
1816     }
1817     //--------------------------------
1818 
1819     return ret;
1820 }
1821 //--------------------------------
1822 
1823 
1824 // comparators
1825 //--------------------------------
1826 
1827 /**
1828  * Checks whether two mps objects are the same.
1829  * Info: handles exceptions and special values.
1830  *
1831  * @return true if they are the same, false otherwise
1832  */
1833 bool mps::operator==(const mps& other) const{
1834 
1835     if (this->exponent_length != other.exponent_length) {
1836         throw std::invalid_argument("ERROR: in == : Exponents do not
     match");
1837     }
1838     if (this->mantissa_length != other.mantissa_length) {
1839         throw std::invalid_argument("ERROR: in == : Mantissas do not
     match");
1840     }
1841 
1842     if(this->isNaN() || other.isNaN()){
1843         return false;
1844     }
1845 
1846     if(this->sign != other.sign) {              // positive positive case
1847         return false;
1848     } else {
1849         return equal(*this, other);
1850     }
1851 }
1852 
```

```
1853 /**
1854  * Checks whether two mps objects are not the same.
1855  * Info: handles exceptions and special values.
1856  *
1857  * @return true if they are not the same, false otherwise
1858  */
1859 bool mps::operator!=(const mps& other) const{
1860 
1861     if (this->exponent_length != other.exponent_length) {
1862         throw std::invalid_argument("ERROR: in != : Exponents do not
        match" );
1863     }
1864     if (this->mantissa_length != other.mantissa_length) {
1865         throw std::invalid_argument("ERROR: in != : Mantissas do not
        match");
1866     }
1867 
1868     if(this->isNaN() || other.isNaN()){
1869         return true;
1870     }
1871 
1872     if(this->sign != other.sign) {  // positive positive case
1873         return true;
1874     } else {
1875         return notEqual(*this, other);
1876     }
1877 }
1878 
1879 /**
1880  * Checks whether the first mps object is larger than the other mps
        object.
1881  * Info: handles exceptions and special values.
1882  *
1883  * @return true if it is larger, false otherwise
1884  */
1885 bool mps::operator>(const mps& other) const{
1886 
1887     if (this->exponent_length != other.exponent_length) {
1888         throw std::invalid_argument("ERROR: in > : Exponents do not match
        " );
1889     }
1890     if (this->mantissa_length != other.mantissa_length) {
1891         throw std::invalid_argument("ERROR: in > : Mantissas do not match
        " );
1892     }
1893 
1894     if(this->isNaN() || other.isNaN()){
1895         return false;
1896     }
1897 
1898     if(!this->sign &&!other.sign) { // positive positive case
1899         return larger(*this, other);
1900     } else if(!this->sign) {        // positive negative case
```

```
1901         return true;
1902     } else if(!other.sign){         // negative positive case
1903         return false;
1904     } else {                        // negative negative case
1905         return larger(other, *this);
1906     }
1907 }
1908
1909 /**
1910  * Checks whether the first mps object is smaller than the other mps
        object.
1911  * Info: handles exceptions and special values.
1912  *
1913  * @return true if it is smaller, false otherwise
1914  */
1915 bool mps::operator<(const mps& other) const{
1916
1917     if (this->exponent_length != other.exponent_length) {
1918         throw std::invalid_argument("ERROR: in < : Exponents do not match
        " );
1919     }
1920     if (this->mantissa_length != other.mantissa_length) {
1921         throw std::invalid_argument("ERROR: in < : Mantissas do not match
        " );
1922     }
1923
1924     if(this->isNaN() || other.isNaN()){
1925         return false;
1926     }
1927
1928     if(!this->sign &&!other.sign) { // positive positive case
1929         return smaller(*this, other);
1930     } else if(!this->sign) {        // positive negative case
1931         return false;
1932     } else if(!other.sign){         // negative positive case
1933         return true;
1934     } else {                        // negative negative case
1935         return smaller(other, *this);
1936     }
1937 }
1938
1939 /**
1940  * Checks whether the first mps object is larger than or equal to than
        the other mps object.
1941  * Info: handles exceptions and special values.
1942  *
1943  * @return true if it is larger or equal, false otherwise
1944  */
1945 bool mps::operator>=(const mps& other) const {
1946
1947     if (this->exponent_length != other.exponent_length) {
1948         throw std::invalid_argument("ERROR: in >= : Exponents do not
        match" );
```

```
1949     }
1950     if (this->mantissa_length != other.mantissa_length) {
1951         throw std::invalid_argument("ERROR: in >= : Mantissas do not
         match" );
1952     }
1953
1954     if(this->isNaN() || other.isNaN()){
1955         return false;
1956     }
1957
1958     if(!this->sign &&!other.sign) { // positive positive case
1959         return largerEqual(*this, other);
1960     } else if(!this->sign) {        // positive negative case
1961         return true;
1962     } else if(!other.sign){         // negative positive case
1963         return false;
1964     } else {                        // negative negative case
1965         return largerEqual(other, *this);
1966     }
1967 }
1968
1969 /**
1970  * Checks whether the first mps object is smaller than or equal to the
         other mps object.
1971  * Info: handles exceptions and special values.
1972  *
1973  * @return true if it is larger, false otherwise
1974  */
1975 bool mps::operator<=(const mps& other) const {
1976
1977     if (this->exponent_length != other.exponent_length) {
1978         throw std::invalid_argument("ERROR: in <= : Exponents do not
         match" );
1979     }
1980     if (this->mantissa_length != other.mantissa_length) {
1981         throw std::invalid_argument("ERROR: in <= : Mantissas do not
         match" );
1982     }
1983
1984     if(this->isNaN() || other.isNaN()){
1985         return false;
1986     }
1987
1988     if(!this->sign &&!other.sign) { // positive positive case
1989         return smallerEqual(*this, other);
1990     } else if(!this->sign) {        // positive negative case
1991         return false;
1992     } else if(!other.sign){         // negative positive case
1993         return true;
1994     } else {                        // negative negative case
1995         return smallerEqual(other, *this);
1996     }
1997 }
```

```
1998
1999
2000 /**
2001  * Performs the actual check of whether two mps objects are the same.
2002  * Info: Does not check for correct input or special values.
2003  *
2004  * @param one first mps object
2005  * @param two second mps object
2006  * @return true if they are the same, false otherwise
2007  */
2008 [[nodiscard]] bool mps::equal(const mps& one, const mps& two){
2009
2010     auto res = compare(one, two);
2011
2012     if(0 == res) {
2013         return true;
2014     } else {
2015         return false;
2016     }
2017 }
2018
2019 /**
2020  * Performs the actual check whether two mps objects are not the same.
2021  * Info: Does not check for correct input or special values.
2022  *
2023  * @param one first mps object
2024  * @param two second mps object
2025  * @return true if they are not the same, false otherwise
2026  */
2027 [[nodiscard]] bool mps::notEqual(const mps& one, const mps& two){
2028
2029     auto res = compare(one, two);
2030
2031     if(0 == res) {
2032         return false;
2033     } else {
2034         return true;
2035     }
2036 }
2037
2038 /**
2039  * Performs the actual check whether the first mps object is larger than
        the second mps object.
2040  * Info: Does not check for correct input or special values.
2041  *
2042  * @param one first mps object
2043  * @param two second mps object
2044  * @return true if the first is larger, false otherwise
2045  */
2046 [[nodiscard]] bool mps::larger(const mps& one, const mps& two){
2047
2048     auto res = compare(one, two);
2049
```

```
2050     if(-1 == res || 0 == res) {
2051         return false;
2052     } else {
2053         return true;
2054     }
2055 }
2056 
2057 /**
2058  * Performs the actual check whether the first mps object is smaller than
         the second mps object.
2059  * Info: Does not check for correct input or special values.
2060  *
2061  * @param one first mps object
2062  * @param two second mps object
2063  * @return true if the first is smaller, false otherwise
2064  */
2065 [[nodiscard]] bool mps::smaller(const mps& one, const mps& two){
2066 
2067     auto res = compare(one, two);
2068 
2069     if(1 == res || 0 == res){
2070         return false;
2071     } else {
2072         return true;
2073     }
2074 }
2075 
2076 /**
2077  * Performs the actual check whether the first mps object is larger than
       or equal to the second mps object.
2078  * Info: Does not check for correct input or special values.
2079  *
2080  * @param one first mps object
2081  * @param two second mps object
2082  * @return true if the first is larger or equal, false otherwise
2083  */
2084 [[nodiscard]] bool mps::largerEqual(const mps& one, const mps& two){
2085 
2086     auto res = compare(one, two);
2087 
2088     if(1 == res || 0 == res) {
2089         return true;
2090     } else {
2091         return false;
2092     }
2093 }
2094 
2095 /**
2096  * Performs the actual check whether the first mps object is smaller than
         or equal to the second mps object.
2097  * Info: Does not check for correct input or special values.
2098  *
2099  * @param one first mps object
```

```
2100  * @param two second mps object
2101  * @return true if the first is smaller or equal, false otherwise
2102  */
2103 [[nodiscard]] bool mps::smallerEqual(const mps& one, const mps& two){
2104 
2105     auto res = compare(one, two);
2106 
2107     if(-1 == res || 0 == res){
2108         return true;
2109     } else {
2110         return false;
2111     }
2112 }
2113 
2114 /**
2115  * Compares two mps objects.
2116  * Info: Does not check for correct input or special values.
2117  *
2118  * @param one first mps object
2119  * @param two second mps object
2120  * @return 1: first is larger; 0: both are the same; -1: second is larger
2121  */
2122 [[nodiscard]] char mps::compare(const mps& one, const mps& two){
2123 
2124     // compare exponent
2125     for(unsigned long i = 0; i < one.exponent_length; i++){
2126 
2127         if(one.exponent[i] && !two.exponent[i]){
2128             return 1;
2129         }
2130         if(two.exponent[i] && !one.exponent[i]){
2131             return -1;
2132         }
2133     }
2134 
2135     // compare mantissa
2136     for(unsigned long i = 0; i < one.mantissa_length; i++){
2137 
2138         if(one.mantissa[i] && !two.mantissa[i]){
2139             return 1;
2140         }
2141         if(two.mantissa[i] && !one.mantissa[i]){
2142             return -1;
2143         }
2144     }
2145 
2146     // equal case
2147     return 0;
2148 }
2149 //-------------------------------
2150 
2151 
2152 // helper methods
```

```
2153 //------------------------------
2154
2155 /**
2156  * Performs a binary addition using a full adder.
2157  * The binary numbers are represented as vectors containing booleans.
2158  *
2159  * @param a reference to the first addend
2160  * @param b reference to the second addend
2161  * @param carry set to true if an adjustment is wanted when the carrier
        bit of the last iteration is 1
2162  * @param carrier_return pointer to a boolean where the last state of the
         carrier bit can be saved
2163  * @return the result as a binary number
2164  */
2165 vector<bool> mps::binaryAddition(const vector<bool>& a, const vector<bool
        >& b, bool* carrier_return){
2166
2167     vector<bool> ret;
2168     bool carrier = false;
2169
2170     // full adder
2171     for(auto i = a.size(); i > 0;){
2172         i--;
2173         ret.insert(ret.begin(), (a[i] ^ b[i]) ^ carrier);
2174         carrier = ((a[i] && b[i]) || (a[i] && carrier)) || (b[i] &&
        carrier);
2175     }
2176
2177     /*
2178     // adjust if carrier bit is 1
2179     if(carry && carrier){
2180         ret.insert(ret.begin(), true);
2181         ret.pop_back();
2182     }
2183      */
2184
2185     // save the state of the carrier bit in carrier_return.
2186     if(carrier_return != nullptr){
2187         *carrier_return = carrier;
2188     }
2189
2190     return ret;
2191 }
2192
2193 /**
2194  * Performs a binary subtraction.
2195  * The binary numbers are represented as vectors containing booleans.
2196  *
2197  * @param minuend reference to the vector, which should be reduced
2198  * @param subtrahend reference to the vector which should be subtracted
2199  * @return the result as a binary number
2200  */
2201 vector<bool> mps::binarySubtraction(const vector<bool>& minuend, const
```

```
    vector<bool>& subtrahend){
2202
2203     bool carrie;
2204     auto tmp = invertAndAddOne(subtrahend, &carrie);
2205
2206     if(carrie){
2207         return minuend; // If there is a carrier bit present at the end,
    the subtrahend is zero.
2208     }
2209
2210     return binaryAddition(minuend, tmp);
2211 }
2212
2213 /**
2214  * Performs an addition but adds directly to the summand. The addend can
    be smaller.
2215  * If so, the addend will be "virtually" shifted to the left so the the
    first bits match.
2216  *
2217  * ATTENTION: The function does not check whether a carrier bit is
    present at the end,
2218  * since it is expected that the vectors either have an appropriate
    structure or that a subtraction is performed.
2219  *
2220  * @param summand pointer to the vector to which the addend is added.
2221  * @param addend reference to the addend which is going to be added to
    the summand.
2222  */
2223 void mps::binarySummation(vector<bool> *summand, const vector<bool> &
    addend, const bool prefix) {
2224
2225     bool carrier = false;
2226     bool tmp;
2227     auto j = addend.size(); // + start
2228
2229     if(prefix) {
2230         j += 2;
2231     }
2232
2233     // full adder
2234     for(auto i = addend.size(); i > 0;){
2235         i--;
2236         j--;
2237         tmp = ((*summand)[j] ^ addend[i]) ^ carrier;
2238         carrier = (((*summand)[j] && addend[i]) || ((*summand)[j] &&
    carrier)) || (addend[i] && carrier);
2239         (*summand)[j] = tmp;
2240     }
2241
2242     if(prefix) { // prefix: false, true
2243
2244         j--;
2245         tmp = !(*summand)[j] ^ carrier;
```

```
2246         carrier = ((*summand)[j] || ((*summand)[j] && carrier)) ||
         carrier;
2247         (*summand)[j] = tmp;
2248 
2249         j--;
2250         (*summand)[j] = (*summand)[j] ^ carrier;
2251     }
2252 
2253 }
2254 
2255 /**
2256  * Performs an addition of two floating point mantissas where an offset
         can also be set.
2257  *
2258  * If an offset is set, the first vector argument is shifted to the right
          (and padded on the left),
2259  * and the second vector argument is shifted to the left (and padded on
         the right).
2260  *
2261  * The padding can be set individually.
2262  *
2263  * It can be decided whether a carrier step should be made at the and.
         This is at the end of the addition of a carrier
2264  * bit is present, and an extra element (bool) will be added at the
         beginning of the resulting vector.
2265  *
2266  * Since this is an addition for floating point mantissas, it considers
         the hidden digit. This is the first bit left of
2267  * the decimal point, which is not saved since it is there by definition.
          To be able to use the code for inverted vectors
2268  * (which is needed for subtraction) the value of the hidden digit can be
          set individually.
2269  *
2270  * The last value of the carrier bit can be returned via cr.
2271  *
2272  * @param lp the vector which is padded on the left (and therefore
         shifted to the right)
2273  * @param rp the vector which is padded on the right (and therefore
         shifted to the left)
2274  * @param off_set the offset which can be set
2275  * @param c whether a carrier bit should be added at the beginning if
         present
2276  * @param p first: padding for lp, second: padding for rp
2277  * @param hd first: hidden digit for lp, second: hidden digit for rp
2278  * @param cr pointer to a bool where the last value of the carrier bit
         can be saved (default: nullptr)
2279  * @return the result as a vector consisting of booleans
2280  */
2281 vector<bool> mps::binaryOffsetAddition(const vector<bool>& lp, const
         vector<bool>& rp, unsigned long off_set, bool c, const bool p[2],
         const bool hd[2],  bool* cr){
2282 
2283     // setting up basic variable.
```

```
2284    vector<bool> ret;
2285    bool carrier = false;
2286
2287    // calculate the right padding part.
2288    //--------------------------------
2289    for(auto i = lp.size(); i > lp.size()-off_set;){
2290        i--;
2291        //ret.insert(ret.begin(), lp[i] ^ carrier);
2292        //carrier = ((lp[i] && false) || (lp[i] && carrier));
2293        ret.insert(ret.begin(), (lp[i] ^ p[1]) ^ carrier);
2294        carrier = ((lp[i] && p[1]) || (lp[i] && carrier)) || (p[1] &&
    carrier);
2295    }
2296    //--------------------------------
2297
2298
2299    // calculate the part between the padding.
2300    //--------------------------------
2301    unsigned long j = rp.size();
2302    for(auto i = lp.size()-off_set; i > 0;){
2303        i--;
2304        j--;
2305        ret.insert(ret.begin(), (lp[i] ^ rp[j]) ^ carrier);
2306        carrier = ((lp[i] && rp[j]) || (lp[i] && carrier)) || (rp[j] &&
    carrier);
2307    }
2308    //--------------------------------
2309
2310
2311    if(off_set > 0){
2312
2313        // calculate the left parts hidden digit.
2314        j--;
2315        ret.insert(ret.begin(), (hd[0] ^ rp[j]) ^ carrier);
2316        carrier = ((hd[0] && rp[j]) || (hd[0] && carrier)) || (rp[j] &&
    carrier);
2317
2318
2319        // calculate the left padding part.
2320        //--------------------------------
2321        for(; j > 0;){
2322            j--;
2323            //ret.insert(ret.begin(), rp[j] ^ carrier);
2324            //carrier = (rp[j] && carrier);
2325            ret.insert(ret.begin(), (p[0] ^ rp[j]) ^ carrier);
2326            carrier = ((p[0] && rp[j]) || (p[0] && carrier)) || (rp[j] &&
     carrier);
2327        }
2328        //--------------------------------
2329
2330
2331        // calculate right parts hidden digit
2332        ret.insert(ret.begin(), (p[0] ^ hd[1]) ^ carrier);
```

```
2333         carrier = ((p[0]  && hd[1]) || (p[0]  && carrier)) || (hd[1] &&
     carrier);
2334
2335     } else {
2336
2337         // calculate the hidden digit of both variables
2338         ret.insert(ret.begin(), (hd[0] ^ hd[1]) ^ carrier);
2339         carrier = ((hd[0] && hd[1]) || (hd[0] && carrier)) || (hd[1] &&
     carrier);
2340     }
2341
2342     // perform carrier step if wanted.
2343     if(c && carrier){
2344         ret.insert(ret.begin(), true);
2345     }
2346
2347     // save carrier bit if wanted.
2348     if(cr != nullptr) {
2349         *cr = carrier;
2350     }
2351
2352
2353     return ret;
2354 }
2355
2356 /**
2357  * Performs the rounding of the mantissa according to IEEE754.
2358  *
2359  * If an rounding upwards did take place and the following addition has a
      carry bit present at the end
2360  * the function returns true.
2361  *
2362  * @param mantissa pointer to the mantissa that should be rounded
2363  * @param mantissa_len The length to which the mantissa should be rounded
2364  * @return if an overflow happened.
2365  */
2366 bool mps::round(vector<bool> *mantissa, unsigned long mantissa_len) {
2367
2368     bool ret = false;
2369
2370     if(mantissa->size() > mantissa_len){
2371         bool tmp = false;
2372         for(auto i = mantissa_len+1 ; i < mantissa->size(); i++){
2373             if ((*mantissa)[i]){
2374                 tmp = true;
2375                 break;
2376             }
2377         }
2378         if((*mantissa)[mantissa_len] && tmp){
2379             mantissa->erase(mantissa->begin() + (long) mantissa_len,
     mantissa->end());
2380             if(addOneToBinary(mantissa)){
2381                 ret = true;
```

```
2382            }
2383        } else if((*mantissa)[mantissa_len]){
2384            mantissa->erase(mantissa->begin() + (long) mantissa_len,
    mantissa->end());
2385            if((*mantissa)[mantissa_len-1]){
2386                if(addOneToBinary(mantissa)){
2387                    ret = true;
2388                }
2389            }
2390        }else {
2391            mantissa->erase(mantissa->begin() + (long) mantissa_len,
    mantissa->end());
2392        }
2393    }
2394
2395    return ret;
2396}
2397
2398/**
2399 * Converts a binary number to an integer.
2400 * The binary number is stored in a vector of booleans.
2401 *
2402 * @param vector vector containing the binary number
2403 * @return the binary number as an integer
2404 */
2405unsigned long mps::binaryToInt(vector<bool> vector){
2406
2407    unsigned long ret = 0;
2408
2409    for(unsigned long i = 0; i < vector.size(); i++){
2410        if(vector[vector.size()-i-1]){
2411            ret += (unsigned long) pow(2,i);
2412        }
2413    }
2414
2415    return ret;
2416}
2417
2418/**
2419 * Converts an integer to a binary number.
2420 * The binary number is stored in a vector of booleans.
2421 *
2422 * @param value the value of the integer
2423 * @return the value as a binary number
2424 */
2425vector<bool> mps::intToBinary(unsigned long value) {
2426
2427    vector<bool> ret;
2428
2429    while (value >= 1){
2430        ret.insert(ret.begin(), value%2);
2431        value /= 2;
2432    }
```

```
2433
2434     return ret;
2435 }
2436
2437 /**
2438  * Returns the bias of the exponent.
2439  *
2440  * @return bias of the exponent
2441  */
2442 long mps::getBias() const{
2443     return ((long) pow (2, exponent_length)) / 2 -1;
2444 }
2445
2446 /**
2447  * Compares the size of a to the size of b.
2448  *
2449  * The return value is defined as values:
2450  * 1 if a > b
2451  * 0 if a == b
2452  * -1 if a < b
2453  *
2454  * ATTENTION: Does not check for same length!
2455  * Only works for vectors of the same size.
2456  *
2457  * Division Case: In the case of a division, the routine must account for
     "hidden" bits.
2458  *
2459  * @param a reference to the vector containing the first binary number
2460  * @param b reference to the vector containing the second binary number
2461  * @param division_case whether the division case is wanted or not
2462  * @return 1 if a > b, 0 if a == b, -1 if a < b
2463  */
2464 char mps::larger(const vector<bool>& a, const vector<bool>& b, const bool
     division_case){
2465
2466     if(!division_case){
2467         for(unsigned long i = 0; i < a.size(); i++){
2468             if(a[i] && !b[i]) { return 1;}
2469             else if(b[i] && !a[i]) {return -1;}
2470         }
2471     } else {
2472
2473         if (a[0]){
2474             return 1;
2475         } else if(!a[1]) {
2476             return -1;
2477         }
2478
2479         for(unsigned long i = 0; i < b.size(); i++){
2480             if(a[i+2] && !b[i]) { return 1;}
2481             else if(b[i] && !a[i+2]) {return -1;}
2482         }
2483
```

```
2484         if(a[a.size()-1]){
2485             return 1;
2486         }
2487     }
2488 
2489     return 0;
2490 }
2491 
2492 /**
2493  * Adds one to a binary number saved inside a vector.
2494  * The vector consists of booleans, which represent a binary number.
2495  *
2496  * @param vector pointer to the vector containing the binary number where
        one should be added
2497  * @return true if a carrier bit was present in the last iteration, false
        otherwise
2498  */
2499 bool mps::addOneToBinary(vector<bool>* vector){
2500 
2501     for(auto i = vector->size(); i > 0;){
2502         i--;
2503         (*vector)[i] = !(*vector)[i];
2504         if((*vector)[i]){
2505             return false;
2506         }
2507     }
2508 
2509     return true;
2510 }
2511 
2512 /**
2513  * Subtracts one to a binary number saved inside a vector.
2514  * The vector consists of booleans, which represent a binary number.
2515  *
2516  * @param vector pointer to the vector containing the binary number where
        one should be subtracted
2517  * @return true if a carrier bit was present in the last iteration, false
        otherwise
2518  */
2519 bool mps::subtractOneFromBinary(vector<bool> *vector){
2520 
2521     for(auto i = vector->size(); i > 0;){
2522         i--;
2523         (*vector)[i] = !(*vector)[i];
2524         if(!(*vector)[i]){
2525             return false;
2526         }
2527     }
2528 
2529     return true;
2530 }
2531 
2532 /**
```

```
2533  * Creates a copy of the vector vec. Then, it inverts the copy, adds one
          to it, and returns it afterward.
2534  *
2535  * @param vec reference to the vector which copy should be inverted, and
          added one to it
2536  * @param carrie pointer to a boolean value, which will be set to true if
           a carrier bit is present at the end
2537  * @return the resulting copied and tempered vector
2538  */
2539 vector<bool> mps::invertAndAddOne(const vector<bool> &vec, bool *carrie,
          const bool division_case){
2540
2541     vector<bool> ret;
2542
2543     if(!division_case){
2544         ret.reserve(vec.size());
2545     } else {
2546         ret.reserve(vec.size()+3);
2547         ret.push_back(true);
2548         ret.push_back(false);
2549     }
2550
2551     for(auto && i : vec){
2552         ret.push_back(!i);
2553     }
2554
2555     if(division_case){
2556         ret.push_back(true);
2557     }
2558
2559     if(carrie != nullptr){
2560         *carrie = addOneToBinary(&ret);
2561     } else {
2562         addOneToBinary(&ret);
2563     }
2564
2565     return ret;
2566 }
2567
2568 /**
2569  * Checks if a vector only consists of true booleans.
2570  *
2571  * @param vector reference to the vector, which should be checked
2572  * @return true if all entries of the vector are true
2573  */
2574 [[nodiscard]] bool mps::allTrue(const vector<bool>& vector) {
2575     return std::all_of(vector.begin(), vector.end(), [](bool i){return i
          ;});
2576 }
2577
2578 /**
2579  * Checks if a vector only consists of false booleans.
2580  *
```

```
2581  * @param vector reference to the vector, which should be checked
2582  * @return true if all entries of the vector are false
2583  */
2584 [[nodiscard]] bool mps::allFalse(const vector<bool>& vector) {
2585     return std::all_of(vector.begin(), vector.end(), [](bool i){return !i
          ;});
2586 }
2587
2588 /**
2589  * Shifts a vector to the left.
2590  *
2591  * @param vector pointer to the vector, which should be shifted to the
          left
2592  */
2593 void mps::shiftLeft(vector<bool>* vec){
2594
2595     vec->erase(vec->begin());
2596     vec->push_back(false);
2597 }
2598 //--------------------------------
```

Listing A.2: mps.cpp