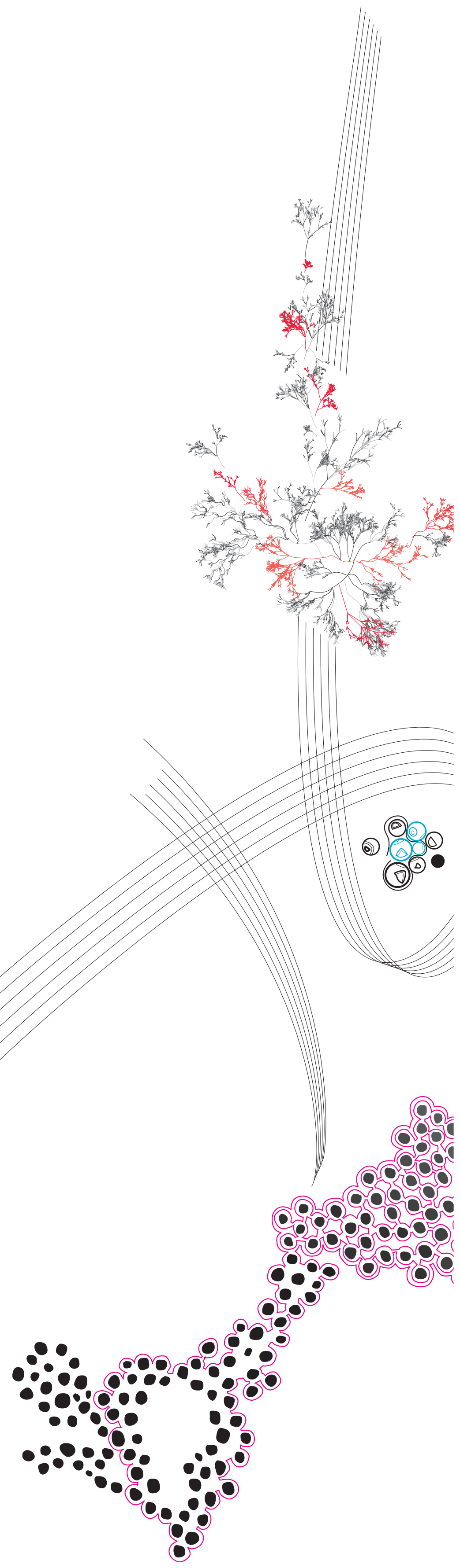
MSc Computer Science
Final Project



Error bounds for floating points in Isabelle/HOL

Jan Douwe Beekman

Supervisors: Peter Lammich
Bram Kohlen

June, 2024

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Contents

Abstract

In computer calculations, floating point numbers are commonly used to represent real numbers. The downside of this is that these numbers cannot represent each real number exactly since the numbers are stored in a finite space and thus a rounding operation is necessary. There are multiple methods to know how big the error is for a calculation. This research expresses the errors in ulps (unit in last place). A single ulp is the distance between two numbers where only the least significant bit is different. Using this measurement we prove a bound on how the error increases after operations like addition and multiplication. The goal of this research includes proving all the conclusions in Isabelle/HOL. We also apply the theorems to an algorithm. The algorithm for which we prove an error bound is iterative matrix multiplication.

Keywords: IEEE-754, Floating Point Number, Isabelle/HOL, Ulp

Chapter 1

Introduction

Real numbers are used in many different problems. An example is linear optimization, which is a mathematical problem. Programmers can also use these numbers in other problems like computer graphics or simulations. Real numbers are, however, infinitely precise, which makes it impossible to represent most of them precisely in computers that have finite memory. This is why in practice these real numbers are not used but multiple alternatives exist, examples include fixed and floating point numbers. The real number value of a fixed point number is simply a binary number multiplied by a predetermined scaling factor. In this research, however, the focus is on floating point numbers which are also called floats. We use an IEEE standard for finite precision floats which includes values for plus and minus infinity, and values that are interpreted as not-a-number [2]. This standard also describes, amongst others, rounding modes, certain operations, and error handling. CPU manufacturers have made CPUs with dedicated operations for floats following this standard as these dedicated operations generally yield better performance.

For this research, we are interested in how the floating point standard represents real numbers. Floats consist of three parts: the sign, the exponent, and the fraction. The sign part is only one bit and indicates whether the number is positive or negative. The exponent part can have any number of bits, but the standard dictates that in a 64-bit system, 11 bits are reserved for the exponent. Lastly, the fraction part uses 53 bits in a 64-bit system. To know which value is represented, the following formula can be used.

$$value = (-1)^{sign} * 2^{exponent}/2^{bias} * (1 + fraction/2^{number\_of\_fraction\_bits})$$
$$bias = 2^{number\_of\_exponent\_bits-1} - 1$$

Here we introduce a bias. If this is not introduced, only values with an absolute value above 1 can be represented. The bias determined by the IEEE standard implies that roughly half of the possible floating point values have an absolute value that is greater than 1, and roughly half of the possible values have an absolute value that is smaller than 1. Another way of writing down the floating point values is as $1.XXXXX * 2^e$. This bias would mean that e could also be smaller than 0. In a 64-bit system, the formula is the following.

$$value_{64} = (-1)^{sign} * 2^{exponent-1023} * (1 + fraction/2^{53})$$

What is shown here is only true for normal numbers. The IEEE standard also defines infinities, not-a-numbers, and subnormal numbers. A number is subnormal when the *exponent* is 0. In that case, the value is as follows.

$$value = (-1)^{sign} * 2/2^{bias} * (fraction/2^{number\_of\_fraction\_bits})$$
$$value_{64} = (-1)^{sign} * 2^{1022} * (fraction/2^{53})$$

Furthermore, when the exponent is "$2^{number\_of\_exponent\_bits} - 1$" it is infinity or not-a-number. When the $fraction$ is 0, it is either plus or minus infinity based on the $sign$. When the $fraction$ is not 0, it will be interpreted as not-a-number.

The downside is that these floats have a finite precision. This means that not all real numbers can be represented in this format. When a real number cannot be represented, it will be rounded, often to the nearest value that can be represented.

Rounding errors sometimes lead to confusing bugs and inaccuracies that can have serious consequences in a lot of fields where accurate results are crucial. One example is when a German election got different results [26]. In Germany, no party with less than 5% of the vote can get a seat in parliament. The Green party had 4.97% of the vote which was rounded to 5%. After finding out that the Green party did not pass the threshold, the seats in the parliament were recalculated. Another example is when the Vancouver stock exchange was indexed at roughly half of the true value [1]. The value of the index was calculated to three decimal places. Instead of correctly rounding to the nearest number, the computer simply dropped any digit after the three decimal places. This is a small rounding error, but if the prices are calculated 2800 times a day over 22 months, the difference gets big. These examples do not show problems due to the rounding errors of floating point numbers but due to rounding errors in general. This does show that such a tiny difference can have huge effects.

Creators of such systems are interested in preventing these kinds of bugs. One way of preventing bugs is by using theorem provers. Interactive theorem provers are tools to formalize mathematical proofs and automatically verify them. It is also possible to formalize algorithms and data structures in these systems which allows us to make mathematical statements about them. Researchers use this to verify whether software and systems are reliable. Nowadays, there is a whole range of theorem provers available. Two major ones are Coq [28] and Isabelle/HOL [27]. Due to our experience and the availability of the Isabelle Refinement Framework [20], we will use Isabelle/HOL in this research. The IRF can be used to reason about computer programs but it needs proofs on data structures to construct a proof about the computer program. In Section 3.2.1, we will explain in more detail how Isabelle works.

This research will aim to provide a method of proving an error bound for programs that use floating point numbers. We express these error bounds using ulps (unit in last place). The value of an ulp is depending on the floating point number and it represents the distance to a different floating point number were only the last bit in the fraction part is different. Section 5.1 will describe the exact definition of an ulp that we will use in this research. In Chapter 4 we go into more detail about the problem statement and goals of this research.

Chapter 2

Related Work

There are multiple formalizations of floats in theorem provers. The different focuses of these formalizations are for example arbitrary precision floats [13], fast proof checking [5, 23], or even combining other formalizations [9]. Arbitrary precision floats are in the form "$fraction * 2^{exponent}$". Here both the $fraction$ and $exponent$ parts can be positive or negative and they are not limited in the number of bits they use. We focus on the IEEE standard [2] which describes finite precision floats. Furthermore, we want to use a formalization where the proofs can be checked fast which makes the process of constructing a correct proof faster. However, we use a different formalization as these formalizations are all in Coq. The Isabelle formalization that we use [33] will be discussed in Section 3.2.2. These formalizations are also used to prove certain properties of algorithms, such as proving the accuracy of solutions for ordinary differential equations [19, 17] or the accuracy of the evaluation of polynomials [7, 11]. Sometimes mistakes are made in proofs when they are done by hand. These formalizations can help to prove these theories again in a theorem prover to catch these mistakes [25]. In some of these works arbitrary precision floating point numbers are used [17]. As we said, there is not a limit on the number of bits for these numbers. This means that they can adapt to the amount of memory needed for the precision of the real number. This approach eliminates the rounding process in many places, making proofs easier. Due to CPUs with dedicated operations finite precision floats are more efficient and many computer programs do not use arbitrary precision floats. This means that the proofs using arbitrary precision floats are less useful since they often are not a true representation of what the computer programs execute.

When floats are used, as mentioned previously, not all real numbers can be represented. This is the case for both finite precision and arbitrary precision floats. Floats can not even represent all rational numbers. This means that when floats are used in a program, the outcome often slightly differs from the outcome if real values were used. In other words, there is a range of real values that could be the intended value of the floating point value. We call this range of real numbers that could be the value if no rounding errors were introduced the error bound. Reasoning about these bounds is useful for knowing how accurate the outcome of an algorithm is. This has been done using theorem provers. Examples are a proof for the error of a single rounding operation [6], a manual proof for the error of 2 specific algorithms [29], and a general tactic that can be used in the theorem provers [34]. External tools are also created to help with this [21, 18, 12, 10]. One of these tools, Gappa, is integrated with Coq [8] such that it can be used for proving theorems regarding floating point numbers in the Coq system [15]. The strength of these methods lies in proving approximations of functions. Many mathematical functions are impossible to implement. An example is the computation of the sine function which is done using an infinite series. Going on forever is not feasible. Instead, the output needs to be accepted to be close enough at a certain point. Such external tools can be used to prove how close this output actually is to the real answer.

2.1 Interval arithmetic

One of the methods that is used to reason about error bounds of floating point numbers is interval arithmetic [22]. The basic idea behind this technique is to represent each number as an interval rather than a single value. When we define and prove all the basic operations on intervals. Using that, it is guaranteed that if the input variables are inside of the input intervals, then the output variable is in the output interval. This is useful for floating point programs since it is also possible to widen the intervals between the operations to take rounding errors into account. There are, however, downsides to this approach. For example, it would be nice to know how much this interval should be widened. Unfortunately, the rounding error differs wildly between large numbers and numbers close to zero. In static analysis, only the code is looked at and therefore the input values are not known. This means that the rounding errors are also not known. The only option for this static analysis is to assume the worst and use the large rounding error each time a rounding operation happens. Alternatively, it is also possible to calculate the rounding error alongside the normal calculation. This is called dynamic analysis. One way of doing this is rounding down for the lower bound while rounding up for the upper bound, just like how Ishii et al. [18] did. In their paper, they defined the operations on intervals in the following way:

$$\begin{aligned}
&For \diamondsuit \in \{+,-,*,\div\} (we\, assume\; 0 \notin \mathbf{y}\; when\; \diamondsuit = \div)\\
&\mathbf{x} \diamondsuit \mathbf{y} \supseteq [\\
&\qquad\quad min\{\triangledown(\underline{x} \diamondsuit \underline{y}),\ \triangledown(\underline{x} \diamondsuit \overline{y}),\ \triangledown(\overline{x} \diamondsuit \underline{y}),\ \triangledown(\overline{x} \diamondsuit \overline{y})\}\\
&\qquad max\{\triangle(\underline{x} \diamondsuit \underline{y}),\ \triangle(\underline{x} \diamondsuit \overline{y}),\ \triangle(\overline{x} \diamondsuit \underline{y}),\ \triangle(\overline{x} \diamondsuit \overline{y})\}]
\end{aligned}$$

Here a bold symbol like $\mathbf{x}$ is a range with a lower bound of $\underline{x}$ and an upper bound of $\overline{x}$. Furthermore, $\triangledown$ means rounding down to the highest lower floating point number, while $\triangle$ means rounding up. In the addition operation, it is easy to show that the minimum is just $\triangledown(\underline{x} \diamondsuit \underline{y})$ and the maximum is $\triangle(\overline{x} \diamondsuit \overline{y})$. When multiplying, however, $\triangledown(\underline{x} \diamondsuit \underline{y})$ might be the new lower bound, but it could also be $\triangledown(\underline{x} \diamondsuit \overline{y})$, $\triangledown(\overline{x} \diamondsuit \underline{y})$ or $\triangledown(\overline{x} \diamondsuit \overline{y})$ and for the upper bound it could also be any of the 4 options mentioned. One example of this is the fact that 2 big negative numbers create a big positive number when they are multiplied. Therefore, more checks need to be made to determine the bounds for a single calculation in the code. To be more specific, Ishii et al. use the rules given in Table 2.1.

This means that there is a lot of overhead when calculating error bounds and it is clear that there are enough situations where speed is important and therefore a lot of overhead is not wanted. Furthermore, you need to know the input variables to calculate the error bounds dynamically and we are interested in measuring error bounds no matter what the input is for a certain algorithm. Since both static and dynamic analysis using interval arithmetic seem to have downsides, we do not choose this approach. It would be interesting to compare this approach of absolute bounds to the approach we will use, but we did not manage to find related work on absolute error bounds for static analysis. We expect this is due to the downsides we mentioned earlier. Later we conclude that our approach will use static analysis. Related work using dynamic bounds is hard to translate to a result of static analysis which means we cannot easily compare the effectiveness of our approach to an approach using absolute bounds.

	$\underline{y} < 0, \overline{y} \leq 0$	$\underline{y} < 0, \overline{y} > 0$	$\underline{y} \geq 0, \overline{y} > 0$	$\underline{y} = \overline{y} = 0$
$\underline{x} < 0, \overline{x} \leq 0$	$[\nabla(\overline{x} \times \overline{y}),$ $\triangle(\underline{x} \times \underline{y})]$	$[\nabla(\underline{x} \times \overline{y}),$ $\triangle(\underline{x} \times \underline{y})]$	$[\nabla(\underline{x} \times \overline{y}),$ $\triangle(\overline{x} \times \underline{y})]$	[0, 0] (**)
$\underline{x} < 0, \overline{x} > 0$	$[\nabla(\overline{x} \times \underline{y}),$ $\triangle(\underline{x} \times \underline{y})]$	(*)	$[\nabla(\underline{x} \times \overline{y}),$ $\triangle(\overline{x} \times \overline{y})]$	[0, 0] (**)
$\underline{x} \geq 0, \overline{x} > 0$	$[\nabla(\overline{x} \times \underline{y}),$ $\triangle(\underline{x} \times \overline{y})]$	$[\nabla(\overline{x} \times \underline{y}),$ $\triangle(\overline{x} \times \overline{y})]$	$[\nabla(\underline{x} \times \underline{y}),$ $\triangle(\overline{x} \times \overline{y})]$	[0, 0] (**)
$\underline{x} = \overline{x} = 0$	[0, 0] (**)	[0, 0] (**)	[0, 0] (**)	[0, 0]

The cell (*) is computed as $[\min(\nabla(x \times \overline{y}), \nabla(\overline{x} \times \underline{y})), \max(\triangle(\underline{x} \times \overline{y}), \triangle(\overline{x} \times \underline{y}))]$.
Note that the multiplications of 0 and $\pm\infty$ result in NaN.
These combinations never happen in the cells not marked with (**).

TABLE 2.1: Rules to determine bounds after multiplication

2.2 Relative error

Alternatively, we can use relative bounds instead of absolute bounds. Here the bound is not simply a range of numbers, but actually a range of numbers divided by the real value. The bound could then be expressed in percentages. If a rounding operation happens, then the bounds for the error should widen no matter whether you are using an absolute or a relative bound. This rounding operation could mean that the value stored in the computer becomes closer to the actual real value, but it could also mean the opposite. There is no way to know this so we assume that the possible distance between the float and the real value becomes bigger. For small numbers, the rounding error is also a small number, but for big numbers the rounding error is big. When statically analyzing a piece of code, it is impossible to know if the value is big or small so the worst case should be assumed. Because of this, if a rounding operation happens, then the absolute bounds of the error should widen by an extreme amount. However, when the relative error is calculated, such an extreme case does not lead to an extreme error. This is because the rounding error is consistently small for normal numbers when it is expressed as a percentage. The problem here lies with subnormal numbers. Normal numbers in the IEEE look something like 1.XXXX * 2 ˆ e while subnormal numbers look like 0.XXXX * 2 ˆ e. It is clear that the distance between 1.0001 * 2 ˆ e and 1.0010 * 2 ˆ e is relatively small while the distance between 0.0001 * 2 ˆ e and 0.0010 * 2 ˆ e is very big when you express the difference in a percentage. If the real number that should be represented is actually 0.00011 * 2 ˆ e, then it could be rounded down to 0.0001 * 2 ˆ e, which would have a relative error of 33%. If this is the error propagation used for each rounding operation, then the calculated error bound is quickly becoming useless.

As will be discussed later, we want to apply our work matrix multiplication. Roux et al. have already used relative bounds for calculating the rounding error for matrix multiplication [29]. In that paper, only normal numbers are used and the mathematical foundation of this paper comes down to the following equations.

$$\begin{aligned}
&\exists \delta \in \mathbb{R}, |\delta| \leq \mathit{eps} \land \mathit{fl}(x \lozenge y) = \\
&\qquad\qquad (1+\delta)(x \lozenge y), \text{ for } \lozenge \in \{+, -\} \\
&\exists \delta, \eta \in \mathbb{R}, |\delta| \leq \mathit{eps} \land |\eta| \leq \mathit{eta} \land \mathit{fl}(x \lozenge y) = \\
&\qquad\qquad (1+\delta)(x \lozenge y) + \eta, \text{ for } \lozenge \in \{*, /\} \\
&\exists \delta \in \mathbb{R}, |\delta| \leq \mathit{eps} \land \mathit{fl}(\sqrt{x}) = (1+\delta)\sqrt{x}.
\end{aligned}$$

Chapter 8 will give formal definitions, but the fl means everything is calculated with floating point operations which include rounding. *eps* and *eta* are very small numbers based on the number of bits used for the exponent and fraction part in a floating point number. In multiplication, the η is used to represent the error that can be introduced when the outcome is rounded down to 0 even though the real value is not 0. When multiplications are chained, this results in the problem that the error equation becomes complicated. For example:

$$
\begin{aligned}
&fl(a * b) = (1+\delta_1)(a * b) + \eta_1 \\
&fl(c * d) = (1+\delta_2)(c * d) + \eta_2 \\
&fl(a * b * c * d) = (1+\delta_3)(((1+\delta_1)(a * b) + \eta_1) * \\
&\qquad\qquad\qquad\qquad\qquad ((1+\delta_2)(c * d) + \eta_2)) + \eta_3 \\
&fl(a * b * c * d) = (1+\delta_3)(1+\delta_2)(1+\delta_1)(a * b * c * d) + \\
&\qquad\qquad (1+\delta_2)(c * d)\eta_1 + (1+\delta_1)(a * b)\eta_2 + \eta_3
\end{aligned}
$$

Later in their paper some different variables and theorems are introduced to rewrite parts that look like $(1+\delta)$ into $(1+\theta)$ to get:

$$
\begin{aligned}
&fl(a * b * c * d) = (1+\theta)(a * b * c * d) + \\
&(1+\delta_2)(c * d)\eta_1 + (1+\delta_1)(a * b)\eta_2 + \eta_3
\end{aligned}
$$

When we run a program executing the code to calculate "$a * b * c * d$", we want to know what the error bounds are. We cannot use this formula for the error bound that we got by static analysis since it includes "$a * b$" and when we have executed the program, we do not know how bit "$a * b$" is. In the paper, they eventually prove that the error of a sum of products is proven to be the following.

$$|fl(\sum_{i=0}^{n-1} a_i\ b_i) - \sum_{i=0}^{n-1} a_i\ b_i| \leq \gamma_n(\sum_{i=0}^{n-1} |a_i\ b_i|) + 2n * eta$$

As can be seen here, the resulting error bound includes both a relative part and an absolute part. When this matrix is multiplied more than once, the error bound would become complicated, just like the small example of "$a * b * c * d$". For this reason, as well as the fact that it is difficult to use relative bounds for subnormal numbers, we again choose a different approach. Since we use a different approach, this work and other work on relative error bounds are not easily translatable to our work. This work does however give an error bound for a dot product. Since we will also calculate an error bound for a dot product, we will compare these results in Chapter 8.

Chapter 3

Preliminary Work

3.1 Unit in last place

Since we concluded in Section 2.1 and 2.2 that both an absolute and a relative bound have their downsides, we choose to express the bound in ulps (unit in the last place). Researchers have defined ulps in different ways. Some definitions are based on a real value instead of being based on the floating point value. An example is the following definition [16]. The definition includes the Hilbert choice operator ε. This operator should be read as 'εe. ... ' as 'some e such that ... '.

$$\begin{aligned}&\text{binade } x = \varepsilon e.\text{abs}(x) <= 2^{e+precision-bias} \wedge\\&\quad \forall e'.\text{abs}(x) <= 2^{e'+precision-bias} \Rightarrow e <= e'\\&\text{ulp } x = 2^{(\text{binade } x)-bias}\end{aligned}$$

The paper using that definition uses a slightly different definition for the value of a floating point meaning precision and bias are different. Instead of 1.XXX $*2^{e-bias}$ they use a format of XXX $*2^{e-bias}$. Even though that definition is different, they still use finite precision floats and not arbitrary precision floats. The definition of an ulp obtains the smallest exponent that is big enough to represent the real value as a float. Two raised to the power of this exponent is already the value of an ulp since adding or subtracting 1 from XXX results in a value with a difference of 1 ulp of the previous floating point number. In practice, this definition means that when a real value is inside of a gap between two adjacent floating point numbers, 1 ulp is the size of that gap. If the real value is exactly representable as a floating point number, 1 ulp is the size of the smallest adjacent gap.

In another example, an ulp is not based on a value but only on the precision of the floating point standard and thus only depends on the number of bits used for the fraction part. The definition then would become $ulp = 2^{-52}$ for a 64-bit floating point number [24].

We are interested in an ulp definition that depends on the value of the floating point number. Therefore, we use the definition that is already defined in the Isabelle/HOL theorem prover. This definition is constructed in such a way that 1 ulp represents the distance between a floating point number and its closest neighbor which has a greater absolute value. This definition is also used by others [14, 30]. We explain the definition in Isabelle/HOL in Section 5.1. A nice property of ulps is that they can be used for both normal numbers and subnormal numbers. If e represents the exponent encoded in the floating point value, this means that the distance between 1.00...001 * 2^e and 1.00...010 * 2^e is 1 ulp and the distance between 0.00...001 * 2^e and 0.00...010 * 2^e is also 1 ulp.

Since this is based on which values are representable by floats, the error introduced by each rounding is very small. Depending on the rounding mode, it can be as small as only

0.5 ulp. This rounding error can be proven for hardware designs [32, 30], but ulps can also be used for software programs. To our knowledge, this has not been done. That is why the idea behind this research is to construct formulas on how the error bounds expressed in ulps propagate through the basic operations. Once this is proven, this method of error expression can be used to prove software algorithms more easily.

3.2 Isabelle formalizations

As we mentioned earlier, we use Isabelle/HOL for this research. We build further on formalizations from others. More specifically, we build further on a formal model of IEEE floating point arithmetic by Lei Yu [33], and on matrices, Jordan normal forms, and spectral radius theory by René Thiemann and Akihisa Yamada [31]. Before we explain these formalizations, we first explain Isabelle itself more. After that, we will explain the parts we use the most for our proofs to differentiate between our work and theirs.

3.2.1 Isabelle syntax

If we formalize data structures, we need to define new types in Isabelle. There are multiple ways to do this, but we only use the "typedef" keyword. In the places we use this keyword, the definition of the type is quite different so we explain those definitions in those places. The alternative would be to use a "datatype" where the type follows one of serveral patterns which may even be recursive.

After defining the types to represent data structures, we define functions using these types. Again, there are multiple ways to do that. We use "definition", "lift_definition", and "fun". One example of where we use "definition" is the following:

definition ulp :: "$('e,\ 'f)$ float $\Rightarrow$ real"
    where "ulp a = valof (one_lp a) $-$ valof (zero_lp a)"

Here we start with the keyword "definition", followed by the name of the function "ulp" and the type of the function. This type describes both the inputs and the output where each one is separated with a $\Rightarrow$ symbol. Therefore we can see the "ulp" function takes a "$('e,\ 'f)$ float" as input and outputs a "real". After that the meaning of the function follows. Here arguments are given a name and used to determine what the output is. Next, we look at an example of a "lift_definition".

lift_definition zero_lp :: "$('e,\ 'f)$ float $\Rightarrow$ $('e,\ 'f)$ float"
    is "$\lambda(s, e, f).(0, e, 0)$"

A "lift_definition" also starts with a name and type. What follows, is quite different. Here we use a lambda function. This function has one argument as input which is interpreted as 3 different parts, namely s, e, and f. This also shows the difference between a "definition" and a "lift_definition". A floating point number can be an abstract concept, but a "lift_definition" uses the actual representation and thus we can split it up into the sign, exponent, and fraction parts of the floating point number.

Lastly, functions can also be defined with the "fun" keyword. An exmaple is the following.

```
fun round :: "roundmode ⇒ real ⇒ ('e, 'f) float"
where
     "round $To\_nearest$ $y$ =
       (if $y \leq -$threshold $TYPE(('e,\ 'f)$ float) then minus_infinity
       else if $y \geq$ threshold $TYPE(('e,\ 'f)$ float) then plus_infinity
       else closest (valof) ($\lambda a$. even (fraction $a$)) {$a$. is_finite $a$} $y$)"
   | "round $float\_To\_zero$ $y$ = ...
   | "round $To\_pinfinity$ $y$ = ...
   | "round $To\_ninfinity$ $y$ = ...
```

The difference between a "fun" and a "definition" is that we can use pattern matching. Here it is used to see if the first argument is $To\_nearest$, $flot\_To\_zero$, $To\_pinfinity$, $To\_ninfinity$, where the cases are split by a "|". This example is not fully written out, but the ... represent the definitions for the other rounding modes.

Lastly, we explain the syntax for constructing proofs. We do this by going over the following proof.

```
lemma sum_powers_matrix_multiplication_error :
shows "($\Sigma i = 1..k.\ (2*x)\hat{}i) = ((2*x)\hat{}(k+1) - 1)/(2*x - 1) - 1$"
     apply(cases "$x = 0$")
     apply(induction $k$)
     apply auto
     subgoal proof −
          have "$0 < x \Rightarrow 1 < 2*x$" by linarith
          with sum_powers_1[where ...] show ?thesis by fastforce
     qed done
```

There are two different types of proofs in Isabelle. The first one is often called "apply style". We use this method to split the goal up into two cases and this means that after the first line of the proof, the goals are the following.

```
goal (2 subgoals) :
1.$x = 0 \Rightarrow$
     real (sum ((ˆ) ($2*x$)) {1..$k$}) =
     ((2 ∗ real $x$)ˆ($k+1$) − 1)/(2 ∗ real $x$ − 1) − 1
2.$x \neq 0 \Rightarrow$
     real (sum ((ˆ) ($2*x$)) {1..$k$}) =
     ((2 ∗ real $x$)ˆ($k+1$) − 1)/(2 ∗ real $x$ − 1) − 1
```

In the next apply statement, we use induction. Many statements in this style only influence the top goal. This means that we only apply induction to the first goal where we assume that x is equal to zero. Since an induction proof has two cases, namely a base case and the induction step, this first goal is split up into two new goals.

goal (3 subgoals) :
1.$x = 0 \Rightarrow$
  real (sum ((^) $(2 * x)$) {1..0}) =
  $((2 * \text{real } x)\hat{}(0 + 1) - 1)/(2 * \text{real } x - 1) - 1$
2. $\bigwedge k :: nat.$
  $(x = 0 \Rightarrow$
  real (sum ((^) $(2 * x)$) {1..k}) =
  $((2 * \text{real } x)\hat{}(k + 1) - 1)/(2 * \text{real } x - 1) - 1) \Rightarrow$
  $x = 0 \Rightarrow$
  real (sum ((^) $(2 * x)$) {1..Suc k}) =
  $((2 * \text{real } x)\hat{}(\text{Suc } k + 1) - 1)/(2 * \text{real } x - 1) - 1$
3.$x \neq 0 \Rightarrow$
  real (sum ((^) $(2 * x)$) {1..k}) =
  $((2 * \text{real } x)\hat{}(k + 1) - 1)/(2 * \text{real } x - 1) - 1$

In the third statement we apply "auto". This is one of the statements which applies to all the open goals. In this example, this method can completely proof the first two goals such that only one remains.

goal (1 subgoals) :
1.$0 < x \Rightarrow$
  $(\Sigma xa = 0..k.\ 2\hat{}xa * \text{real } x\hat{}xa) =$
  $(2 * \text{real } x * (2\hat{}k * \text{real } x\hat{}k) - 1)/(2 * \text{real } x - 1)$

For this last goal, we use the other type of proof. This method is called "Intelligible semi-automated reasoning" or Isar. In this style, facts can be proven seperatly. For example, the first line of this type of proof, proves that "$0 < x \Rightarrow 1 < 2 * x$". This fact is proven "by linarith" which is short for "apply linarith done", meaning that we use the previous style to prove an individual fact. The next line starts with "with". This means that we use the previous fact and a lemma called "sum_powers_1" to prove a new fact. Instead of using the word "have" followed by a fact in quotes, we use "show ?thesis". This simply means that the fact we try to prove is the end goal. After this part is finished so we end it with the keyword "qed". Lastly the complete lemma is proven so we finish it with the keyword "done".

3.2.2 Floating point arithmetic

First we look at the definition for floats.

typedef (overloaded) ($'e$ :: len, $'f$ :: len) float =
  "UNIV :: (1 word $\times$ $'e$ word $\times$ $'f$ word) set"

Here, a float is defined as a combination of three words. A word is a string of a certain number of bits. The first word represents the sign bit and only needs a word with a single bit. After that, there are bits allocated to represent the exponent. Here $'e$ bits are used. As can be seen, $'e$ is of the type class 'len'. Practically this means that it is a natural number. After that, there are also bits for the fraction of the float. This uses $'f$ bits. Because $'e$ and $'f$ are of the type class 'len', we can extract how big they are as number of the type 'nat'. We use the functions $LENGTH('e)$ and $LENGTH('f)$ for this.

When we have a definition for a floating point number, we also need a method of interpreting a floating point number. The first step is to read the values of the sign, fraction, and exponent parts. This can be done with the following functions.

```
lift_definition sign :: "('e, 'f) float ⇒ nat" is
    "λ(s :: 1 word, _ :: 'e word, _ :: 'f word). unat s".

lift_definition exponent :: "('e, 'f) float ⇒ nat" is
    "λ(_, e :: 'e word, _). unat e".

lift_definition fraction :: "('e, 'f) float ⇒ nat" is
    "λ(_, _, f :: 'f word). unat f".
```

These three functions take as input a float, and have as output a natural number. They interpret a float as three parts or in other words, the sign, fraction, and exponent parts. The functions take the word and use the 'unat' function to return the word as an unsigned natural number. This returns the values of the bits representing the sign, fraction, and exponent parts as a natural number.

As was described in Chapter 1, before we can calculate the value of a floating point number, we need to know what the bias is. The bias is used to decrease the exponent and thus to represent numbers with an absolute value that is smaller than 1. In Isabelle, the bias is defined in the following way.

```
definition bias :: "('e, 'f) float itself ⇒ nat"
    where "bias x = 2^(LENGTH('e) − 1) − 1"
```

This leads us to the following definition for the value of a floating point number.

```
lift_definition valof :: "('e, 'f) float ⇒ real"
is "λ(s, e, f).
    let x = (TYPE(('e, 'f) float)) in
    (if e = 0
     then (−1 :: real)^(unat s) ∗ (2/(2^bias x)) ∗ (real_of_word f/2^(LENGTH('f)))
     else (−1 :: real)^(unat s) ∗ ((2^(unat e))/(2^bias x))∗
                    (1 + real_of_word f/2^LENGTH('f)))"
```

Here we see that the value of a float depends on whether the exponent is 0 since that decides if it is a normal or subnormal number. Furthermore, we can see that this definition is the same as the definitions given in Chapter 1. An interesting part of this formula is that the function can also be calculated for a value representing infinity or not-a-number. This does not make a lot of sense since these numbers do not represent a real value according to the IEEE standard. In practice, you should not use this "valof" function when the float is infinity or not-a-number, but this formalization is structured in such a way that you need to check that yourself before using the "valof" function. The IEEE standard [2] dictates when a number is finite, infinite and not-a-number. These rules are shown in table 3.1. The Isabelle formalization has the following functions to determine which type a floating

exponent	**fraction = 0**	**fraction ≠ 0**
000..000	$\pm zero$	subnormal number
000..001 - 111..110	normal value	
111..111	$\pm infinity$	not-a-number

TABLE 3.1: Rules to determine bounds after multiplication

point number is.

definition is_nan :: "$('e,\ 'f)$ float $\Rightarrow$ bool"
    where "is_nan $a \leftrightarrow$ exponent $a =$ emax $TYPE(('e,\ 'f)$ float$) \wedge$ fraction $a \neq 0$"

definition is_infinity :: "$('e,\ 'f)$ float $\Rightarrow$ bool"
    where "is_infinity $a \leftrightarrow$ exponent $a =$ emax $TYPE(('e,\ 'f)$ float$) \wedge$ fraction $a = 0$"

definition is_normal :: "$('e,\ 'f)$ float $\Rightarrow$ bool"
    where "is_normal $a \leftrightarrow 0 <$ exponent $a \wedge$ exponent $a <$ emax $TYPE(('e,\ 'f)$ float$)$"

definition is_denormal :: "$('e,\ 'f)$ float $\Rightarrow$ bool"
    where "is_denormal $a \leftrightarrow$ exponent $a = 0 \wedge$ fraction $a \neq 0$"

definition is_zero :: "$('e,\ 'f)$ float $\Rightarrow$ bool"
    where "is_zero $a \leftrightarrow$ exponent $a = 0 \wedge$ fraction $a = 0$"

definition is_finite :: "$('e,\ 'f)$ float $\Rightarrow$ bool"
    where "is_finite $a \leftrightarrow$ (is_normal $a \vee$ is_denormal $a \vee$ is_zero a)"

The words denormal and subnormal can be used interchangeably. Here the function 'emax' is used. We know that a number is infinity or not-a-number when the bits used for the exponent are all 1. The formal definition is as follows.

definition emax :: "$('e,\ 'f)$float itself $\Rightarrow$ nat"
    where "emax $x =$ unat $(-1 ::\ 'e$ word$)$"

All of this is useful to reason about floating point numbers, but it is not enough to reason about floating point algorithms. Since rounding is needed for that, we explain that first.

```
fun round :: "roundmode ⇒ real ⇒ ('e, 'f) float"
where
    "round $To\_nearest$ $y$ =
     (if $y \leq -$threshold $TYPE(('e,\ 'f)$ float) then minus_infinity
     else if $y \geq$ threshold $TYPE(('e,\ 'f)$ float) then plus_infinity
     else closest (valof) ($\lambda a.$ even (fraction $a$)) {$a.$ is_finite $a$} $y$)"
     |...|...|...
```

The rounding function is not only defined for the $To\_nearest$ mode, but also for $float\_To\_zero$, $To\_pinfinity$, and $To\_ninfinity$. We are not interested in those so we represent these parts of the function with dots. There are a few new terms introduced here. First, we see "threshold". This is the number representing the boundary between the numbers that are rounded towards infinity and those that are not. To be more specific, the "threshold" function is defined as follows:

definition threshold :: ”$('e,\ 'f)$ float itself ⇒ real”
  where ”threshold $x = (2\hat{}(\text{emax } x - 1)/2\hat{}\text{bias } x) * (2 - 1/(2\hat{}(\text{Suc}(\text{fracwidth } x))))$”

abbreviation fracwidth :: ”$('e,\ 'f)$ float itself ⇒ nat” where
  ”fracwidth _ $\equiv LENGTH('f)$”

Next, we look at the formalization of "closest".

definition ”closest $v\ p\ s\ x =$
  (SOME a. is_closest $v\ s\ x\ a \wedge ((\exists b.$ is_closest $v\ s\ x\ b \wedge p\ b) \rightarrow p\ a))$”

The "SOME" operator returns a value for which the function is true. It does not necessarily mean that the function is only true for one value or that there even exists a value for which it is true. In this application, there are a few arguments such as s which is a set of floating point values the returned value should belong to. If this is empty, then there will be no possible floating point value to return in the "closest" function. The Isabelle formalization only uses the "closest" function with arguments such that there exists at least one possible value to return. We found at least one case where multiple floating point values could be returned by the "closest" function. To be more specific, in every case where 0 is the closest value, -0 is also the closest value.

Knowing what the "SOME" operator means, we can see that he "closest" function returns a value that "is_closest" and when there is a value that "is_closest" for which a tie-breaker p is true, then the tie-breaker should also be true for the returned value. It is important to mention that this is a formalization that cannot be executed in Isabelle. Implementations of the IEEE standard should result in a floating point number that follows these rules, but due to operators like "SOME" and "$\exists$", Isabelle cannot give the rounded floating point value. The "is_closest" function is defined as follows.

definition ”is_closest $v\ s\ x\ a \leftrightarrow a \in s \wedge (\forall b. b \in s \rightarrow |v\ a - x| \leq |v\ b - x|)$”

To put it into words, this claims that a value a is the closest value to x when it is in set s and when all other values in s have a greater or equal distance. This is because v is the function "valof" in all the cases when this function is called. Furthermore, we know that the set s is the set of all finite numbers since that is how the rounding function is defined for the mode $To\_nearest$.

With these definitions, it is possible to define the basic operators. These operators are defined in such a way that they first take care of all the edge cases that lead to infinity or not-a-number, and then they return the rounded value of the basic operation executed on real numbers. The "fmul_add" operator shows that there are 10 cases where the returned value is not a finite floating point nummber.

```
definition fmul_add :: "roundmode ⇒ ('t, 'w) float ⇒ ('t, 'w) float ⇒ "
                              ('t, 'w) float ⇒ ('t, 'w) float
    where "fmul_add $mode$ $x$ $y$ $z$ = (let
        $signP$ = if sign $x$ = sign $y$ then 0 else 1;
        $infP$ = is_infinity $x$ ∨ is_infinity $y$
    in
        if is_nan $x$ ∨ is_nan $y$ ∨ is_nan $z$ then some_nan
        else if is_infinity $x$ ∧ is_zero $y$∨
                    is_zero $x$ ∧ is_infinity $y$∨
                    is_infinity $z$ ∧ $infP$ ∧ $signP \neq$ sign $z$
            then some_nan
        else if is_infinity $z$ ∧ (sign $z = 0$) ∨ $infP$ ∧ ($signP = 0$)
            then plus_infinity
        else if is_infinity $z$ ∧ (sign $z = 1$) ∨ $infP$ ∧ ($signP = 1$)
            then minus_infinity
        else let
            $r1$ = valof $x$ ∗ valof $y$;
            $r2$ = valof $z$;
            $r = r1 + r2$
        in
            if $r = 0$ then (
                if $r1 = 0 \wedge r2 = 0 \wedge signP$ = sign $z$ then zerosign $signP$ 0
                else if $mode = To\_ninfinity$ then −0
                else 0
            ) else (
                zerosign (if $r < 0$ then 1 else 0)(round $mode$ $r$)
            )
    )"
```

One new function is introduced here and that is "zerosign". This is a function to determine the difference between 0 and -0. Since both are representable in the IEEE standard, the formalization has rules to decide which one the result needs to be. This function only changes the number when the value is already 0 since it is defined in the following way.

```
definition zerosign :: "nat ⇒ ('e, 'f) float ⇒ ('e, 'f) float"
    where "zerosign $s$ $a$ =
        (if is_zero $a$ then (if $s = 0$ then 0 else −0) else $a$)"
```

3.2.3 Vectors and matrices

The approach taken by Lei Yu to define vectors and matrices is to use functions. This is different from a pattern-based approach. Take for example lists. In Isabelle, lists are defined as follows.

```
datatype (set : 'a) list =
     Nil
    | Cons (hd : 'a) (tl : "'a list")
```

Here a list is either an empty list, or it is an element followed by another list. A vector is defined completely differently as it can be seen as a function with an index as input. To be more specific, a vector is defined in the following way.

> typedef $'a$ vec = "$\{(n,\ \text{mk\_vec}\ n\ f) \mid n\ f ::\ \text{nat} \Rightarrow\ 'a.\ \text{True}\}$"
>
> lift_definition vec :: "nat $\Rightarrow$ (nat $\Rightarrow\ 'a) \Rightarrow\ 'a$ vec"
> is "$\lambda n\ f.\ (n,\ \text{mk\_vec}\ n\ f)$"

As can be seen, a vector is a tuple. Before explaining the newly introduced function "mk_vec", we first explain the meaning of the elements of the tuple. For that, the following definitions exist.

> lift_definition dim_vec :: "$'a$ vec $\Rightarrow$ nat" is fst.
> lift_definition vec_index :: "$'a$ vec $\Rightarrow$ (nat $\Rightarrow\ 'a)$"(infixl "$" 100) is snd.

Here we see that the first element is retrieved with "dim_vec". In other words, the first element represents the dimension of the vector. The second element is retrieved with the function "vec_index". The returned value is a function with as input a natural number, and as output something of type $'a$ when the elements of the vector are of the type $'a$. The "vec_inex" therefore gives the following meaning to the second element of the tuple. The second element is a function that gets an index as input and returns the value on that spot in the vector.

There is only one thing left unexplained. That is the "mk_vec" function. This is defined like this.

> definition mk_vec :: "nat $\Rightarrow$ (nat $\Rightarrow\ 'a) \Rightarrow$ (nat $\Rightarrow\ 'a)$" where
> "mk_vec $n\ f \equiv \lambda i.$ if $i < n$ then $f\ i$ else undef_vec $(i - n)$"
>
> definition undef_vec :: "nat $\Rightarrow\ 'a$" where
> "undef_vec $i \equiv [\,]\ !\ i$"

As can be seen, the "mk_vec" function is used to take care of the cases where the index is out of bounds. Using this implementation, no index comparisons have to be performed in different parts of the formalization. With these definitions, we can define useful functions like a scalar product.

> definition scalar_prod :: "$'a$ vec $\Rightarrow\ 'a$ vec $\Rightarrow\ 'a$ :: semiring_0" (infix " $\bullet$ " 70)
> where "$v \bullet w \equiv \Sigma i \in \{0..<\ \text{dim\_vec}\ w\}.v\$i * w\$i$"

This scalar product (or dot product) is very useful for our application since the multiplication between a stochastic vector and matrix will use this function. Sadly, this function will use basic operations like addition and multiplication definitions that work slightly differently from the operations for floats. This is because addition and multiplication are executed seperately, but for floating point numbers, the "fmul_add" operator exists. This function multiplies the first two inputs numbers and adds a third input and thus takes 3 inputs. Furthermore, every floating point operation needs a rounding mode as an extra input. This means that we can use this specific "scalar_prod" for the calculation over reals, but for floats, we will need to define our own functions later.

Matrices are defined in a very similar way to vectors. The biggest difference is that a vector is defined as a 2-tuple but a matrix is defined as a 3-tuple. Technically, a 3-tuple is defined as a 2-tuple, where the second element is a 2-tuple. This makes reading the values slightly more complex as reading the second element is actually reading the first element of the second element. The definitions for matrices look as follows.

typedef $'a$ mat = "$(nr,\ nc,$ mk_mat $nr\ nc\ f)$ | $nr\ nc\ f$ :: nat × nat ⇒ $'a$. True"

lift_definition mat :: "nat ⇒ nat ⇒ (nat × nat ⇒ $'a$) ⇒ $'a$ mat"
    is "$\lambda nr\ nc\ f.$ $(nr,\ nc,$ mk_mat $nr\ nc\ f)$"

definition mk_mat :: "nat ⇒ nat ⇒ (nat × nat ⇒ $'a$) ⇒ (nat × nat ⇒ $'a$)" where
    "mk_mat $nr\ nc\ f \equiv \lambda(i,j)$. if $i < nr \wedge j < nc$ then $f\ (i,j)$
        else undef_mat $nr\ nc\ f\ (i,j)$"

lift_definition dim_row :: "$'a$ mat ⇒ nat" is fst.
lift_definition dim_col :: "$'a$ mat ⇒ nat" is "fst ∘ snd".
lift_definition index_mat :: "$'a$ mat ⇒ (nat × nat ⇒ $'a$)"(infixl "$$" 100) is "snd ∘ snd".

Again, these definitions can then be used to define many functions. The functions we use the most in this research are the following.

definition row :: "$'a$ mat ⇒ nat ⇒ $'a$ vec" where
    "row $A\ i$ = vec (dim_col A) ($\lambda j.$ A $$ (i,j))"

definition mult_mat_vec :: "$'a$ :: semiring_0 mat ⇒ $'a$ vec ⇒ $'a$ vec"(infixl " $*_v$ " 70)
    where "$A *_v\ v \equiv$ vec (dim_row A) ($\lambda i.$ row $A\ i \bullet v$)"

The function "row" returns a specific row of the matrix and "mult_mat_vec" is a dot product between a matrix and a vector.

Chapter 4

Problem statement

It appears like nearly all the work that has been done in theorem provers regarding floating point numbers was done using Coq. Coq proofs are not easily convertible to equivalent proofs in other theorem provers. Proving theorems in Isabelle therefore means, we construct our proofs from scratch. A formalization of floating point numbers is already available for Isabelle [33], but to the best of our knowledge, no work has been done for accuracy bounds. Isabelle has the potential to verify software systems by using the Isabelle Refinement Framework [20] (IRF), but such a framework can not easily help with a proof if fundamental properties are not already formalized. This means that if Coq proofs existed that achieve our goals, it would still be valuable work to replicate them in Isablle/HOL since we can then combine them with existing work in the IRF in new ways that are not directly possible in Coq. We did not find those proofs, but obviously, we can also formalize new lemmas in Isablle/HOL to achieve the same value. Furthermore, we expect this work to be valuable since our approach uses ulps for error propagation, and to our knowledge, this has not been formalized in any theorem prover.

Therefore, this research aims to create a library that assists the user in proving the accuracy bounds of algorithms. We do this by providing the essential lemmas on how errors propagate through the basic operators like addition and multiplication, as well as how the errors are introduced in the rounding operator which happens after each calculation.

We expect that formalizing this in Isabelle is a useful step in the research related to formal proofs using theorem provers. This usefulness, however, should also be shown by giving an example proof using the created lemmas, and thus the goal includes a proof of error bounds for a method of calculating probabilities. To be more specific, this method includes the multiplication of vectors and matrices which represent a Markov chain [3]. By doing this iteratively, the resulting vector should stabilize and give a probability distribution.

4.1 Research Questions

To conclude, we formulate the following research questions:

- What are the necessary lemmas to prove an error bound for algorithms measured in ulps?
- Is it possible to construct a proof for the error bound measured in ulps of a program that calculates a probability distribution over a Markov chain using these lemmas?
- How tight are the error bounds constructed using these lemmas?

Chapter 5

Approach

To achieve the goals of this research, we divide our work into four steps. First, we need to formulate how we express the error bounds. Secondly, we need to understand how rounding works in floating point numbers. This makes it possible to understand how errors are introduced. The third step is about how errors propagate through basic operations like addition and multiplication. Lastly, the fourth step applies this to an algorithm of matrix multiplication.

5.1 Error expression

As we said in the research questions in Section 4.1, the goal is to express the error bound using ulps. This is why we defined the following function:

definition ulp_accuracy :: "real $\Rightarrow$ $('e,\ 'f)$ float $\Rightarrow$ real $\Rightarrow$ bool"
where "ulp_accuracy $a\ c\ u \equiv$ (is_finite c) $\wedge$ $|(\text{valof } c - a)| \leq u * (\text{ulp } c)$"

Here a is the abstract value which is a real number, c is the concrete value which is a floating point number, and u is a real number that describes the accuracy expressed in ulps. There are a couple of functions used here. The function "is_finite" is true when a floating point number is finite and thus, it is neither infinite nor is it not-a-number. The valof function takes a floating point value and returns the value as a real number. The full definitions of these 2 functions are already given in Section 3.2.2. The framework in Isabelle from Lei Yu [33] has a definition for the ulp function. The definition was as follows:

definition ulp :: "$('e,\ 'f)$ float $\Rightarrow$ real"
where "ulp a = valof (one_lp a) $-$ valof (zero_lp a)"

lift_definition zero_lp :: "$('e,\ 'f)$ float $\Rightarrow$ $('e,\ 'f)$ float"
is "$\lambda(s, e, f).(0, e, 0)$"

lift_definition zero_lp :: "$('e,\ 'f)$ float $\Rightarrow$ $('e,\ 'f)$ float"
is "$\lambda(s, e, f).(0, e, 1)$"

The definitions for "zero_lp" and "one_lp" are functions that take a floating point number as input. The sign bits will be set to 0, the exponent bits will not change and the fraction bits will be set to the value 0 and 1 respectively. The distance between these 2 values is indeed the value of 1 ulp, and is equal to the distance to the closest representable value.

This definition can be difficult to work with so we have also proven a lemma to show how big an ulp is.

> ulp (c :: ($'e$, $'f$) float) =
> 2^(max (exponent c) 1)/2^(2^($LENGTH('e)-1)-1+LENGTH('f)$)

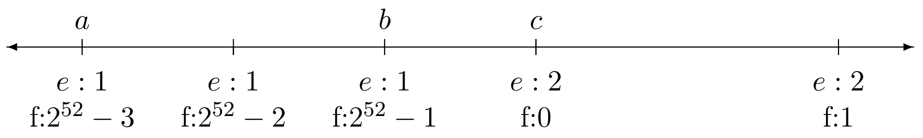
Here $LENGTH('e)$ is the number of bits used to encode the exponent part of the floating point number and $LENGTH('f)$ is the number of bits for the fraction part.

5.2 Error introduction

The second step is formalizing the mathematics behind the rounding error. We know that an error could be introduced each time an operation is performed since the floating point standard dictates that the value should then be rounded. Obviously, we need to also know how big that error is to determine how big the error of the output of the algorithm is. We use the mathematical definition for the values of floating point numbers to determine which real numbers can be represented precisely. Next, we determine how big the gaps are between these numbers.

We did not expect much difficulty here since 1 ulp represents the distance between two floating point numbers. It is intuitive to think when the rounding mode is set to round to the nearest number, the maximum error would be 0.5 ulp. We have proven that this is indeed the case, but it took more time to prove than expected.

First, we proved that when a real number is between the thresholds for a floating point standard, or in other words when it is not rounded to plus or minus infinity, there exists a floating point number for which the real number is in a range of 0.5 ulp. This is not enough to prove that the closest value has an accuracy of 0.5 ulp. A lower accuracy measured in ulps does not necessarily mean a smaller distance and a smaller distance does not necessarily mean a lower accuracy measrued in ulps. Take for example the following numbers.



If a is a real number, then it is 2 ulps away from the floating point number b, but only 1.5 ulps away from the float point number c. This is the case because the exponent of c is bigger and thus the size of an ulp is twice as big. We did show the following lemma.

> assumes "ulp_accuracy a (b :: ($'e$, $'f$) float) 0.5"
>   and "$1 < LENGTH('f)$"
>   and "$\forall$(b :: ($'e$, $'f$) float).$b \in a$. is_finite $a \rightarrow$
> |valof (c :: ($'e$, $'f$) float) $-$ a| $\leq$ |valof $b - a$|"
>   and "is_finite c"
>   shows "ulp_accuracy a c 0.5"

Lemma 5.1: Being closest implies an accuracy of 0.5 ulp

As explained in Section 5.1, "ulp_accuracy a b 0.5" means that the distance between a and b is less or equal to 0.5 ulp.

This lemma shows that when you have a float b that has a distance of 0.5 ulp or less to the real a, and you have a float c which is closer or has an equal distance to a, then it also has a distance of 0.5 ulp or less to the real a. Here it does not matter if the ulp of b is smaller, equal or bigger than the ulp of c.

We have proven this using a very long proof, which leads us to hope there was a more efficient proving method, but it is enough to make useful conclusions. This proof will be explained in detail in Section 7.1. From this proof, we can conclude that if there exists a floating point number with an accuracy of 0.5 ulp, then the float to which the real number will be rounded will also have an accuracy of 0.5 ulp. Together with what we first had proven, namely that there exists a floating point number with an accuracy of 0.5 ulp, we can easily prove the following conclusion.

```
    assumes "$|a| <$ threshold $TYPE(('e,\ 'f)$ float)"
        and "$a\_r = (($round $To\_nearest\ a) :: ('e,\ 'f)$ float)"
        and "$1 < LENGTH('e)$"
        and "$1 < LENGTH('f)$"
      shows "ulp_accuracy $a\ a\_r$ 0.5"
```

Lemma 5.2: Rounding creates an accuracy of 0.5 ulp

Here we have a lemma about a real number a and a floating point number a_r. This floating point number is the number to which a is rounded. This lemma claims that when the absolute value of a is small enough such that a_r is not infinite, then the distance between a and a_r is less or equal to 0.5 ulp.

5.3 Error propagation

After that, the third step is to formalize how the error bounds propagate through the basic operations. We encountered some difficulties here. Subtraction gave one of these difficulties. It can occur that 2 different big numbers are both rounded to the same number. Take for example the following 2 numbers. First we have $a : 2^{1023}$, and secondly we have $b : 2^{1023} * (1 + 1/2^{53})$. In a 64-bit system, both numbers would be rounded to the binary string of 0111111111100000000000000000000000000000000000000000000000000000 or in a more readable form 2^{1023}. This would mean that the difference between the real values is $2^{1023} * (1 + 1/2^{53}) - 2^{1023} = 2^{1023}/2^{53} = 2^{970}$ even though the relative distance is only $100/2^{53}\%$. When a subtraction happens of $b - a$, the result will become 0 since the real values are rounded to the same floating point value. There is still an error present since the error was introduced before subtraction. The real value should be 2^{970} while the floating point value is 0. This means that the difference between the real values and the floating point value is 100% of the real value, and thus, no matter how you express the error, it becomes practically impossible to conclude anything useful regarding the error. For the floating point number 0, 1 ulp is $2^{1-1023-53}$ or 2^{-1075}. This means that in our method of expressing errors, the error bound is 2^{2045} ulps.

When encountering such a challenge, we considered two paths to take. Firstly, lemmas can be constructed which are valid in all circumstances. This could, however, lead to error bounds that do not seem useful, just like in the example. Also, we expected this path to be more difficult. Secondly, we could ignore these challenging cases and only claim an error bound when certain preconditions hold. Such a precondition can be on the input values, like saying everything needs to be positive, or on the program itself, like saying no subtraction can occur. This would result in a final product that can still be extremely useful in some instances, instead of slightly useful in all cases.

The use case that sparked the interest in this research was regarding probabilities in a Markov chain. Here every number is between 0 and 1 with only addition and multiplication. This is a very specific case, so the second approach is preferable for the difficulty of subtraction. We also think that this approach is more feasible to achieve so we choose this approach.

The problem with subtraction was not the only difficulty we encountered. For multiplication, we realized that we could prove tighter error bounds with tighter preconditions. Therefore, we need to decide how tight we make our proof. Again, we look at the Markov chains. Here the numbers are between 0 and 1. Next to that, we also need to set a precondition on the input accuracies.

In the end, we had the following preconditions. We defined the errors for the addition of positive floating point numbers. Plus 0 is considered to be positive since the floating point standard also defines minus 0 which we consider to be negative. Furthermore, we defined the errors for the multiplication of positive floating point numbers, where these floating point numbers represent real values that are greater or equal to 0 and smaller or equal to 1. For multiplications, the accuracies on the inputs need to be smaller or equal to $2^{number\_of\_fraction\_bits}$.

5.4 Application

We use the created lemmas to prove the accuracy property of an algorithm. This is the fourth step of the research. This application is for the accuracy of the iterative multiplication of a vector and a probability matrix. As was described in Section 3.2.3, matrices and vectors are already defined in Isabelle. This included many operations, but we still needed to define operations for when all elements in the matrices and vectors are floats. This extension is described in Section 5.4.1.

The biggest difficulty we encountered was not for the floating point calculations, but for the real calculations. We had a precondition that claims all real values are between 0 and 1, but when a vector is multiplied by a matrix, the output does not necessarily only contain values between 0 and 1. For a matrix that represents a Markov chain, however, this is the case because all columns in the matrix add up to 1. This leads to the fact that the sum of the input is the same as the sum of the output. Together with the facts that the sum of the input vector is 1 and the output vector cannot get negative numbers, we can conclude all values will be between 0 and 1.

When such an application of the lemmas is performed and time permitted, the next step could have looked at a less restricted use case and thus also be able to conclude an accuracy in other programs. This was not the case, so our lemmas are only applicable to a limited number of programs.

5.4.1 Extension of matrix formalization

For lists, there exist the functions "list_all". This function takes two arguments. The first is a predicate, and the second is a list. If this function is true, it would mean that the predicate is true for each element in the list. This function did not exist for vectors and matrices so we defined them for ourselves.

definition vec_all :: "$('a \Rightarrow \text{bool}) \Rightarrow\ 'a\ \text{vec} \Rightarrow \text{bool}$"
where "$\text{vec\_all}\ f\ a \equiv (\forall (i :: \text{nat}).\ i < \text{dim\_vec}\ a \rightarrow f\ (a\$i))$"

definition mat_all :: "$('a \Rightarrow \text{bool}) \Rightarrow\ 'a\ \text{mat} \Rightarrow \text{bool}$"
where "$\text{mat\_all}\ f\ a \equiv (\forall (i :: \text{nat})\ (j :: \text{nat}).\ (i < \text{dim\_row}\ a \wedge j < \text{dim\_col}\ a) \rightarrow f\ (a\$\$(i,\ j)))$"

We also thought it could be useful to make claims about each row or column of a matrix so we introduced the following definitions. In the end we never used the "mat_all_row" function, but an example where we can use "mat_all_col" is to check if the sum of a column is equal to 1.

definition mat_all_row :: "$('a\ \text{vec} \Rightarrow \text{bool}) \Rightarrow\ 'a\ \text{mat} \Rightarrow \text{bool}$"
where "$\text{mat\_all\_row}\ f\ a \equiv (\forall (i :: \text{nat}).\ (i < \text{dim\_row}\ a) \rightarrow f\ (\text{row}\ a\ i))$"

definition mat_all_col :: "$('a\ \text{vec} \Rightarrow \text{bool}) \Rightarrow\ 'a\ \text{mat} \Rightarrow \text{bool}$"
where "$\text{mat\_all\_col}\ f\ a \equiv (\forall (i :: \text{nat}).\ (i < \text{dim\_col}\ a) \rightarrow f\ (\text{col}\ a\ i))$"

Later we use these functions to make claims about the input and output vectors and matrices. An example is the following precondition where we claim that each element in ars and each element in brs is greater or equal to 0 and less or equal to 1.

"$\text{vec\_all}\ (\lambda r.\ 0 \leq r \wedge r \leq 1)\ ars \wedge \text{vec\_all}\ (\lambda r.\ 0 \leq r \wedge r \leq 1)\ brs$"

Furthermore, "list_all2" is also defined in Isabelle. This function is used to describe a relation between 2 lists. Again, a predicate is given as an argument. The function "list_all2" is true when this predicate holds when any 2 elements of the lists in the same position are provided. Just like "list_all", this is not defined for matrices and vectors so we defined it ourselves.

definition vec_all2 :: "$('a \Rightarrow\ 'b \Rightarrow \text{bool}) \Rightarrow\ 'a\ \text{vec} \Rightarrow\ 'b\ \text{vec} \Rightarrow \text{bool}$"
where "$\text{vec\_all2}\ f\ a\ b \equiv (\text{dim\_vec}\ a = \text{dim\_vec}\ b) \wedge (\forall (i :: \text{nat}).\ i < \text{dim\_vec}\ a \rightarrow f\ (a\$i)\ (b\$i))$"

definition mat_all2 :: "$('a \Rightarrow\ 'b \Rightarrow \text{bool}) \Rightarrow\ 'a\ \text{mat} \Rightarrow\ 'b\ \text{mat} \Rightarrow \text{bool}$"
where "$\text{mat\_all2}\ f\ a\ b \equiv (\text{dim\_row}\ a = \text{dim\_row}\ b) \wedge (\text{dim\_col}\ a = \text{dim\_col}\ b) \wedge (\forall (i :: \text{nat})\ (j :: \text{nat}).\ (i < \text{dim\_row}\ a \wedge j < \text{dim\_col}\ a) \rightarrow f\ (a\$\$(i,j))\ (b\$\$(i,j)))$"

An example of where we use this is the following.

"$\text{vec\_all2}\ (\lambda f\ r.\ \text{ulp\_accuracy}\ r\ (f :: ('e,\ 'f)\ \text{float})\ 0)\ (bfs :: ('e,\ 'f)\ \text{float vec})\ brs$"

Here we have 2 vectors. We have bfs which is a vector of floating point numbers, and we have brs which is a vector of real numbers. The relation is described with a lambda function that claims the "ulp_accuracy" is 0 between the 2 inputs. The "ulp_accuracy" function is defined in Section 5.1. For now, it is relevant to know that "$\lambda f\ r.\ \text{ulp\_accuracy}\ r\ (f :: ('e,\ 'f)\ \text{float})\ 0$" is true when floating point number f and real number r have the exact same value. This means that the vector of floating point values has the exact same values as the vector of real numbers.

Next to these functions, we also define operations on vectors and matrices. To be more specific, we define dot products. The formalization we use has a definition for a dot product for reals, but not for floats. This is why need to define it for floats.

```
definition float_vec_mul :: "('e, 'f) float vec ⇒ ('e, 'f) float vec ⇒ ('e, 'f) float"
    where "float_vec_mul as bs = fold
                    (λi s. fmul_add To_nearest (as$nat i) (bs$nat i) s)
                    [0..int (dim_vec as) − 1] 0"

definition float_mult_mat_vec :: "('e, 'f) float mat ⇒ ('e, 'f) float mat ⇒
    ('e, 'f) float vec"
    where "float_mult_mat_vec A v ≡ vec (dim_row A)
                    (λi. float_vec_mul (row A i) v)"
```

We also define a new dot product for reals following the same structure to make the proofs easier. We have proven that our definition is equivalent to the other definition that was given in the matrix formalization.

```
definition real_vec_mul :: "real vec ⇒ real vec ⇒ real"
    where "real_vec_mul as bs = fold (λi s. (as$nat i) ∗ (bs$nat i) + s)
    [0..int (dim_vec as) − 1] 0"

lemma real_vec_mul_scalar_product :
assumes "dim_vec as = dim_vec bs"
  shows "real_vec_mul as bs = as • bs"
```

5.5 Conclusion of approach

When these four steps are taken, we can answer research questions as described in Section 4.1. We will answer these questions in Chapter 9, but we discuss here if this approach is enough to answer the questions.

To recap, the first question is: "What are the necessary lemmas to prove an error bound for algorithms measured in ulps?". When the fourth step of our approach is completed, we have proven a bound for an algorithm. From this, we can conclude which lemmas are necessary.

The second question is: "Is it possible to construct a proof for the error bound measured in ulps of a program that calculates a probability distribution over a Markov chain using these lemmas?". Again, when the fourth step is completed, we know that this is possible because in the fourth step we construct this proof.

Lastly, the third question is: "How tight are the error bounds constructed using these lemmas?". After completing the four steps in this approach we do not have an answer to this question. For that, we need another step to prove the tightness of the lemmas which we use to construct error bounds. We did not manage to perform this step but in Section 9.3 we give a reasoned guess for the answer of this question.

Chapter 6

Results

In this chapter we will explain some of the important lemmas we have proven in the Isablle/HOL system. We do this by explaining what the preconditions mean and why they are necessary. In Section 9.1.1, we will give a list of the most important proofs. In Chapter 7, we explain proofs of some lemmas.

6.1 Lemmas

The lemmas that we have proven for this research can be split up into a couple of different types. First, there are lemmas to prove something regarding the rounding step. The conclusion of these lemmas is the following as we already showed in Section 5.2:

assumes "$|a| <$ threshold $TYPE(('e,\ 'f)$ float)"
  and "$a_r = (($round $To\_nearest\ a) :: ('e,\ 'f)$ float)"
  and "$1 < LENGTH('e)$"
  and "$1 < LENGTH('f)$"
 shows "ulp_accuracy $a\ a\_r$ 0.5"

Lemma 6.1: Rounding creates an accuracy of 0.5 ulp

This lemma proves that when a real value a is rounded to the floating point value a_r, the distance between a and a_r is at most 0.5 ulp. The assumptions or preconditions contain a few functions. The first assumption claims that the real number a is smaller than a certain number. This threshold is formally defined in Section 3.2.2 and it is the point where a number will be rounded towards plus or minus infinity. In the next precondition, we define a_r as the floating point value to which a will be rounded. We also define the rounding mode to be $To\_nearest$. The last 2 assumptions make a claim about the length of $'e$ and $'f$. This simply means that the floating point number uses at least 2 bits for the exponent and 2 bits for the fraction. This lemma shows that when a real number is rounded, the real value will be within 0.5 ulp of the rounded floating point value.

We discussed in Section 5.3 that we also need lemmas regarding how errors propagate through basic mathematical operations. We constructed proofs for addition and multiplication inside a context in Isabelle/HOL. This means that we can use a group of assumptions for each lemma inside of that context. The main goal of using this is that proofs will be cleaner and easier to understand. We chose the following assumptions.

```
context
    fixes $a\_float$ $b\_float$ $c\_float$ :: "('e, 'f)float"
     and $a\_real$ $b\_real$ $c\_real$ :: "real"
     and $a\_accuracy$ $b\_accuracy$ $c\_accuracy$ :: "real"
    assumes $a\_rel$ : "ulp_accuracy $a\_real$ $a\_float$ $a\_accuracy$"
        and $b\_rel$ : "ulp_accuracy $b\_real$ $b\_float$ $b\_accuracy$"
        and $c\_rel$ : "ulp_accuracy $c\_real$ $c\_float$ $c\_accuracy$"
        and $len\_e$ : "$1 < LENGTH('e)$"
        and $len\_f$ : "$1 < LENGTH('f)$"
        and $sign\_a$ : "sign $a\_float = 0$"
        and $sign\_b$ : "sign $b\_float = 0$"
        and $sign\_c$ : "sign $c\_float = 0$"
```

We fix 3 floating point numbers. Each one of them has a corresponding real number and an accuracy to represent the error bound between the real number and the floating point number. In our assumptions, we use the function "ulp_accuracy" to display how big this error bound is. For example, the distance between a_float and a_real is less or equal to $a\_accuracy$ ulps. Furthermore, we have assumptions on the length of $'e$ and $'f$. This assumption is needed to know how big the rounding error is therefore we use this assumption here again. Lastly, we have assumptions on the signs of the floating point numbers. This means that all floating point numbers are greater or equal to 0. As was discussed in Section 5.3, we chose to only construct proofs for addition and multiplications using real numbers that are between 0 and 1.

Using these assumptions we can prove how errors propagate. Here we show that when you add 2 numbers with an accuracy of $a\_accuracy$ and $b\_accuracy$, then the result has an accuracy that is better or equal to $a\_accuracy + b\_accuracy$.

```
assumes "|valof $a\_float$ + valof $b\_float$| < threshold $TYPE(('e, 'f)$ float)"
  shows "|(valof $a\_float$ + valof $b\_float$) − ($a\_real + b\_real$)| ≤
          ($a\_accuracy + b\_accuracy$) ∗ ulp (fadd $To\_nearest$ $a\_float$ $b\_float$)"
```

LEMMA 6.2: Addition adds the input accuracies

We do need to add one more assumption. This assumption includes the "threshold" function again. This assumption is needed to know that "fadd $To\_nearest$ a_float b_float" is finite. An ulp represents the value of the least significant bit, but if a value is infinite, this is not applicable. The formal definition will still return a value for an ulp of an infinite floating point number so we need to check if the floating point number is finite before using the "ulp" function in "ulp (fadd $To\_nearest$ a_float b_float)".

Next to addition, we have also proven a lemma on error propagation through multiplication.

assumes "$|\text{valof } a\_float * \text{valof } b\_float| <$ threshold $TYPE(('e,\ 'f)$ float)"
and "$a\_accuracy \leq (2 :: \text{real})\hat{}LENGTH('f)$"
and "$b\_accuracy \leq (2 :: \text{real})\hat{}LENGTH('f)$"
and "$0 \leq a\_real$" and "$a\_real \leq 1$"
and "$0 \leq b\_real$" and "$b\_real \leq 1$"
shows "$|(\text{valof } a\_float * \text{valof } b\_float) - (a\_real * b\_real)| \leq$
$(2 * a\_accuracy + 2 * b\_accuracy) * \text{ulp } (\text{fmul } To\_nearest\ a\_float\ b\_float)$"

Lemma 6.3: Multiplication doubles and adds the input accuracies

Again we use an assumption to make sure the resulting floating point number does not pass the threshold to become infinite, but we also introduce new assumptions.

For normal numbers, we can use the accuracy in ulps to get an accuracy expressed as a factor. Or expressed in equations: "$\text{valof } a\_float = (1 + x) * a\_real$" where "$|x| <= a\_accuracy/2^{LENGTH('f)}$". This means that the distance between the resulting floating point number and the resulting real is the following:

$$\begin{aligned}&|\text{valof } a\_float * \text{valof } b\_float - a\_real * b\_real| =\\&|\text{valof } a\_float * \text{valof } b\_float - (1 + x_1) * \text{valof } a\_float * (1 + x_2) * \text{valof } b\_float| =\\&|\text{valof } a\_float * \text{valof } b\_float - (1 + x_1 + x_2 + x_1 * x_2) * \text{valof } a\_float * \text{valof } b\_float|\end{aligned}$$

When we put a limit on $a\_accuracy$ and $b\_accuracy$, we put a limit on x_1 and x_2. For this reason, we put a limit on "$x_1 * x_2$" which means we can simplify the equation. We chose a limit of "$(2 :: \text{real})\hat{}LENGTH('f)$" since for normal numbers this would mean that the error bound should not exceed 100% which is a nice round number. If we chose a lower limit, "$x_1 * x_2$" would be smaller and thus we could prove a tighter bound.

Next, we have assumptions that limit a_real and b_real to be between 0 and 1. For subnormal numbers, we use the accuracy in ulps to get an accuracy expressed as an absolute distance. Or expressed in equations: "$\text{valof } a\_float = a\_real + x$" where "$|x| <= a\_accuracy * \text{ulp } a\_float$". When a_float is subnormal and b_float is normal, this means that the distance between the resulting floating point number and the resulting real is the following:

$$\begin{aligned}&|\text{valof } a\_float * \text{valof } b\_float - a\_real * b\_real| =\\&|\text{valof } a\_float * \text{valof } b\_float - (\text{valof } a\_float + x_1) * b\_real| =\\&|\text{valof } a\_float * \text{valof } b\_float - \text{valof } a\_float * b\_real + x_1 * b\_real|\end{aligned}$$

Using the assumption that a_real and b_real are between 0 and 1, we know that "$|x_1 * b\_real| \leq |x_1|$". Using this, we can simplify the equation

$$\begin{aligned}&|\text{valof } a\_float * \text{valof } b\_float - \text{valof } a\_float * b\_real + x_1 * b\_real| \leq\\&|\text{valof } a\_float * \text{valof } b\_float - \text{valof } a\_float * b\_real| + |x_1| =\\&|\text{valof } a\_float * \text{valof } b\_float - \text{valof } a\_float * (1 + x_2) * \text{valof } b\_float| + |x_1| =\\&|x_2 * \text{valof } a\_float * \text{valof } b\_float| + |x_1|\end{aligned}$$

When a computer calculates an addition or a multiplication, the result will be rounded so we need to combine this lemma with the rounding lemma. Therefore, the error bound is increased by 0.5 ulp. For addition this results in the following lemma.

assumes "$|$valof a_float + valof $b\_float| <$ threshold $TYPE(('e, 'f)$ float)"
shows "ulp_accuracy $(a\_real + b\_real)$ (fadd $To\_nearest\ a\_float\ b\_float$)
$(a\_accuracy + b\_accuracy + 0.5)$"

Lemma 6.4: Accuracy of the fadd operator

And for multiplication it looks like this.

assumes "$|$valof $a\_float *$ valof $b\_float| <$ threshold $TYPE(('e, 'f)$ float)"
and "$a\_accuracy \leq (2 ::$ real$)\hat{}LENGTH('f)$"
and "$b\_accuracy \leq (2 ::$ real$)\hat{}LENGTH('f)$"
and "$0 \leq a\_real$" and "$a\_real \leq 1$"
and "$0 \leq b\_real$" and "$b\_real \leq 1$"
shows "ulp_accuracy $(a\_real * b\_real)$ (fmul $To\_nearest\ a\_floatb\_float$)
$(2 * a\_accuracy + 2 * b\_accuracy + 0.5)$"

Lemma 6.5: Accuracy of the fmul operator

Here no new assumptions are introduced. All the preconditions can also be seen in the previous lemma.

Furthermore, we can also prove a lemma on error propagation through the "fmul_add" operator. This operator takes three arguments a, b, and c. It will multiply the first two and add the third so the result is "$a * b + c$". A benefit of using this operator instead of the "fmul" and "fadd" operators, is that the result is only rounded once.

We can use our previous lemmas to prove the following:

assumes and "$|$valof $a\_float *$ valof $b\_float +$ valof $c\_float|$
$<$ threshold $TYPE(('e, 'f)$ float)"
and "$a\_accuracy \leq (2 ::$ real$)\hat{}LENGTH('f)$"
and "$b\_accuracy \leq (2 ::$ real$)\hat{}LENGTH('f)$"
and "$0 \leq a\_real$"
and "$a\_real \leq 1$"
and "$0 \leq b\_real$"
and "$b\_real \leq 1$"
shows "ulp_accuracy $(a\_real * b\_real + c\_real)$
(fmul_add $To\_nearest\ a\_float\ b\_float\ c\_float$)
$(2 * a\_accuracy + 2 * b\_accuracy + c\_accuracy + 0.5)$"

Lemma 6.6: Accuracy of the fmul_add operator

6.2 Application

Using the previous lemma we can prove properties about the error bound for iterative matrix multiplication. There are 12 preconditions so we will explain them one by one. The resulting lemma is the following.

```
assumes a_rel : "mat_all2 (λf r. ulp_accuracy r (f :: ('e, 'f) float) 0)
          (afs :: ('e, 'f) float mat) ars"
    and b_rel : "vec_all2 (λf r. ulp_accuracy r (f :: ('e, 'f) float) 0)
          (bfs :: ('e, 'f) float vec) brs"
    and signs : "mat_all (λf. sign f = 0) afs∧
           vec_all (f. sign f = 0) bfs"
    and probs : "mat_all (λr.0 ≤ r ∧ r ≤ 1) ars ∧  vec_all (λr.0 ≤ r ∧ r ≤ 1) brs"
    and len_e : "1 < LENGTH('e)"
    and len_f : "1 < LENGTH('f)"
    and lim : "(Σi = 1..k.(2 ∗ dim_vec bfs)ˆi)/4 ≤ (2 :: real)ˆLENGTH('f)"
    and fin : "vec_all (is_finite)(((float_mult_mat_vec afs)ˆˆk) bfs)"
    and dim_1 : "dim_col afs = dim_vec bfs"
    and dim_2 : "dim_col afs = dim_row afs"
    and sum_1 : "mat_all_col (λv :: real vec. sum (($) v)
          {0 :: nat.. < dim_vec v} = (1 :: real))(ars :: real mat)"
    and sum_2 : "sum (($) (brs :: real vec)) 0 :: nat.. < dim_vec brs = (1 :: real)"
  shows "vec_all2 (λf r.ulp_accuracy r f ((Σi = 1..k.(2 ∗ dim_vec bfs)ˆi)/4))
          (((float_mult_mat_vec afs)ˆˆk) bfs) (((mult_mat_vec ars)ˆˆk) brs) ∧
          vec_all (λf.sign f = 0) (((float_mult_mat_vec afs)ˆˆk) bfs)"
```

Lemma 6.7: Accuracy of a dot product between a vector and a matrix

For *a_rel* and *b_rel*, we use the previously defined "mat_all2" and "vec_all2" functions. As was explained in Section 5.4.1, these functions describe a relation between 2 matrices or 2 vertices. In this lemma the relation tells us something about the "ulp_accuracy", and more specifically, it tells us that there are no rounding errors in the beginning. This means that each float in *afs* or *bfs* has the exact same value as the real number in *ars* or *brs* in the same place. Next to that, the definitions the "mat_all2" and "vec_all2" functions are structured to also imply that the matrices and vertices have the same dimensions. This needs to be true for the dot product to be a valid operation.

The next precondition, *signs*, uses a "mat_all" and "vec_all". In our precondition, the first argument checks that the floating point number has a sign bit which is 0, and thus the floating point value is bigger or equal to 0. This is used to show that when the matrix is multiplied by the vector, the output vector will also have numbers that are bigger or equal to 0.

Next, we use "mat_all" and "vec_all" again but now to make a claim about the real numbers in the matrix *ars* and the vector *brs*. We know probabilities are supposed to be between 0% and 100%. This is represented as values between 0 and 1. This is why the fourth precondition, *probs*, assumes that all the starting real values are between 0 and 1.

Assumptions *len_e* and *len_f* claim that the lengths of $'e$ and $'f$ are greater than 1. This simply means that both the exponent part and the fraction part of the floating point number should be longer than 1 bit. Most computers will use floating point numbers which have in total 32 or 64 bits so in practice, these preconditions will be met.

The seventh assumption, *lim*, needs more explanation. With this assumption, we put a limit on the size of the error bound in the conclusion. What this precondition says, is that if the error bound is bad enough, then the postcondition might not hold and thus the error might be even worse. The limit of this error bound for this proof is "$(2 :: \text{real})\hat{}\,LENGTH('f)$", which for normal numbers would be a maximum of 100%. In practise an error bound which is this big does not appear to be useful anyway, so we do not think this precondition limits the usefulness of this proof.

The fin assumption claims that the output from the floating point calculation is still finite. If the output would contain an infinite number, an error bound would make little sense for that element. This precondition might seem unnecessary since probabilities would always be between 0 and 1, but sadly the real numbers staying between 0 and 1 does not imply the floating point numbers stay between 0 and 1 as well. This example will be given in Section 9.3.4.

If you multiply a matrix by a vector, the matrix should have the same amount of columns as the length of the vector. This is why "dim_col afs = dim_vec bfs" should be true for the multiplication to be valid. The result would have a length equal to the number of rows of the matrix which is then again used as an input. This is why the matrix should be a square matrix and "dim_col afs = dim_row afs" should also be true.

Lastly, in this application, we wanted to use a stochastic matrix which means each column of the matrix sums up to 1. This leads the output vector to have the same sum as the input vector which we also want to sum up to 1. This is written down in sum_1 and sum_2. The "mat_all_col" function works just like the "mat_all" function but applies the function to each column instead of each element.

As can be seen, these last 4 assumptions are relevant since we calculate a dot product between a stochastic matrix and vector and this should be a valid operation. All other assumptions are related to the assumptions for the proof of the error propagation for the "fmul_add" operator.

These assumptions are enough to prove an error bound for the iterative matrix multiplication. Next to that, it also proves something about the sign of the output that looks a lot like the third precondition. This is needed for the induction proof, but proving that the signs stay 0 could also be proven in a different lemma. In Chapter 7 we will go into the proofs of how we show these conclusions.

Chapter 7

Isabelle/HOL proofs

We explain some proofs in more detail. We show the proof for the error introduction by a single rounding operation, the proof for error propagation through addition, and the proof for the error propagation through a dot product between vectors. These proofs consist of multiple lemmas, but not all helper lemmas will be explained as those can be seen in the repository [4].

7.1 Error of rounding operation

In this section we show pieces of the proof and give an explanation afterward each time.

```
lemma closest_means_ulp_0_5 :
  assumes "ulp_accuracy $a$ $b$ 0.5" and "$LENGTH('f) > 1$"
  and "($\forall b.\ b \in \{a.$ is_finite $a\} \to |$valof $c - a| \leq$
    $|$valof $b - a|$)" and "is_finite $c$"
  shows "ulp_accuracy $a$ $c$ 0.5"
```

The first part of the proof lays out the preconditions. The most important thing to explain is that we define 3 variables here. a is a real number, b is a float that has an ulp accuracy of 0.5 in relation to a and c is a float that is the closest floating point number to a and therefore also has a distance to a which is smaller or equal to the distance between a and b.

```
apply (cases "exponent $b \leq$ exponent $c \lor$ exponent $b = 1$")
subgoal proof -
from ulp_equivelance have ulp_same : "$\neg$exponent $b \leq$ exponent $c$
  $\Rightarrow$ exponent $b = 1 \Rightarrow$ ulp $b$ = ulp $c$"
  by (metis ...)
moreover have "exponent $b \leq$ exponent $c \to$ ulp $b \leq$ ulp $c$"
  by (auto simp add:ulp_equivelance divide_right_mono)
ultimately have "exponent $b \leq$ exponent $c$ $\lor$ exponent $b = 1 \Rightarrow$ ulp $b \leq$ ulp $c$"
  by argo
then show "exponent $b \leq$ exponent $c \lor$ exponent $b = 1$
  $\Rightarrow$ ulp_accuracy $a$ $c$ (5/10)"
   using assms apply(simp add:ulp_accuracy_def) by force
qed
```

We start the proof by splitting it up in 2 cases. The first case is when the exponent of c is bigger or when the exponent of b is 1. In practice, this means that the first case is when the ulp of c is bigger or equal to the ulp of b, because a higher exponent implies a greater or equal ulp and when the exponent is either 0 or 1, then the ulps are equal. From this, we can conclude that if 1 ulp covers a greater or equal distance, and the distance between a and c is smaller or equal, then the distance between a and c is also fewer or equal ulps to the distance between a and b. This leads us to conclude that the "ulp_accuracy" is also 0.5.

The second case will have a completely different strategy. We will prove the values for the sign, exponent, and fraction parts such that we can determine the distance between b and c. Based on this we can prove an accuracy measured in ulps.

```
subgoal proof -
{
  assume a : "¬exponent b ≤ exponent $c$"
     and b : "exponent $b \neq 1$"
  from a b have exp_not_0 : "exponent $b \neq 0$"
    by simp
  from sign_valof_diff have "sign $b \neq$ sign $c \Rightarrow$ |valof $c - a$| ≥
    |valof $b$| + |valof $c$| − |valof $b - a$|" by (smt ...)
  with assms exp_not_0 ulp_accuracy_def have "sign $b \neq$ sign $c \Rightarrow$
    |valof $c - a$| > |valof $b - a$|"
    apply(simp add: assms ulp_accuracy_def ulp_is_smaller)
    by (smt ...)
  with assms ulp_accuracy_def have "sign $b =$ sign $c$"
    by (smt ...)
} note sign_equality = this
```

To prove the equality of signs of b and c, we start by showing that the assumptions of this case implies that b is a normal number. If the signs were different, one float would be positive and the other would be negative and therefore, the distance between c and a is at least the distance between c and b minus the distance between a and b. The value of b is at least the smallest representable normal number. We therefore know that the distance between c and a is at least the smallest representable normal number minus half an ulp of b. From "ulp_is_smaller" we know that the smallest normal number is greater than an ulp, so the distance between c and a is greater than half an ulp of b. This would mean c is not closer to a compared to b, and that is different from our assumptions. From this, we conclude that it is impossible for b and c to have different signs.

```
{
  assume a : " ≠ exponent b ≤ exponent $c$"
     and b : "exponent $b \neq 1$"
  assume to_disprove : "exponent $c <$ exponent $b - 1$"
```

In the part to determine the exponent we again make the assumptions for this case. Furthermore, we make an assumption called "to_disprove". We want to prove this part by contradiction. Including the "to_disprove" assumption means that we do not have to include it in every line of our proof.

```
{
  from a b have b_at_least_2 : "(exponent $b$) $-\ 2 + 1 =$ (exponent $b$) $-\ 1$"
    by auto
  from abs_valof_max have "|valof $c$| $<$ ($2 * 2$^(exponent $c$)/2^bias
    $TYPE(('e,\ 'f)$ float))"
    by blast
  moreover from to_disprove have "exponent $c \leq$ exponent $b - 2$"
    by linarith
  ultimately have "|valof $c$| $\leq$ ($2 * 2$^(exponent $b - 2$)/2^bias
    $TYPE(('e,\ 'f)$ float))"
    by (smt ...)
  with b_at_least_2 have "|valof $c$| $\leq$ (2^((exponent $b$) $-\ 1$)/2^
    bias $TYPE(('e,\ 'f)$ float))"
    by (metis ...)
} note val_c_simplified = this
```

After that we rewrite "|valof c|" in such a form that it will be easier to work with later.

```
{
  from a have "|valof $b$| = (2^(exponent $b$)/2^bias $TYPE(('e,\ 'f)$ float))$*$
    (2^$LENGTH('f)$ + real (fraction $b$))/2^$LENGTH('f)$"
    by (simp add:abs_valof_eq2)
  with float_frac_ge have "|valof $b$| $\geq$ (2^(exponent $b$)/2^bias
    $TYPE(('e,\ 'f)$ float)) $*$ (2^$LENGTH('f)$)/2^$LENGTH('f)$"
    by (smt ...)
  with a have "|valof $b$| $\geq$ (2^(exponent $b - 1 + 1$)/2^bias $TYPE(('e,\ 'f)$
    float))"
    by fastforce
  then have val_b_simplified : "|valof $b$| $\geq$ ($2 * 2$^(exponent $b - 1$)/2^bias
    $TYPE(('e,\ 'f)$ float))"
    by (metis ...)
} note val_b_simplified = this
```

We do the same for "$|$valof $b|$".

```
        {
          have step_1 : "$((2\hat{\ }LENGTH('f)) - 1) * (2\hat{\ }$(exponent $b - 1)/2\hat{\ }$(bias
            $TYPE(('e,\ 'f)$ float) $+ LENGTH('f))) > (2\hat{\ }$(exponent
            $b - 1)/2\hat{\ }$(bias $TYPE(('e,\ 'f)$ float) $+ LENGTH('f)))$"
            apply(simp add:div_less divide_less_eq frac_less)
            using assms power_2_at_least_4 by fastforce
          from assms ulp_accuracy_def bias_def ulp_equivelance a
            half_times_power have "$|$valof $b - a| \leq 0.5 * (2\hat{\ }$max
            (exponent $b$) $1/2\hat{\ }$(bias $TYPE(('e,\ 'f)$ float) $+ LENGTH('f)))$"
            by metis
          with a half_times_power have val_b_min_a : "$|$valof $b - a| \leq$
            $(2\hat{\ }$(exponent $b - 1)/2\hat{\ }$(bias $TYPE(('e,\ 'f)$ float) $+ LENGTH('f)))$"
            by auto
          with to_disprove a b val_b_simplified val_c_simplified have
            "$|$valof $c - a| \geq (2\hat{\ }LENGTH('f)) * (2\hat{\ }$(exponent $b - 1)/2\hat{\ }$(bias
            $TYPE(('e,\ 'f)$ float) $+ LENGTH('f))) - (2\hat{\ }$(exponent
            $b - 1)/2\hat{\ }$(bias $TYPE(('e,\ 'f)$ float) $+ LENGTH('f)))$"
            apply (simp add: power_add) by argo
          with val_b_min_a step_1 have "$|$valof $c - a| > |$valof $b - a|$"
            by argo
        } note c_min_a_bigger = this
```

Combining everything leads us to showing "$|$valof $c - a| > |$valof $b - a|$", but from our assumptions at the start of the lemma, we know this is false. From this we can conclude that the assumption called "to_disprove" is false and that the exponent of c is exactly the exponent of b minus 1.

```
      with assms have "exponent $c =$ exponent $b - 1$"
        by (auto simp add:ulp_accuracy_def)
    } note exp_c_exp_b_min_1 = this
    with assms have exp_c_exp_b_min_1 : "$\neg$exponent $b \leq$ exponent $c \Rightarrow$
      exponent $c =$ exponent $b - 1$"
      by fastforce
    with ulp_equivelance have ulp_b_2_ulp_c : "$\neg$exponent $b \leq$ exponent $c \Rightarrow$
      exponent $b \neq 1 \Rightarrow$ ulp $b = 2 *$ ulp $c$"
      by (simp add:ulp_equivelance pow_f_min_1)
    from exp_c_exp_b_min_1 have exp_c_not_0 : "$\neg$exponent $b \leq$ exponent $c \Rightarrow$
      exponent $b \neq 1 \Rightarrow$ exponent $c \neq 0$"
      by linarith
    then have exp_c_bigger_equal_1 : "$\neg$exponent $b \leq$ exponent $c \Rightarrow$ exponent $b \neq$
      $1 \Rightarrow$ max (exponent $c$) $1 =$ exponent $c$"
      by fastforce
```

When we know exactly what the exponent of c is, we can also know what the ulp is.

Furthermore, we also know the exponent is not 0, which we use in the proof later.

```
{
  assume a : "¬exponent $b$ ≤ exponent $c$"
     and b : "exponent $b \neq 1$"

  from a b have "||valof $b$| − |valof $c$|| = |valof $b$ − valof $c$|"
    apply (cases "sign $b = 0$")
    by (simp_all add: sign_pos sign_equality sign_cases
    valof_nonpos valof_nonneg)
  moreover from a abs_valof_ge_exp_ge
    have val_b_g_val_c : "|valof $b$| > |valof $c$|"
    using linorder_not_less by blast
  ultimately have val_b_min_val_c : "|valof $b$ − valof $c$| = |valof $b$| − |valof $c$|"
    by argo
```

Next we prove the fractions of both b and c. We start this by again making the assumptions for this part. We use what we have proven about the sings to determine the difference between the value of b and the value of c.

```
{
  assume to_disprove : "fraction $c \leq 2\hat{}LENGTH('f) - 2$"
  then have "(2^(exponent $c$)/2^bias $TYPE(('e,\ 'f)$ float))∗
    $(2\hat{}LENGTH('f)$ + real (fraction $c$))/2^$LENGTH('f)$ ≤
    (2^(exponent $c$)/2^bias $TYPE(('e,\ 'f)$float))∗
    $(2\hat{}LENGTH('f) + 2\hat{}LENGTH('f) - 2)/2\hat{}LENGTH('f)$"
    using mult_le_cancel_left_pos[where ...]
    by (smt ...)
  then have bad_val_c_expanded : "|valof $c$| ≤ (2^(exponent $c$)/2^bias
    $TYPE(('e,\ 'f)$ float)) ∗ $(2\hat{}LENGTH('f) + 2\hat{}LENGTH('f) - 2)$
    /2^$LENGTH('f)$"
    using a b exp_c_exp_b_min_1 by(simp add:abs_valof_eq2)
  moreover from a b exp_c_exp_b_min_1 have "|valof $b$| ≥ (2^(
    exponent $c$)/2^bias $TYPE(('e,\ 'f)$ float)) ∗ (2^$LENGTH('f)$+
    2^$LENGTH('f)$)/2^$LENGTH('f)$"
    using mult_left_mono[where ...]
    by (simp add:abs_valof_eq2 pow_f_min_1 power_commutes)
  ultimately have "|valof $b$ − valof $c$| ≥ (2^(exponent $c$)/2^bias $TYPE$
    $(('e,\ 'f)$ float)) ∗ 2/2^$LENGTH('f)$"
    using val_b_min_val_c by argo
  then have b_min_c : "|valof $b$ − valof $c$| ≥ ulp $c$ ∗ 2"
    using a b exp_c_bigger_equal_1
    by (simp add: bias_def ulp_equivelance ulp_divisor_rewrite)
```

To prove the fraction of c, we make an assumption that we want to disprove. The assumption that we make leads the distance between b and c being at least 2 ulps of c.

```
from a b assms ulp_b_2_ulp_c exp_c_exp_b_min_1 have
  "|valof $b - a$| $\leq$ ulp $c$"
  by(simp add:ulp_accuracy_def)
moreover with assms have "|valof $c - a$| $\leq$ ulp $c$"
  using ulp_accuracy_def dual_order.trans by blast
ultimately have dists : "|valof $b - a$| $=$ ulp $c \wedge$ |valof $c - a$| $=$ ulp $c$"
  using b_min_c by argo
moreover from a b exp_c_not_0 ulp_is_smaller have "|valof $c$| $>$ ulp $c$"
  by force
ultimately have "$|a|$ $=$ |valof $c$| $+$ ulp $c$"
  using val_b_g_val_c by argo
```

From this, we conclude that the distance between b and c is exactly 2 ulps of c and that a is exactly in the middle.

```
  with ulp_increase_abs_diff assms have "$|a|$ $=$ |valof (ulp_increase $c$)|"
    by fastforce
  with valof_minus have "$|a -$ valof (ulp_increase $c$)$| = 0 \vee$
    $|a -$ valof ($-$ulp_increase $c$)$| = 0$"
    by auto
  moreover have "is_finite (ulp_increase $c$)"
    using ulp_increase_exp assms exp_c_exp_b_min_1
    ulp_accuracy_def eq_exp_eq_finite by (smt ...)
  moreover then have "is_finite ($-$ulp_increase $c$)"
    by fastforce
  ultimately have "$\exists x.|a -$ valof $x| = 0 \wedge$ is_finite $x$"
    by blast
  then have "$x.|a -$ valof $x| < |a -$ valof $c| \wedge$ is_finite $x$"
    using dists ulp_positive by (metis ...)
  with assms(3) have "False" by fastforce
} note to_disprove_false2 = this
```

When we know that the distance between b and c is 2 ulps and that a is in the middle, then we also know that there is a float which has the exact same value as a. Therefore, c is not the closest value anymore and we conclude that the assumption "to_disprove" is indeed false.

```
then have "fraction $c \geq 2\hat{}LENGTH('f) - 1$"
  apply (simp add: nat_le_real_less of_nat_diff) by argo
with float_frac_le have "real (fraction $c$) $= 2\hat{}LENGTH('f) - 1$"
  by (metis ...)
then have "$2\hat{}LENGTH('f) +$ real (fraction $c$) $= 2\hat{}$
  $LENGTH('f) + 2\hat{}LENGTH('f) -$ Suc 0)"
  apply simp by (metis ...)
then have val_c_eq : "|valof $c$| $=$ ($2\hat{}$(exponent $c$)$/2\hat{}$bias $TYPE($
  $('e, 'f)$ float)) $*$ ($2\hat{}LENGTH('f) + 2\hat{}LENGTH('f) -$
  $1)/2\hat{}LENGTH('f)$"
  using a b exp_c_exp_b_min_1 by (simp add:abs_valof_eq2)
```

Using the disproven property, we can prove the value for the fraction of c and thus we can prove the absolute value of c.

```
from assms have "$|\text{valof } c - a| \leq 0.5 * \text{ulp } b$"
   by (smt ...)
with ulp_b_2_ulp_c a b have "$|a| \leq |\text{valof } c| + \text{ulp } c$"
   by linarith
with val_c_eq ulp_equivelance have "$|a| \leq (2\hat{}(\text{exponent } c)/2\hat{}\text{bias}$
   $TYPE(('e,\ 'f)\ \text{float})) * (2\hat{}LENGTH('f) + 2\hat{}LENGTH('f)-$
   $1)/2\hat{}LENGTH('f) + 2\hat{}\text{max (exponent } c)\ 1/$
   $2\hat{}(2\hat{}(LENGTH('e) - 1) - 1 + LENGTH('f))$"
   by metis
with a b exp_c_exp_b_min_1 have "$|a| \leq (2\hat{}(\text{exponent } c)/2\hat{}\text{bias}$
   $TYPE(('e,'f)\ \text{float})) * (2\hat{}LENGTH('f) + 2\hat{}LENGTH('f)-$
   $1)/2\hat{}LENGTH('f) + 2\hat{}\text{exponent } c/2\hat{}(2\hat{}$
   $(LENGTH('e) - 1) - 1 + LENGTH('f))$"
   by force
with a b exp_c_exp_b_min_1 ulp_divisor_rewrite bias_def have
   "$|a| \leq (2\hat{}(\text{exponent } c)/2\hat{}\text{bias } TYPE(('e,\ 'f)\ \text{float}))*$
   $(2\hat{}LENGTH('f) + 2\hat{}LENGTH('f) - 1)/2\hat{}LENGTH('f)+$
   $2\hat{}\text{exponent } c/2\hat{}\text{bias } TYPE(('e,\ 'f)\ \text{float})/2\hat{}LENGTH('f)$"
   by (metis ...)
then have "$|a| \leq ((2\hat{}(\text{exponent } c)/2\hat{}\text{bias } TYPE(('e,\ 'f)\ \text{float}))*$
   $(2\hat{}LENGTH('f) + 2\hat{}LENGTH('f) - 1) + 2\hat{}$
   $\text{exponent } c/2\hat{}\text{bias } TYPE(('e,\ 'f)\ \text{float}))/2\hat{}LENGTH('f)$"
   by argo
then have "$|a| \leq (2\hat{}(\text{exponent } c)/2\hat{}\text{bias } TYPE(('e,\ 'f)\ \text{float}))*$
   $((2\hat{}LENGTH('f) + 2\hat{}LENGTH('f) - 1) + 1)$
   $/2\hat{}LENGTH('f)$"
   by (smt ...)
then have "$|a| \leq (2\hat{}(\text{exponent } c)/2\hat{}\text{bias } TYPE(('e,\ 'f)\ \text{float}))*$
   $2 * 2\hat{}LENGTH('f)/2\hat{}LENGTH('f)$"
   by fastforce
then have "$|a| \leq 2 * 2\hat{}(\text{exponent } c)/2\hat{}\text{bias}$
   $TYPE(('e,\ 'f)\ \text{float}) * 2\hat{}LENGTH('f)/2\hat{}LENGTH('f)$"
   by argo
then have abs_a_le : "$|a| \leq 2\hat{}(\text{exponent } c + 1)/2\hat{}\text{bias}$
   $TYPE(('e,\ 'f)\ \text{float}) * 2\hat{}LENGTH('f)/2\hat{}LENGTH('f)$"
   by force
```

Using the value of c and the fact that the distance to a is smaller than the distance between

a an b, we can prove an upper limit for a.

```
{
  assume to_disprove: "fraction $b \geq 1$"
  then have "$(2\hat{}LENGTH('f) + \text{real (fraction } b))/2\hat{}LENGTH('f) \geq (2\hat{}$
    $LENGTH('f) + 1)/2\hat{}LENGTH('f)$"
    by (simp add:divide_right_mono)
  then have "$(2\hat{}(\text{exponent } b)/2\hat{}\text{bias } TYPE(('e,\ 'f)\text{ float})) * (2\hat{}LENGTH('f)$
    $+\text{real (fraction } b))/2\hat{}LENGTH('f) \geq (2\hat{}(\text{exponent } b)/2\hat{}\text{bias}$
    $TYPE(('e,\ 'f)\text{float})) * (2\hat{}LENGTH('f) + 1)/2\hat{}LENGTH('f)$"
    using mult_le_cancel_left_pos[where ...]
    by (smt ...)
  then have bad_val_b_expanded : "$|\text{valof } b| \geq (2\hat{}(\text{exponent } b)/2\hat{}\text{bias}$
    $TYPE(('e,\ 'f)\text{ float})) * (2\hat{}LENGTH('f) + 1)/2\hat{}LENGTH('f)$"
    using a b exp_c_exp_b_min_1 by (simp add:abs_valof_eq2)
```

To prove the value of the fraction of b, we again use an assumption that we want to disprove. With this assumption we can determine a lower bound for the absolute value of b.

```
    with a b exp_c_exp_b_min_1 abs_a_le have "$|\text{valof } b - a| \geq (2\hat{}$
      $(\text{exponent } b)/2\hat{}\text{bias } TYPE(('e,\ 'f)\text{ float})) * (2\hat{}LENGTH('f) + 1)/2\hat{}$
      $LENGTH('f) - 2\hat{}(\text{exponent } b)/2\hat{}\text{bias } TYPE(('e,\ 'f)\text{ float}) * 2\hat{}$
      $LENGTH('f)/2\hat{}LENGTH('f)$"
      by force
    with a b have "$|\text{valof } b - a| \geq \text{ulp } b$"
      apply(simp add:ulp_equivelance ulp_divisor_rewrite bias_def) by argo
    moreover from ulp_positive have "$0.5 * \text{ulp } b < \text{ulp } b$"
      by auto
    ultimately have "False"
      using assms ulp_accuracy_def ulp_positive by force
  }note to_disprove_false3 = this
```

Using the upper bound of a, we can show that the distance between a and b would be too big if the assumptions called "to_disprove" was correct.

```
then have frac_b : "fraction $b = 0$"
    by fastforce
from a b have val_b_eq : "$|\text{valof } b| = (2\hat{}(\text{exponent } b)/2\hat{}\text{bias } TYPE(('e,\ 'f)$
  $\text{float})) * (2\hat{}LENGTH('f))/2\hat{}LENGTH('f)$"
  by (simp add:abs_valof_eq2 frac_b)
with val_b_min_val_c val_c_eq a b assms have "$|\text{valof } b - \text{valof } c| = 2\hat{}$
  $\text{exponent } c/2\hat{}\text{bias } TYPE(('e,\ 'f)\text{ float})/2\hat{}LENGTH('f)$"
  using closest_mean_ulp_0_5_helper_rewrite_to_ulp
  using exp_c_exp_b_min_1 by metis
with a b exp_c_exp_b_min_1 have diff_is_ulp_c : "$|\text{valof } b - \text{valof } c| = \text{ulp } c$"
  by (simp add:ulp_equivelance ulp_divisor_rewrite bias_def)
```

Knowing the fraction of b, we can determine the distance between b and c.

```
from abs_a_le a b exp_c_exp_b_min_1 val_b_eq have upper_bound :
  "$|a| \leq |\text{valof } b|$"
  by fastforce
from assms ulp_accuracy_def have "$|\text{valof } b - a| \leq 0.5 * \text{ulp } b$"
  by blast
with a b ulp_b_2_ulp_c have "$|\text{valof } b| - \text{ulp } c \leq |a|$"
  by auto
with val_b_min_val_c diff_is_ulp_c have lower_bound : "$|\text{valof } c| \leq |a|$"
  by argo
```

Using the distances in ulps, we can give a new upper and lower bound for a. To be more specific, the lower bound is c and the upper bound is b.

```
have c_positive_a_positive : "valof $c \geq 0 \to a \geq 0$"
  using a b exp_c_exp_b_min_1 assms(3) diff_0_right finite_zero nle_le
  zero_val_exponent by fastforce
with lower_bound upper_bound sign_equality a b valof_nonneg have sign_0 :
  "sign $c = 0 \to |\text{valof } b - \text{valof } c| = |\text{valof } c - a| + |\text{valof } b - a|$"
  by auto
have c_positive_a_positive : "valof $c \leq 0 \to a \leq 0$"
  using a b exp_c_exp_b_min_1 assms(3) diff_0_right finite_zero nle_le
  zero_val_by fastforce
with lower_bound upper_bound sign_equality a b valof_nonpos have
  "sign $c = 1 \to |\text{valof } b - \text{valof } c| = |\text{valof } c - a| + |\text{valof } b - a|$"
  by auto
with sign_0 sign_cases have "$|\text{valof } b - \text{valof } c| = |\text{valof } c - a| + |\text{valof } b - a|$"
  by metis
```

With the upper and lower bound, we can prove that a is between b and c.

```
    moreover from assms ulp_accuracy_def have "$|\text{valof } b - a| \geq |\text{valof } c - a|$"
      by blast
    ultimately have "$|\text{valof } c - a| \leq 0.5 * \text{ulp } c$"
      using ulp_positive diff_is_ulp_c by argo
  }
  with assms ulp_accuracy_def show "$\neg$(exponent $b \leq$ exponent $c$
      $\vee$ exponent $b = 1) \Rightarrow$ ulp_accuracy $a$ $c$ 0.5"
      by blast
qed
done
```

Using the fact that a is closer to c than to b, we know that the distance between a and c is smaller than half of the distance between b and c. This makes the accuracy of c to a expressed in ulps 0.5.

7.2 Error propagation through addition

To prove a bound for the error propagation through addition, we use two lemmas. We will first describe a proof that makes a claim regarding the distance between the sum of the reals and the sum of the floats. This is different from the addition operator for floats since this does not include rounding.

```
lemma addition_error_propagation_distance :
  assumes "|valof $a\_float$ + valof $b\_float$| < threshold $TYPE(('e, 'f)$ float)"
  shows "|(valof $a\_float$ + valof $b\_float$) − ($a\_real + b\_real$)| ≤ ($a\_accuracy$+
    $b\_accuracy$) ∗ ulp (fadd $To\_nearest$ $a\_float$ $b\_float$)"
```

To construct a proof for error propagation through addition, we use some preconditions. These preconditions have been described in Section 6.1. There we also describe the context in which we prove this lemma. This means that there are more preconditions than those listed here.

```
proof −
  have step_1 : "valof (fadd $To\_nearest$ $a\_float$ $b\_float$) = valof (
    round $To\_nearest$ (valof $a\_float$ + valof $b\_float$))"
    apply(simp add:fadd_def valof_zerosign del:round.simps)
    using assms ulp_accuracy_def float_distinct_finite by blast
  moreover obtain rounded_sum where rounded_sum_def : "$rounded\_sum$ = (round
    $To\_nearest$ (valof $a\_float$ + valof $b\_float$)"
    by blast
  ultimately have ulp_same : "ulp (fadd $To\_nearest$ $a\_float$ $b\_float$) = ulp
    $rounded\_sum$"
    using abs_valof_e_exp_e ulp_equivelance by metis
```

First we introduce a variable called *rounded_sum*. This is equal to the result of the addition for floats which is rounded. We express the errors in ulps of a specific floating point number. The resulting error should be expressed in ulps of the resulting floating point number. In other words, the resulting error should be expressed in ulps of *rounded_sum*.

```
from assms ulp_accuracy_def have finite_a_b : "is_finite $a\_float$ ∧
  is_finite $b\_float$"
  by blast
with addition_rounding_increases valof_nonneg assms rounded_sum_def have
  "valof $a\_float$ ≤ valof $rounded\_sum$ ∧ valof $b\_float$ ≤ valof $rounded\_sum$"
  by (metis ...)
with valof_nonneg assms abs_of_nonneg have "|valof $a\_float$| ≤
  |valof $rounded\_sum$| ∧ |valof $b\_float$| ≤ |valof $rounded\_sum$|"
  by (smt ...)
```

Since we calculate the addition of positive numbers, we know that the result is greater or equal to the input variables. This also means that an ulp of *rounded_sum* is greater or equal to an ulp of an *a_float* or *b_float*.

```
with abs_valof_ge_exp_ge exp_ge_ulp_ge assms ulp_accuracy_non_negative have
  "$a\_accuracy$ ∗ ulp $a\_float$ ≤ $a\_accuracy$ ∗ ulp $rounded\_sum$ ∧ $b\_accuracy$ ∗ ulp
  $b\_float$ ≤ $b\_accuracy$ ∗ ulp $rounded\_sum$"
  using mult_left_mono by metis
```

This also means that an ulp of $rounded\_sum$ is greater or equal to an ulp of an a_float or b_float.

```
moreover from assms ulp_accuracy_def have "$|$valof $a\_float - a\_real| \leq$
   $a\_accuracy *$ ulp $a\_float \wedge |$valof $b\_float - b\_real| \leq b\_accuracy *$ ulp $b\_float$"
   by fast
ultimately have rounded_sum_dist : "$|$(valof $a\_float$ + valof $b\_float) - (a\_real+$
   $b\_real)| \leq (a\_accuracy + b\_accuracy) *$ ulp $rounded\_sum$"
   by argo
```

In addition we can simply add the distances, and thus add the errors.

```
      with ulp_same show ?$thesis$ by presburger
   qed
```

Together with the fact that an ulp from $rounded\_sum$ is the same as an ulp of the result of the "fadd" operator, we can prove this lemma.

The second proof expresses the error bound of the "fadd" operator. There are no new preconditions that are not already in the previous lemma.

```
   lemma addition_error_propagation_ulp_accuracy :
      assumes "$|$valof $a\_float$ + valof $b\_float| <$ threshold $TYPE(('e,\ 'f)$ float)"
      shows "ulp_accuracy $(a\_real + b\_real)$ (fadd $To\_nearest\ a\_float\ b\_float$)
         $(a\_accuracy + b\_accuracy + 0.5)$"
```

The difference with the previous lemma is that the postcondition now describes an "ulp_accuracy" instead of a distance. This also means that the accuracy is measured between the real value and the rounded floating point value instead of the unrounded sum of the floating point values.

```
proof -
   have step_1 : "valof (fadd $To\_nearest\ a\_float\ b\_float$) = valof (round $To\_nearest$
      (valof $a\_float$ + valof $b\_float$))"
      apply(simp add:fadd_def valof_zerosign del:round.simps)
      using assms ulp_accuracy_def float_distinct_finite by blast
   moreover obtain rounded_sum where rounded_sum_def : "$rounded\_sum$ = (round
      $To\_nearest$ (valof $a\_float$ + valof $b\_float$))"
      by blast
   ultimately have ulp_accuracy_same : "ulp_accuracy $(a\_real + b\_real)$(fadd
      $To\_nearest\ a\_float\ b\_float)$ $(a\_accuracy + b\_accuracy + 0.5)$ = ulp_accuracy
      $(a\_real + b\_real)$ $rounded\_sum$ $(a\_accuracy + b\_accuracy + 0.5)$"
      using valof_same_ulp_accuracy_same by blast
   from step_1 rounded_sum_def have ulp_same : "ulp (fadd $To\_nearest\ a\_float$
      $b\_float$) = ulp $rounded\_sum$"
      using abs_valof_e_exp_e exp_e_ulp_e by metis
```

We again start with defining $rounded\_sum$ as a floating point number since we defined ulps to be based on a floating point value and not a real value.

```
   with assms addition_error_propagation_distance have "$|$(valof $a\_float$ + valof
      $b\_float) - (a\_real + b\_real)| \leq (a\_accuracy + b\_accuracy) *$ ulp $rounded\_sum$"
      by metis
   moreover from rounding_0_5_ulp ulp_accuracy_def assms rounded_sum_def have
      "$|$valof $rounded\_sum$ − (valof $a\_float$ + valof $b\_float)| \leq 0.5 *$ ulp $rounded\_sum$"
      by fast
```

Next, we use two other lemmas to get the distance between the sum of the floating point values and the sum of the real values. The first of these lemmas is the one we discussed previously in this section. The other lemma makes a claim about the error introduced through rounding.

```
ultimately have rounded_sum_dist : "|valof $rounded\_sum - (a\_real + b\_real)| \leq$
  $(0.5 + a\_accuracy + b\_accuracy) *$ ulp $rounded\_sum$"
  by argo
```

These two distances can be added to get the total distance.

```
  from is_finite_closest have "is_finite (closest valof ($\lambda a.$ even (fraction $a$))
    (Collect is_finite) (valof $a\_float$ + valof $b\_float$))"
    by blast
  with rounded_sum_def assms have "is_finite $rounded\_sum$"
    by fastforce
  with rounded_sum_dist have "ulp_accuracy $(a\_real + b\_real)$ $rounded\_sum$
    $(a\_accuracy + b\_accuracy + 0.5)$"apply (simp add:ulp_accuracy_def)
    by argo
  with ulp_accuracy_same show ?$thesis$ by fast
qed
```

Together with the fact that the result is finite, we can prove the "ulp_accuracy" of the resulting floating point value.

7.3 Error propagation through dot product

To prove a bound on the error propagation through a dot product, we use three lemmas. First, we prove an error bound for a sum of products. The products are between elements of the vectors with the same indices. This does not mean every index is used once. The second lemma proves an error bound for a dot products meaning that every index is used exactly once. Lastly, the third lemma will use the dot product function on vectors of reals as it is defined by René Thiemann and Akihisa Yamada [31] instead of our definition for a dot product between vectors of reals.

```
lemma vector_multiplication_helper :
  assumes a_rel : "vec_all2 (λf r. ulp_accuracy r f a_accuracy)
      afs ars"
  and b_rel : "vec_all2 (λf r. ulp_accuracy r f b_accuracy)
      bfs brs"
  and signs : "vec_all (λf. sign f = 0) afs ∧ vec_all (λf. sign f = 0) bfs"
  and probs : "vec_all (λr. 0 ≤ r ∧ r ≤ 1) ars ∧ vec_all (λr. 0 ≤ r ∧ r ≤ 1) brs"
  and "1 < LENGTH('e)"
  and "a_accuracy ≤ 2^LENGTH('f)"
  and "b_accuracy ≤ 2^LENGTH('f)"
  and "1 < LENGTH('f)"
  and "is_finite (fold (λi s. fmul_add To_nearest (afs $ nat i) (bfs $ nat i) s) is 0)"
  and "dim_vec afs = dim_vec bfs"
  and "∀i ∈ set is. (0 ≤ i ∧ i < dim_vec afs)"
  shows "ulp_accuracy
      (fold (λi s. (ars $ nat i) * (brs $ nat i) + s) is 0) (fold (λi s. fmul_add
      To_nearest (afs $ nat i) (bfs $ nat i) s) is 0) ((2 * a_accuracy+
      2 * b_accuracy + 0.5) * (length is)) ∧ sign (fold (λi s. fmul_add
      To_nearest (afs $ nat i) (bfs $ nat i) s) is 0) = 0"
```

This first proof uses a list of indices that are multiplied and added. This results in the formulas "fold ($\lambda i\ s.\ (ars\ \$\ \text{nat}\ i) * (brs\ \$\ \text{nat}\ i) + s)\ is\ 0$" and "fold ($\lambda i\ s.$ fmul_add $To\_nearest\ (afs\ \$\ \text{nat}\ i)\ (bfs\ \$\ \text{nat}\ i)\ s)\ is\ 0$". Apart from this formula, the preconditions used for this lemma are not very different from the preconditions described in Section 6.2. The formula that is new uses the "fold" function. This can be used to go over a list and apply a function for each element. In our formula this would mean the following:

```
fold (λi s. (ars $ nat i) * (brs $ nat i) + s)) (x#xs) 0 =
    fold (λi s. (ars $ nat i) * (brs $ nat i) + s) xs ((ars $ nat x) * (brs $ nat x) + 0)
```

This shows that the list *is* is a list of indices for the vectors. For each index, the corresponding elements of the vectors are multiplied. These results are then added and the total sum is returned.

```
using assms proof (induction "is" rule: rev_induct)
```

The proof is constructed used a rule called "rev_induct". This is an induction proof but in reverse. Induction proofs have two steps. First, there is the base case, which for lists is an empty list. The second case should show that when the theorem is true for a list of length n, it should also hold for length $n + 1$. In Isabelle this means the theorem should also hold when an element is prepended to a list. Using "rev_induct" means an element is appended to a list.

```
case Nil
with ulp_accuracy_zero zero_simps(1) show ?case apply simp by blast
```

The first case is using an empty list. This can be proven in just one line.

```
next
  case (snoc x xs)
  from snoc(12) have subset_precondition : "∀i ∈ set (xs). (0) ≤ i ∧ i < dim_vec
    afs"
    by force
```

For the second case we start by rewriting one of the preconditions such that we know each index in xs is valid index.

```
from fold_append have folded_floats : "(fold (λi s. fmul_add To_nearest
   (afs $ nat i) (bfs $ nat i) s) (xs@[x]) 0) = fmul_add To_nearest (afs $ nat x)
   (bfs $ nat x) (fold (λi s. fmul_add To_nearest (afs $ nat i) (bfs $ nat i) s)
   (xs) 0)"
   by simp
with three_mul_added_numbers(4) snoc(10) have fin_c : "is_finite (fold
   (λi s. fmul_add To_nearest (afs $ nat i) (bfs $ nat i) s) (xs) 0)"
   by auto
```

After that, we use the definition of the "fold" function to rewrite "fold ($\lambda i\ s$. fmul_add *To_nearest* (afs \$ nat i) (bfs \$ nat i) s) (xs@[x]) 0". Furthermore we rewrite a second precondition.

```
from snoc(12) have a_l : "nat x < dim_vec afs"
   by auto
with vec_all2_dim snoc(2) have a_l2 : "nat x < dim_vec ars"
   by metis
from a_l snoc(11) have b_l : "nat x < dim_vec bfs"
   by presburger
with vec_all2_dim snoc(3) have b_l2 : "nat x < dim_vec brs"
   by metis
```

Next, we prove that x is a valid index for each vector.

```
have u_a1 : "ulp_accuracy (ars $ nat x) (afs $ nat x) a_accuracy"
   using vec_all2_map[OF snoc(2) a_l] by blast
have u_a2 : "ulp_accuracy (brs $ nat x) (bfs $ nat x) b_accuracy"
   using vec_all2_map[OF snoc(3) b_l] by blast
have u_a3 : "ulp_accuracy (fold (λi. (+) (ars $ nat i * brs $ nat i)) xs 0)
   (fold (λi. fmul_add To_nearest (afs $ nat i) (bfs $ nat i)) (xs)0)
   ((2 * a_accuracy + 2 * b_accuracy + 5/10) * real (length (xs)))"
   using snoc fin_c subset_precondition by blast
have s1 : "sign (afs$ nat x) = (0)"
   using snoc(4) by (simp add: a_l vec_all_def)
have s2 : "sign (bfs$ nat x) = (0)"
   using snoc(4) by (simp add: b_l vec_all_def)
have s3 : "sign ((fold (λi. fmul_add To_nearest (afs $ nat i)
   (bfs $ nat i)) (xs) 0)) = 0"
   using snoc fin_c subset_precondition by blast
have fin_total : "is_finite (fmul_add To_nearest (afs $ nat x) (bfs $ nat x)
   (fold (λi s. fmul_add To_nearest (afs $ nat i) (bfs $ nat i) s) (xs) 0))"
   using folded_floats snoc(10) by argo
have r1 : "0 ≤ ars $ nat x"
   using snoc(5) by (simp add: a_l2 vec_all_def)
have r2 : "ars $ nat x ≤ 1"
   using snoc(5) by (simp add: a_l2 vec_all_def)
have r3 : "0 ≤ brs $ nat x"
   using snoc(5) by (simp add: b_l2 vec_all_def)
have r4 : "brs $ nat x ≤ 1"
   using snoc(5) by (simp add: b_l2 vec_all_def)
```

We use the "fmul_add" function and we have proven how errors propagate through that operator. To use that lemma, we prove each precondition.

```
    from multiplication_addition_error_propagation_ulp_accuracy2[OF u_a1 u_a2
      u_a3 snoc(6) snoc(9) s1 s2 s3 fin_total snoc(7) snoc(8) r1 r2 r3 r4] show ?case
      apply simp using three_mul_added_numbers_positive[OF fin_total s1 s2 s3]
      by argo
  qed
```

Using all the preconditions combined with the lemma on error propagation through the "fmul_add" operator, we can finish this proof. The next step will be to use the "float_vec_mul" and "real_vec_mul" functions instead of the "fold" function to use each index exactly once.

```
lemma vector_multiplication_real_vec_mul :
  assumes a_rel : "vec_all2 (λf r. ulp_accuracy r f a_accuracy) afs ars"
  and b_rel : "vec_all2 (λf r. ulp_accuracy r f b_accuracy) bfs brs"
  and signs : "vec_all (λf. sign f = 0) afs ∧ vec_all (λf. sign f = 0) bfs"
  and probs : "vec_all (λr. 0 ≤ r ∧ r ≤ 1) ars ∧ vec_all (λr. 0 ≤ r ∧ r ≤ 1) brs"
  and len_e : "1 < LENGTH('e)"
  and lim_a : "a_accuracy ≤ 2^LENGTH('f)"
  and lim_b : "b_accuracy ≤ 2^LENGTH('f)"
  and len_f : "1 < LENGTH('f)"
  and fin : "is_finite (float_vec_mul afs bfs)"
  and dim : "dim_vec afs = dim_vec bfs"
  shows "ulp_accuracy (real_vec_mul ars brs) (float_vec_mul afs bfs)
      ((2 * a_accuracy + 2 * b_accuracy + 0.5) * dim_vec afs) ∧ sign
      (float_vec_mul afs bfs) = 0"
  using assms(9) apply(simp add:real_vec_mul_def float_vec_mul_def)
      using assms(1) assms(10) vec_all2_dim vector_multiplication_helper[where ?is
      = "[0..int(dim_vec afs) − 1]", OF assms(1) assms(2) assms(3)
      assms(4) assms(5) assms(6) assms(7) assms(8)] by force
```

The "float_vec_mul" and "real_vec_mul" functions are defined with a "fold" and use "$[0..int(\text{dim\_vec } afs) - 1]$" as *is*. This means that the proof can be very short if we have proven the previous lemma.

```
lemma vector_multiplication :
  assumes $a\_rel$ : "vec_all2 ($\lambda f\ r.$ ulp_accuracy $r\ f\ a\_accuracy$) $afs\ ars$"
  and $b\_rel$ : "vec_all2 ($\lambda f\ r.$ ulp_accuracy $r\ f\ b\_accuracy$) $bfs\ brs$"
  and $signs$ : "vec_all ($\lambda f.$ sign $f = 0$) $afs \wedge$ vec_all ($\lambda f.$ sign $f = 0$) $bfs$"
  and $probs$ : "vec_all ($\lambda r.\ 0 \leq r \wedge r \leq 1$) $ars \wedge$ vec_all ($\lambda r.\ 0 \leq r \wedge r \leq 1$) $brs$"
  and "$1 < LENGTH('e)$"
  and "$a\_accuracy \leq 2\hat{}LENGTH('f)$"
  and "$b\_accuracy \leq 2\hat{}LENGTH('f)$"
  and "$1 < LENGTH('f)$"
  and "is_finite (float_vec_mul $afs\ bfs$)"
  and "dim_vec $afs$ = dim_vec $bfs$"
  shows "ulp_accuracy ($ars \bullet brs$) (float_vec_mul $afs\ bfs$)(($2 * a\_accuracy +$
      $2 * b\_accuracy + 0.5) *$ dim_vec $afs$) $\wedge$ sign (float_vec_mul $afs\ bfs$) = 0"
  using vector_multiplication_real_vec_mul[OF a_rel b_rel signs probs assms(5)
      assms(6) assms(7) assms(8) assms(9) assms(10)] apply simp
      using assms(10) a_rel b_rel vec_all2_dim real_vec_mul_scalar_product
      by metis
```

Lastly, we want to use "$ars \bullet brs$" instead of "real_vec_mul $ars\ brs$". We have proven that these two functions are equal, so this proof can also be short.

Chapter 8

Comparison Of Results

As we said earlier, Roux et al. [29] expressed an error bound for a dot product between vectors using a relative bound. The error bound that they have gotten is the following.

$$|fl(\sum_{i=0}^{n-1} a_i\ b_i) - \sum_{i=0}^{n-1} a_i\ b_i| \leq \gamma_n(\sum_{i=0}^{n-1} |a_i\ b_i|) + 2n * eta$$

The variable eta for a floating point format is $2^{-bias-number\_of\_fraction\_bits}$. For a 64-bit system is 2^{-1075}. γ_n is defined in the following way:

$$\forall n \in \mathbb{N}, \gamma_n = n * eps/(1 - n * eps)$$

The variable eps is $2^{-1-number\_of\_fraction\_bits}$, so for a 64-bit system it is 2^{-53}. Furthermore, we introduce the function fl. This is the floating point evaluation of an expression from left to right.

Here we can see that the relative error is smaller or equal to $\gamma_n * 100\%$ of the real number value. With the definition of γ_n, we know that the error bound is $n * 2^{-53}/(1 - n * 2^{-53}) * 100\%$. As they have only proven a bound for normal numbers, this error bound assumes that, amongst others, the result of the dot product is a normal number. We have shown results for an error bound of matrix multiplication, but we also have lemmas regarding a dot product between vectors. The results are expressed differently as Roux et al. expressed the error as a factor of the real number value, but we expressed the error in ulps. The lemma that shows our error bound is the following.

assumes a_rel : "vec_all2 ($\lambda f\ r.$ ulp_accuracy r (f :: ($'e$, $'f$) float) $a\_accuracy$)
(afs :: ($'e$, $'f$) float vec) ars"
and b_rel : "vec_all2 ($\lambda f\ r.$ ulp_accuracy r (f :: ($'e$, $'f$) float) $b\_accuracy$)
(bfs :: ($'e$, $'f$) float vec) brs"
and $signs$: "vec_all ($\lambda f.$ sign $f = 0$) afs $\wedge$
vec_all ($\lambda f.$ sign $f = 0$) bfs"
and $probs$: "vec_all ($\lambda r.0 \leq r \wedge r \leq 1$) $ars \wedge$ vec_all ($\lambda r.0 \leq r \wedge r \leq 1$) brs"
and "$1 < LENGTH('e)$"
and "$a\_accuracy \leq (2$ :: real)$\hat{}LENGTH('f)$"
and "$b\_accuracy \leq (2$ :: real)$\hat{}LENGTH('f)$"
and "$1 < LENGTH('f)$"
and "is_finite (float_vec_mul afs bfs)"
and "dim_vec afs = dim_vec bfs"
shows "ulp_accuracy ($ars \bullet brs$) (float_vec_mul afs bfs)
$((2 * a\_accuracy + 2 * b\_accuracy + 0.5) *$ dim_vec $afs) \wedge$
sign (float_vec_mul afs bfs) = 0"

Lemma 8.1: Accuracy of a dot product between two vectors

Roux et al. started without any rounding errors, meaning that $a\_accuracy$ and $b\_accuracy$ are 0. This would lead to an error of "$0.5 *$ dim_vec afs" ulps. Obviously, this is not directly comparable since the errors are expressed in different ways. To compare this to the γ_n of Roux et al. we use the "rel_accuracy" function. This function expresses the accuracy in a factor of the floating point value.

definition rel_accuracy :: "real $\Rightarrow$ ($'e$, $'f$) float $\Rightarrow$ real $\Rightarrow$ bool"
where "rel_accuracy $a\ c\ u \equiv$ (is_finite c) $\wedge$ |(valof $c - a$)| $\leq$
|valof c| $* u/2\hat{}LENGTH('f)$"

We have proven lemmas how "rel_accuracy" and "ulp_accuracy" compare to transform one into the other.

assumes "rel_accuracy a (c :: ($'e$, $'f$) float) u"
and "$u \geq 0$"
shows "ulp_accuracy a c ($2 * u$)"

Lemma 8.2: Accuracy conversion from relative error to ulp error

assumes "ulp_accuracy a (c :: ($'e$, $'f$) float) u"
and "exponent $c \neq 0$"
shows "rel_accuracy a c u"

Lemma 8.3: Accuracy conversion from ulp error to relative error

We observe that, there is an extra preconditions for the conversion from "rel_accuracy" to "ulp_accuracy" and an extra precondition for the other way around. First, we have "$u \geq$

0". In practice, this is always true, except for 1 case. We already assume that "rel_accuracy $a\ c\ u$" is true, sow we know that "$|(\text{valof } c - a)| \leq |\text{valof } c| * u/2\hat{\ }LENGTH('f)$" is true. Since "$|(\text{valof } c - a)|$" is greater or equal to 0 and "$2\hat{\ }LENGTH('f)$" is greater than 0, we conclude that "$0 \leq |\text{valof } c| * u$". For u to be negative, "valof c" needs to be 0. Furthermore, "$|(\text{valof } c - a)|$" needs to be 0, so the only case where u is negative is when both "valof c" and a are 0.

The next precondition is "exponent $c \neq 0$". This means that c is a normal number. As was said earlier, normal numbers in the IEEE standard look something like 1.XXXX * 2 ^ e while subnormal numbers look like 0.XXXX * 2 ^ e. If we have the floating point number of 0.00...001 and have an error of a single ulp, then the relative error is 100% even though an error of a single ulp for any normal number is way smaller.

Using this "rel_accuracy" function together with the fact that the error bound was "$0.5 * \text{dim\_vec } afs$" ulps, we can conclude that for a dot product that results in a normal number, we have proven a relative error of "$0.5 * \text{dim\_vec } afs/2\hat{\ }LENGTH('f) * 100\%$" of the floating point value.

We observe that n in γ_n represents the dimension of the vector. Furthermore, we know that $LENGTH('f)$ in a 64-bit system is 52 This leads us to conclude that the bound we have proven is "$0.5 * n/2^{52} * 100\%$" or "$n/2^{53} * 100\%$" of the floating point value. To make the comparison fair, we need to acknowledge that the proof of Roux et al. does not use the "mul_add" operator. If they did, their error bound probably would still have been γ_n. If we did not use that operator, a rounding error would be introduced in both the multiplication and the addition. This would result in our error bound would being twice as bad and thus "$n/2^{52} * 100\%$" of the floating point value.

We can easily see that our bound of "$n/2^{53} * 100\%$" is better than "$n * 2^{-53}/(1 - n * 2^{-53}) * 100\%$". These bounds are however still not comparable since theirs is relative to the real value while ours is relative to the floating point value. To transform these bounds we will use the following formulae.

$$\begin{aligned}
&|x - y| \leq |x| * b \Rightarrow \\
&(1 - b) * |x| \leq |y| \leq (1 + b) * |x| \Rightarrow \\
&|x| \leq |y|/(1 - b) \land |y|/(1 + b) \leq |x| \Rightarrow \\
&|x - y| \leq |y| * |\frac{1}{1 - b} - 1| \land |y| * |\frac{1}{1 + b} - 1| \leq |x - y| \Rightarrow \\
&|x - y| \leq |y| * |\frac{b}{1 - b}| \land |y| * |\frac{b}{1 + b}| \leq |x - y|
\end{aligned}$$

Here we have two values x and y and the difference between the two values is $b * 100\%$ of $|x|$. We observe that if we want to express the difference to be a percentage of $|y|$, we get an error bound of $|\frac{b}{1 + b}| * 100\%$. We do make some assumptions such as that bound b is positive and not equal to 1.

We can use this to transform our relative error bound to be "$n * 2^{-53}/(1 - n * 2^{-53}) * 100\%$" of the real value and thus exactly the same as the relative error bound of Roux et al. If we transform their error bound to be relative to the floating point value, however, we get an error bound of "$n * 2^{-53}/(1 - 2 * n * 2^{-53}) * 100\%$", which is bigger and thus worse than our error bound. This leads us to conclude that our method for determining an error bound for a dot product between 2 vectors is at worst equal and at best better than the method of Roux et al. Furthermore, we have a note about the "rel_accuracy" function used earlier. This relative accuracy is only for normal numbers. When a normal number looks like "$1.00...000 * 2^e$", an ulp accuracy of X would result in a relative error bound of "$X/2\hat{\ }LENGTH('f) * 100\%$". If the normal number looks like "$1.11...111 * 2^e$", the relative bound is actually "$X/2\hat{}(LENGTH('f) + 1) * 100\%$". In this conversion between "ulp_accuracy" and "rel_accuracy" we assume the worst case. This again demonstrates that only under the worst assumptions, our error bound is equal to the bound of Roux et al. but there are many cases where our error bound is better.

We do, however, have to mention that our error bound has more preconditions such as real numbers being between 0 and 1. This means that this comparison cannot be made for all dot products between a vector and a matrix. Furthermore, our error bound is heavily dependent on whether the "fmul_add" operator is used or not while their error bound is not. Both are arguments to conclude our method is worse.

We cannot make a comparison between our bound and theirs with regards to error propagation. The reason for this is that Roux et al. did not prove such a bound and we cannot conclude what their bound would be for iterative matrix multiplication. The relative part of γ_n is based on the number of chained operations. This grows linearly in iterative matrix multiplication, instead of exponentially like our bound. However, the problem is that their bound also includes an absolute part. This absolute part makes it hard to prove a bound for iterative matrix multiplication, just like we explained in Section 2.2. We expect that this is the reason they did not prove a bound for iterative matrix multiplication, and therefore also the reason we cannot compare these bounds for executing the multiplication more that once.

Chapter 9

Answers To Research Questions

9.1 Question 1

The first question was stated as "What are the necessary lemmas to prove an error bound for algorithms measured in ulps?". We answer this question in three sections. First, we give an overview of the major lemmas. After that, we go into detail about the limitations of these lemmas. Lastly, we will summarize the answer to this question.

9.1.1 Overview of lemmas

There are three lemmas for proving the error introduced by rounding. The first one proves that as long as a real number is not rounded to infinity, there exists a floating point number with an accuracy of 0.5 ulp or better. The second lemma proves that as long as a floating point number exists with an accuracy of 0.5 ulp, then the closest floating point number also has an accuracy of 0.5 ulp or better. The third lemma combines these two lemmas to prove that the rounding operation introduces an error or 0.5 ulp or less.

```
assumes  "$|a| <$ threshold $TYPE(('e,\ 'f)$ float)"
     and "$1 < LENGTH('e)$"
   shows "$\exists(b :: ('e,\ 'f)$ float).ulp_accuracy $a\ b$ 0.5"
```

Lemma 9.1: There exists a float with an accuracy of 0.5 ulps

```
assumes "ulp_accuracy $a$ $(b :: ('e,\ 'f)$ float) 0.5"
     and "$LENGTH('f) > 1$"
     and "$(\forall(d :: ('e,\ 'f)$ float).$d \in a$. is_finite $a \rightarrow$
        $|$valof $(c :: ('e,\ 'f)$ float$) - a| \leq |$valof $d - a|)$"
     and "is_finite $c$"
   shows "ulp_accuracy $a\ c$ 0.5"
```

Lemma 9.2: Being closest implies an accuracy of 0.5 ulps

assumes "$|a| <$ threshold $TYPE(('e,'f)float)$"
    and "$a_r = $ ((round $To\_nearesta$) :: $('e,\ 'f)$ float)"
    and "$1 < LENGTH('e)$"
    and "$1 < LENGTH('f)$"
  shows "ulp_accuracy a a_r 0.5"

Lemma 9.3: Rounding creates an accuracy of 0.5 ulp

As was explained in Section 6.1, we prove all lemmas regarding error propagations inside a context. This means that all lemmas have the assumptions of the context. These assumptions are the following.

context
    fixes a_float b_float c_float :: "$('e,\ 'f)$float"
     and a_real b_real c_real :: "real"
     and $a\_accuracy$ $b\_accuracy$ $c\_accuracy$ :: "real"
    assumes a_rel : "ulp_accuracy a_real a_float $a\_accuracy$"
        and b_rel : "ulp_accuracy b_real b_float $b\_accuracy$"
        and c_rel : "ulp_accuracy c_real c_float $c\_accuracy$"
        and len_e : "$1 < LENGTH('e)$"
        and len_f : "$1 < LENGTH('f)$"
        and $sign\_a$: "sign $a\_float = 0$"
        and $sign\_b$: "sign $b\_float = 0$"
        and $sign\_c$: "sign $c\_float = 0$"

Lemma 9.4: Preconditions of this context

We have proven lemmas on how the error propagates through basic operations. For addition, the resulting error is the sum of the input errors.

assumes "|valof a_float + valof $b\_float| <$ threshold $TYPE(('e,\ 'f)$ float)"
  shows "|(valof a_float + valof $b\_float) - (a\_real + b\_real)| \leq$
      $(a\_accuracy + b\_accuracy) *$ ulp (fadd $To\_nearest$ a_float b_float)"

Lemma 9.5: Addition adds the input accuracies

For multiplication, the accuracy depends on whether the floating point values are normal or subnormal. When both numbers are subnormal, the resulting error is the sum of the input errors. When one is normal and the other is subnormal, the resulting error is twice the error of the normal number plus the error of the subnormal number. When both are normal, the resulting error is twice the sum of the input errors.

assumes "exponent $a\_float = 0$"
and "exponent $b\_float = 0$"
and "$a\_real \leq 1$"
and "$a\_real \geq 0$"
and "|valof $a\_float *$ valof $b\_float| <$ threshold $TYPE(('e,\ 'f)$ float)"
shows "|(valof $a\_float *$ valof $b\_float) - (a\_real * b\_real)| \leq$
$(a\_accuracy + b\_accuracy) *$ ulp (fmul $To\_nearest\ a\_float\ b\_float$)"

Lemma 9.6: Multiplication of subnormal floats adds the input accuracies if a is between 0 and 1

assumes "exponent $a\_float = 0$"
and "exponent $b\_float = 0$"
and "$b\_real \leq 1$"
and "$b\_real \geq 0$"
and "|valof $a\_float *$ valof $b\_float| <$ threshold $TYPE(('e,\ 'f)$ float)"
shows "|(valof $a\_float *$ valof $b\_float) - (a\_real * b\_real)| \leq$
$(a\_accuracy + b\_accuracy) *$ ulp (fmul $To\_nearest\ a\_float\ b\_float$)"

Lemma 9.7: Multiplication of subnormal floats adds the input accuracies if b is between 0 and 1

assumes "exponent $a\_float \neq 0$"
and "exponent $b\_float = 0$"
and "$a\_real \leq 1$"
and "$a\_real \geq 0$"
and "|valof $a\_float *$ valof $b\_float| <$ threshold $TYPE(('e,\ 'f)$ float)"
shows "|(valof $a\_float *$ valof $b\_float) - (a\_real * b\_real)| \leq$
$(2 * a\_accuracy + b\_accuracy) *$
ulp (fmul $To\_nearest\ a\_float\ b\_float$)"

Lemma 9.8: Multiplication of a normal and a subnormal float doubles the accuracy of the normal number and adds the accuracy of the subnormal number if a is between 0 and 1

assumes "exponent $b\_float \neq 0$"
and "exponent $a\_float = 0$"
and "$b\_real \leq 1$"
and "$b\_real \geq 0$"
and "$|$valof $a\_float *$ valof $b\_float| <$ threshold $TYPE(('e,\ 'f)$ float)"
shows "$|$(valof $a\_float *$ valof $b_float) $- (a\_real * b\_real)| \leq$
$(a\_accuracy + 2 * b\_accuracy)*$
ulp (fmul $To\_nearest\ a\_float\ b\_float$)"

LEMMA 9.9: Multiplication of a normal and a subnormal float doubles the accuracy of the normal number and adds the accuracy of the subnormal number if b is between 0 and 1

assumes "exponent $a\_float \neq 0$"
and "exponent $b\_float \neq 0$"
and "$a\_accuracy \leq (2 ::$ real)$\hat{}LENGTH('f)$"
and "$b\_accuracy \leq (2 ::$ real)$\hat{}LENGTH('f)$"
and "$|$valof $a\_float *$ valof $b\_float| <$ threshold $TYPE(('e,\ 'f)$ float)"
shows "$|$(valof $a\_float *$ valof $b_float) $- (a\_real * b\_real)| \leq$
$(2 * a\_accuracy + 2 * b\_accuracy)*$
ulp (fmul $To\_nearest\ a\_float\ b\_float$)"

LEMMA 9.10: Multiplication of subnormal floats doubles and adds the input accuracies

Without the knowledge of whether the floating point numbers are normal or subnormal, we have proven that the resulting error is better or equal to twice the sum of the input errors.

assumes "$|$valof $a\_float *$ valof $b\_float| <$ threshold $TYPE(('e,\ 'f)$ float)"
and "$a\_accuracy \leq (2 ::$ real)$\hat{}LENGTH('f)$"
and "$b\_accuracy \leq (2 ::$ real)$\hat{}LENGTH('f)$"
and "$0 \leq a\_real$"
and "$a\_real \leq 1$"
and "$0 \leq b\_real$"
and "$b\_real \leq 1$"
shows "$|$(valof $a\_float *$ valof $b_float) $- (a\_real * b\_real)| \leq$
$(2 * a\_accuracy + 2 * b\_accuracy)*$
ulp (fmul $To\_nearest\ a\_float\ b\_float$)"

LEMMA 9.11: Multiplication doubles and adds the input accuracies

Combining these error bounds with the error introduced in the rounding step, we can prove an error bound for the "fadd", "fmul", and "fmul_add" operators.

```
assumes "$|$valof $a\_float$ + valof $b\_float| <$ threshold $TYPE(('e, 'f)$ float)"
  shows "ulp_accuracy $(a\_real + b\_real)$
      (fadd $To\_nearest$ $a\_float$ $b\_float$)
      $(a\_accuracy + b\_accuracy + 0.5)$"
```

Lemma 9.12: Accuracy of the fadd operator

```
assumes "$|$valof $a\_float * $ valof $b\_float| <$ threshold $TYPE(('e, 'f)$ float)"
    and "$a\_accuracy \leq (2 ::$ real)$\hat{}LENGTH('f)$"
    and "$b\_accuracy \leq (2 ::$ real)$\hat{}LENGTH('f)$"
    and "$0 \leq a\_real$"
    and "$a\_real \leq 1$"
    and "$0 \leq b\_real$"
    and "$b\_real \leq 1$"
  shows "ulp_accuracy $(a\_real * b\_real)$ (fmul $To\_nearest$ $a\_float$ $b\_float$)
      $(2 * a\_accuracy + 2 * b\_accuracy + 0.5)$"
```

Lemma 9.13: Accuracy of the fmul operator

```
assumes "$|$valof $a\_float * $ valof $b\_float$ + valof $c\_float| <$
      threshold $TYPE(('e, 'f)$ float)"
    and "$a\_accuracy \leq (2 ::$ real)$\hat{}LENGTH('f)$"
    and "$b\_accuracy \leq (2 ::$ real)$\hat{}LENGTH('f)$"
    and "$0 \leq a\_real$"
    and "$a\_real \leq 1$"
    and "$0 \leq b\_real$"
    and "$b\_real \leq 1$"
  shows "ulp_accuracy $(a\_real * b\_real + c\_real)$
      (fmul_add $To\_nearest$ $a\_float$ $b\_float$ $c\_float$)
      $(2 * a\_accuracy + 2 * b\_accuracy + c\_accuracy + 0.5)$"
```

Lemma 9.14: Accuracy of the fmul_add operator

9.1.2 Limitations of the lemmas

The previous section shows a lot of necessary lemmas. One important note to make is that these lemmas are only useful for a subset of algorithms. To determine for which algorithms we can prove an error bound, we can look at the preconditions of the lemmas. In Section 6.1 we discuss why we use these preconditions. Here we will discuss how these preconditions limit us in proving error bounds for algorithms. The lemma regarding the error propagation through the "fmul_add" operator 9.14 has all preconditions that limit the set of algorithms on which we can prove an error bound. We first look at the context of 9.4. The assumptions a_rel, b_rel, and c_rel are used to give meaning to all the variables. They also give the first limitation. For the "ulp_accuracy" function to be true, the floating point number has to be finite. This means that our lemmas are only applicable to programs where only finite numbers are used. The next assumptions are *len_e* and *len_f*. They claim the number of bits used for the fraction part and the exponent part are more than one. Since most computers use a total of 32 or 64 bits to represent a floating point number, this precondition does not limit the number of algorithms for which we can prove error bounds. The last assumptions in this context are *sign_a*, *sign_b*, and *sign_c*. We assume the signs of the floating point numbers are 0. This means that the values of the floating point values are greater or equal to 0. Therefore, our lemmas are only applicable to programs that only use positive numbers.

Next, we look at the preconditions specific to the lemma that proves an error bound of the "fmul_add" operator 9.14. The first precondition claims that the absolute value of the floating points added and multiplied is smaller than the threshold of the floating point format. This means that the resulting floating point value will remain finite and limits our lemma to programs where the outcome is finite. The next preconditions, *lim_a* and *lim_b*, limit *a_accuracy* and *b_accuracy*. Since the error bound only increases for each operator when using our lemmas, this means that after a certain amount of operations, the error bound gets too big. In practice, this means that our lemmas are only applicable to algorithms that are not too long and do not have too many operations. Lastly, 4 assumptions limit the real values *a_real* and *b_real*. These values are limited to being between 0 and 1. This is the case for certain algorithms that calculate probabilities but for algorithms for which this is not the case, we cannot claim an error bound due to these assumptions.

Next to the preconditions, there is another important limitation. We have only proven lemmas for error propagation through multiplication and addition. We did not prove anything for other operations like division or subtraction. This means that we can only prove an error bound for an algorithm that uses only addition and multiplication.

9.1.3 Answer to the question

To answer this question, we need to provide the lemmas that show how errors are introduced and propagated when floating point operations are used. These lemmas are 9.12, 9.13, and 9.14. Here we see that in addition, the input errors are added, while for multiplication the input errors are doubled before being added. Furthermore, errors are introduced when the values are rounded, so each operation adds 0.5 ulp to the error bound.

We also have a second group of lemmas. This group of lemmas consists of all the lemmas that we needed to prove the lemmas in the first group. When we construct a proof, we have a lot of freedom to decide when to split up a lemma into multiple lemmas to make the process easier and to make proofs more understandable. This is why we did not mention all these proofs in this chapter. Both the lemmas that are discussed and those that are not discussed can be found on Github [4]. We did show lemmas on how errors are introduced through rounding (9.1 - 9.3) and how errors propagate through addition (9.5) and multiplication (9.6 - 9.11).

As can be seen in 9.6 through 9.10, sometimes adding preconditions can improve the result. The difference between these lemmas is whether the exponents of the floating point numbers are 0 and the resulting accuracy. When the exponent is 0, the floating point number is subnormal. When we combine these lemmas into one lemma, we assume the worst case. That is why 9.11 has an error bound of twice the sum of the input errors. If we kept the lemmas separated, we could have gotten lemmas with tighter bounds for algorithms where we know what the exponents of the floating point numbers are.

9.2 Question 2

The second question was stated as "Is it possible to construct a proof for the error bound measured in ulps of a program that calculates a probability distribution over a Markov chain?".

The simple answer is: yes, this is possible. To be more specific, it is possible with an error bound of "$(\Sigma i = 1..k.(2 * \text{dim\_vec } bfs)\hat{\ } i)/4$". Here "dim_vec bfs" is the size of the vector and k is the amount of times the vector is multiplied with the matrix.

The first step to get there was to determine the error bound of the dot product of two vectors. For this, we used the following lemma.

assumes a_rel : "vec_all2 ($\lambda f\ r.$ ulp_accuracy r (f :: ($'e$, $'f$) float) $a\_accuracy$) (afs :: ($'e$, $'f$) float vec) ars"
and b_rel : "vec_all2 ($\lambda f\ r.$ ulp_accuracy r (f :: ($'e$, $'f$) float) $b\_accuracy$) (bfs :: ($'e$, $'f$) float vec) brs"
and $signs$: "vec_all ($\lambda f.$ sign $f = 0$) $afs \wedge$ vec_all ($\lambda f.$ sign $f = 0$) bfs"
and $probs$: "vec_all ($\lambda r. 0 \leq r \wedge r \leq 1$) $ars \wedge$ vec_all ($\lambda r. 0 \leq r \wedge r \leq 1$) brs"
and len_e : "$1 < LENGTH('e)$"
and lim_a : "$a\_accuracy \leq (2 :: \text{real})\hat{\ }LENGTH('f)$"
and lim_b : "$b\_accuracy \leq (2 :: \text{real})\hat{\ }LENGTH('f)$"
and len_f : "$1 < LENGTH('f)$"
and fin : "is_finite (float_vec_mul afs bfs)"
and dim : "dim_vec afs = dim_vec bfs"
shows "ulp_accuracy (real_vec_mul ars brs) (float_vec_mul afs bfs) $((2 * a\_accuracy + 2 * b\_accuracy + 0.5) *$ dim_vec afs) $\wedge$ sign (float_vec_mul afs bfs) = 0"

Lemma 9.15: Accuracy of a dot product between two vectors

We explained in Section 5.4.1 that "float_vec_mul" and "real_vec_mul" are the dot products for vectors of floating point values and real values respectively. This lemma shows that when the elements in the input vectors have an error bound of $a\_accuracy$ and $b\_accuracy$, then the resulting value has an error bound of "$(2 * a\_accuracy + 2 * b\_accuracy + 0.5) *$ dim_vec afs". The reason for this is that a single "fmul_add" operation has a resulting error bound of "$2 * a\_accuracy + 2 * b\_accuracy + c\_accuracy + 0.5$", and in a dot product, the amount of "fmul_add" operations will be the size of the input vectors. For each "fmul_add" operation, the error thus far can be expressed as $c\_accuracy$, so multiplying "$2 * a\_accuracy + 2 * b\_accuracy + 0.5$" by the dimension of the vector will be the total resulting error. This can be used to construct the lemma for the dot product between a vector and a matrix. This results in the following lemma.

assumes a_rel : "mat_all2 (λf r. ulp_accuracy r (f :: ($'e$, $'f$) float) $a\_accuracy$)
(afs :: ($'e$, $'f$) float vec) ars"
and b_rel : "vec_all2 (λf r. ulp_accuracy r (f :: ($'e$, $'f$) float) $b\_accuracy$)
(bfs :: ($'e$, $'f$) float vec) brs"
and $signs$: "mat_all (λf. sign $f = 0$) $afs \wedge$
vec_all (λf. sign $f = 0$) bfs"
and $probs$: "mat_all ($\lambda r.0 \leq r \wedge r \leq 1$) $ars \wedge$ vec_all ($\lambda r.0 \leq r \wedge r \leq 1$) brs"
and len_e : "$1 < LENGTH('e)$"
and lim_a : "$a\_accuracy \leq (2 :: \text{real})\hat{}LENGTH('f)$"
and lim_b : "$b\_accuracy \leq (2 :: \text{real})\hat{}LENGTH('f)$"
and len_f : "$1 < LENGTH('f)$"
and fin : "is_finite (float_mult_mat_vec afs bfs)"
and dim : "dim_col afs = dim_vec bfs"
shows "vec_all2 (λf r.ulp_accuracy r f
(($2 * a\_accuracy + 2 * b\_accuracy + 0.5$) $*$ dim_vec afs))
(float_mult_mat_vec afs bfs) (mult_mat_vec ars brs)$\wedge$
vec_all (λf. sign $f = 0$) (float_mult_mat_vec afs bfs)

Lemma 9.16: Accuracy of a dot product between a vector and a matrix

This lemma has the functions "float_mult_mat_vec" and "mult_mat_vec" which are the dot product between a vector and a matrix for floating point numbers and real numbers respectively. The output of a dot product between a vector and a matrix is simply a vector where each element is the dot product between the input vector and a row of the matrix. This is why we get the same error bound of "$(2 * a\_accuracy + 2 * b\_accuracy + 0.5) *$ dim_vec afs", but this time we use a "vec_all" function to indicate each element in the vector has this error bound.

The next step is to do this multiplication more than once. For that, we have the following lemma. This lemma is also shown in Section 6.2.

assumes a_rel : "mat_all2 ($\lambda f\ r.$ ulp_accuracy r (f :: ($'e$, $'f$) float) 0)
(afs :: ($'e$, $'f$) float mat) ars"
and b_rel : "vec_all2 ($\lambda f\ r.$ ulp_accuracy r (f :: ($'e$, $'f$) float) 0)
(bfs :: ($'e$, $'f$) float vec) brs"
and $signs$: "mat_all ($\lambda f.$ sign $f = 0$) $afs \wedge$
vec_all ($f.$ sign $f = 0$) bfs"
and $probs$: "mat_all ($\lambda r.\ 0 \leq r \wedge r \leq 1$) $ars \wedge$ vec_all ($\lambda r.\ 0 \leq r \wedge r \leq 1$) brs"
and len_e : "$1 < LENGTH('e)$"
and len_f : "$1 < LENGTH('f)$"
and lim : "$(\Sigma i = 1..k.(2 *$ dim_vec $bfs)\hat{}i)/4 \leq (2$ :: real$)\hat{}LENGTH('f)$"
and fin : "vec_all (is_finite)(((float_mult_mat_vec afs)^^k) bfs)"
and dim_1 : "dim_col afs = dim_vec bfs"
and dim_2 : "dim_col afs = dim_row afs"
and sum_1 : "mat_all_col (λv :: real vec. sum (($) v)
$\{0$ :: nat.. $<$ dim_vec $v\}$ = (1 :: real))(ars :: real mat)"
and sum_2 : "sum (($) ($brs$ :: real vec)) 0 :: nat.. $<$ dim_vec brs = (1 :: real)"
shows "vec_all2 ($\lambda f\ r.$ ulp_accuracy $r\ f$ (($\Sigma i = 1..k.(2 *$ dim_vec bfs$)\hat{}i)/4$))
(((float_mult_mat_vec afs)^^k) bfs) (((mult_mat_vec ars)^^k) brs) $\wedge$
vec_all ($\lambda f.$ sign $f = 0$) (((float_mult_mat_vec afs)^^k) bfs)"

LEMMA 9.17: Accuracy of a dot product between a vector and a matrix executed iteratively

Here a_rel and b_rel assume that we start off with an error of 0. The matrix is the same in each multiplication which tells us that "$(2 * a\_accuracy + 2 * b\_accuracy + 0.5) *$ dim_vec afs" from the previous lemma is "$(2 * b\_accuracy + 0.5) *$ dim_vec afs" since $a\_accuracy$ is always 0. Also, afs represents a row of the matrix afs and since we have the dim_1 and dim_2, we can replace "dim_vec afs" with "dim_vec bfs". After 1 multiplication, the error bound will be "$0.5*$dim_vec bfs". The next steps are "(dim_vec $bfs)^2 +$ $0.5*$dim_vec bfs", and "$2*$(dim_vec $bfs)^3 +$ (dim_vec $bfs)^2 + 0.5*$dim_vec bfs". Here we can see a pattern emerging. When we call the amount of multiplications between the vector and the matrix k, then the pattern is "$(\Sigma i = 1..k.(2 *$ dim_vec $bfs)\hat{}i)/4$".

9.3 Question 3

The third question was stated as "How tight are the error bounds constructed using these lemmas?".

During this research, we did not manage to provide an answer to this question. We do, however, have a reasoned guess for the answer. It is clear that "$(\Sigma i = 1..k.(2 *$ dim_vec $bfs)\hat{}i)/4$" grows very quickly and thus the error bound is not very useful if either the dimensions or the number of iterations are big. In practice, most errors will be a lot smaller. This suggests that the error bounds constructed using these lemmas are very loose. We know different reasons why this formula will be an overestimation.

9.3.1 Limits of this approach

The method that we use makes it possible to prove an error bound when statically analyzing the code. This means that we do not know all the input values which makes our bounds more loose.

The first reason for that is related to the rounding step. We assume the rounding step increases the error. This is not the only possibility since sometimes no error is introduced here. Another option is that the rounding operation gets the floating point number closer to the real number. This would lead to the error bound decreasing, but we cannot be sure of that and thus we always assume the worst-case scenario.

Secondly, we also assume the worst in the error propagation. For example, in the case of the addition of a and b with $a\_accuracy$ and $b\_accuracy$ where "$a\_accuracy >= b\_accuracy$". Then we know that for normal numbers the worst case will result in an accuracy of "$a\_accuracy + b\_accuracy/2$". However, when a and b are subnormal, the resulting accuracy might be "$a\_accuracy + b\_accuracy$". When you are statically analyzing code, you do not know if the numbers are normal or subnormal, so you have to assume the worst.

9.3.2 Limits of our proofs

Another reason for our bounds to be loose is that the proofs can simply be improved. If a and b are multiplied, then we have proven that the result has an error bound of "$2{*}a\_accuracy{+}2{*}b\_accuracy$". Assuming that "$a\_accuracy >= b\_accuracy$" we think that this bound can be improved to at least "$2 * a\_accuracy + b\_accuracy$". However, we have not formally proven this.

9.3.3 Limits of the application

We also think this method might work better in different applications. If the matrix represents a Markov chain, then there is a good chance that many values will be 0. When something is multiplied by 0, the result will be 0. When the input 0 has no error, this leads to the output error becoming 0. However, when statically analyzing code, you do not know when this is the case for the vertex. This leads us to increase the error bound even when we know the error is 0. The values in the matrix never change, so we might know how many elements in the matrix are 0. We think this information could improve the error bound, but we did not manage to include this in our proofs.

Furthermore, we observed it is possible to create a vector and matrix where an operation happens that results in a bad error bound, but that would often mean that other values become practically 0 or that you need a lot more nodes in the Markov chain that do practically nothing meaning that the "dim_vec bfs" in "$(\Sigma i = 1..k.(2{*}\text{dim\_vec } bfs)\hat{}i)/4$" is a big overestimation. To put it in other words, there will be a lot of operations where the error bound is not increasing at all. This might mean that the chosen application simply does not show the strength of this method.

The error bound we construct is highly influenced by the number of operations an algorithm has. Even if these operations have little influence, we did not manage to prove that they did not have a big influence. When we calculate a probability distribution of a Markov chain, there simply are a lot of operations meaning our error bound gets big.

9.3.4 Other Limits

We are confident that we could have proven a tighter bound if we also had proven that the floating point numbers always stayed between 0 and 1. Sadly we found a counterexample, and thus we did not manage to prove this tighter bound using this information. The counterexample we found had a very simple matrix as can be seen below, but the vector is more complicated. First, it is useful to define 2 values.

$$v1 = float('0011111111101111111111111111111111111111111111111111111111111010')$$
$$v2 = float('0011110010101000000000000000000000000000000000000000000000000000')$$

These are 64-bit floating point numbers, meaning they contain 1 sign bit, 11 exponent bits, and 52 fraction bits. Therefore, the numbers would have the following value:

$$v1 = 2^{-1} * (1 + \frac{2^{52} - 6}{2^{52}})$$

$$v2 = 2^{-53} * (1 + \frac{2^{51}}{2^{52}})$$

or

$$v2 = 2^{-53} * (1 + \frac{1}{2})$$

We can use these values to get a value greater than 1 when we calculate the multiplication using 64-bit floating point numbers.

$$\begin{bmatrix} v1 \\ v2 \\ v2 \\ v2 \\ v2 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 + \frac{1}{2^{52}} \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

This might be a small difference, but it leads us to believe we can construct vertices and matrices which get bigger errors.

Chapter 10

Conclusions and Future Work

As we said in Section 9.2, we have proven an error bound for iterative matrix multiplication. The bound is measured in ulps can be calculated using the formula:

$$(\Sigma i = 1..k.(2 * \text{dim\_vec } bfs)\hat{} i)/4$$

The dimensions of the vector and matrix are the same and are represented with "dim_vec bfs". The amount of iterations for the multiplication is represented with k.

In Section 2.1 and Section 2.2 we described the downsides of alternative methods. To be more specific, we described that using an absolute bound would be difficult to use in static analysis and that using a relative bound would be difficult to use in chained multiplications. Our method of using ulps does not have these downsides. Furthermore, in Section 3.1, we described that we can use ulps for both normal and subnormal numbers which is also a strength of this method.

We got to the formula for the error bound by proving that for addition the input errors are added, and for multiplication, the errors are added and then multiplied by 2. Furthermore, we have proven that the error introduced for rounding is 0.5 ulp. This would mean that if each element in the input vector would have an error of e_k, then the error for the output, $e_{(k+1)}$, can easily be calculated. We start by realizing that each multiplication doubles the input error, and thus we know that for each of these multiplications, the error becomes "$2 * e_k + 2 * 0 = 2 * e_k$". The 0 here is the error from the element in the matrix. Each operation also has a rounding step meaning it becomes "$2 * e_k + 0.5$". Next, we look at how many operations happen for a single entry in the output vector. This number is the size of the input vector and thus "dim_vec bfs". Since the addition only adds all errors, the output error will be a summation of this for "dim_vec bfs" times, so it will be "dim_vec $bfs * (2 * e_k + 0.5)$". Note here that it is "$2 * e_k + 0.5$" instead of "$2 * e_k + 1$" since the "fmul_add" operation does the multiplication and addition together with only rounding once. We have proven that this pattern of "$e_{(k+1)} = \text{dim\_vec } bfs * (2 * e_k + 0.5)$" results in "$e_k = (\Sigma i = 1..k.(2 * \text{dim\_vec } bfs)\hat{} i)/4$" when "$e_0 = 0$".

We think that this is a loose bound which can be improved significantly. The first method we propose is to use the fact that errors do not change if the corresponding value in the matrix is 0. Secondly, we think that the tightness for error propagation through multiplication can be improved and this might influence the error bound in the output vector. Currently, when a and b are multiplied with accuracy's $a\_accuracy$ and $b\_accuracy$, the result will have an accuracy of "$2 * a\_accuracy + 2 * b\_accuracy$". We know that $a\_accuracy$ and $b\_accuracy$ can double, but not at the same time. The last improvement that we could think of is ignoring subnormal numbers. Subnormal numbers are very small so a conclusion where only a claim is made about the elements which are normal numbers might still be useful. We are confident the error propagation can be a lot tighter for addition if only normal numbers are used so this too might influence the error bound in the output vector significantly. When a and b are added, then we have proven the result will have an accuracy of "$a\_accuracy + b\_accuracy$". If a is bigger than b, we should be able to prove an accuracy of "$a\_accuracy + b\_accuracy/2$".

Lastly, we think that this is not a good method for error estimation in matrix multiplication. We realized it is not difficult to construct a series of operations to increase the error significantly, however, when creating a vector and a matrix to get these operations in the correct order, many of the values in the vector will often be 0 or small enough to have practically no influence after 1 step. This means that the number of operations that increase the error is far less than the total number of operations. This, in turn, leads to us overestimating the error when we use our formula.

Next to the goals we set, another goal could have been to integrate an external tool into Isabelle/HOL as a tactic, just like Boldo et al. [8] did for Gappa and Coq. We did not do this because the specific use case of calculating a probability distribution over a Markov chain that sparked the interest seemed difficult to prove in such an external tool. This, however, does not mean it would have no use and such an integration could be future work since it would still make it easier to prove lemmas regarding floating point numbers in Isabelle.

Based on these conclusions, we propose 4 areas for future work.

- Investigate if the error bound can be improved using amongst others our suggested methods.
- Investigate whether iterative matrix multiplication is indeed a bad example because the edge cases can never occur.
- Investigate if this method can be applied in different algorithms and decide whether this method can give useful results.
- Integrate an external tool like Gappa into Isabelle/HOL.

Bibliography

[1] The vancouver stock exchange sources. URL: https://www5.in.tum.de/~huckle/Vancouv.pdf.

[2] *IEEE standard for floating-point arithmetic*, Jul 2019. doi:10.1109/ieeestd.2019.8766229.

[3] Christel Baier and Joost-Pieter Katoen. *Principles Of Model Checking*, volume 950, chapter 10.1. 2008.

[4] Jan Douwe Beekman. Materthesisisabelle. *GitHub repository*, 2024. https://github.com/Jodopro/MasterThesisIsabelle.

[5] Guillaume Bertholon, Érik Martin-Dorel, and Pierre Roux. Primitive floats in coq. *Leibniz International Proceedings in Informatics*, 141, 2019. doi:10.4230/LIPIcs.ITP.2019.7.

[6] Sylvie Boldo. Computing a correct and tight rounding error bound using rounding-to-nearest. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10152, 2017. doi:10.1007/978-3-319-54292-8_4.

[7] Sylvie Boldo and Marc Daumas. A simple test qualifying the accuracy of horner's rule for polynomials. *Numerical Algorithms*, 37, 2004. doi:10.1023/B:NUMA.0000049487.98618.61.

[8] Sylvie Boldo, Jean Christophe Filliâtre, and Guillaume Melquiond. Combining coq and gappa for certifying floating-point programs. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5625, 2009. doi:10.1007/978-3-642-02614-0_10.

[9] Sylvie Boldo and Guillaume Melquiond. Flocq: A unified library for proving floating-point algorithms in coq. *Proceedings - Symposium on Computer Arithmetic*, 2011. doi:10.1109/ARITH.2011.40.

[10] Sylvie Boldo and Guillaume Melquiond. Some formal tools for computer arithmetic: Flocq and gappa. *Proceedings - Symposium on Computer Arithmetic*, 2021-June, 2021. doi:10.1109/ARITH51176.2021.00031.

[11] Sylvie Boldo and César Muñoz. Provably faithful evaluation of polynomials. *Proceedings of the ACM Symposium on Applied Computing*, 2, 2006. doi:10.1145/1141277.1141586.

[12] Marc Daumas and Guillaume Melquiond. Certification of bounds on expressions involving rounded operators. *ACM Transactions on Mathematical Software*, 37, 2010. doi:10.1145/1644001.1644003.

[13] Marc Daumas, Laurence Rideau, and Laurent Théry. A generic library for floating-point numbers and its application to exact computing. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2152, 2001. doi:10.1007/3-540-44755-5_13.

[14] James Demmel and Yozo Hida. Fast and accurate floating point summation with application to computational geometry. *Numerical Algorithms*, 37, 2004. doi:10.1023/B:NUMA.0000049458.99541.38.

[15] Florent De Dinechin, Christoph Lauter, and Guillaume Melquiond. Certifying the floating-point implementation of an elementary function using gappa. *IEEE Transactions on Computers*, 60, 2011. doi:10.1109/TC.2010.128.

[16] John Harrison. A machine-checked theory of floating point arithmetic. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1690, 1999. doi:10.1007/3-540-48256-3_9.

[17] Fabian Immler and Johannes Hölzl. Numerical analysis of ordinary differential equations in isabelle/hol. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7406, 2012. doi:10.1007/978-3-642-32347-8_26.

[18] Daisuke Ishii and Tomohito Yabu. Computer-assisted verification of four interval arithmetic operators. *Journal of Computational and Applied Mathematics*, 377, 2020. doi:10.1016/j.cam.2020.112893.

[19] Ariel E. Kellison and Andrew W. Appel. Verified numerical methods for ordinary differential equations. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 13466, 2022. doi:10.1007/978-3-031-21222-2_9.

[20] Peter Lammich. Refinement to imperative/hol. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9236, 2015. doi:10.1007/978-3-319-22102-1_17.

[21] Victor Magron, George Constantinides, and Alastair Donaldson. Certified roundoff error bounds using semidefinite programming. *ACM Transactions on Mathematical Software*, 43, 2017. doi:10.1145/3015465.

[22] Guillaume Melquiond. Proving bounds on real-valued functions with computations. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5195, 2008. doi:10.1007/978-3-540-71070-7_2.

[23] Guillaume Melquiond. Floating-point arithmetic in the coq system. *Information and Computation*, 216, 2012. doi:10.1016/j.ic.2011.09.005.

[24] Anane Mohamed, Anane Nadjia, Bessalah Hamid, and Issad Mohamed. Reconfigurable architecture for elementary functions evaluation. *2009 IEEE/ACS International Conference on Computer Systems and Applications, AICCSA 2009*, 2009. doi:10.1109/AICCSA.2009.5069429.

[25] Jean Michel Muller and Laurence Rideau. Formalization of double-word arithmetic, and comments on "tight and rigorous error bounds for basic building blocks of double-word arithmetic. *ACM Transactions on Mathematical Software*, 48, 2022. doi:10.1145/3484514.

[26] Peter G. Neumann. Rounding error changes parliament makeup. *The Risks Digest*, Apr 1992. URL: http://catless.ncl.ac.uk/Risks/13/37#subj4.

[27] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/Hol: A Proof Assistant for higher-order logic.* Springer, 2002.

[28] Christine Paulin-Mohring. Introduction to the coq proof-assistant for practical software verification. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7682, 2012. doi:10.1007/978-3-642-35746-6_3.

[29] Pierre Roux. Formal proofs of rounding error bounds: With application to an automatic positive definiteness check. *Journal of Automated Reasoning*, 57, 2016. doi:10.1007/s10817-015-9339-z.

[30] Michael J. Schulte and Earl E. Swartzlander. Hardware designs for exactly rounded elementary functions. *IEEE Transactions on Computers*, 43, 1994. doi:10.1109/12.295858.

[31] René Thiemann and Akihisa Yamada. Matrices, jordan normal forms, and spectral radius theory. *Archive of Formal Proofs*, August 2015. https://isa-afp.org/entries/Jordan_Normal_Form.html, Formal proof development.

[32] Tao Yao, Deyuan Gao, Xiaoya Fan, and Jari Nurmi. Correctly rounded architectures for floating-point multi-operand addition and dot-product computation. *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors*, 2013. doi:10.1109/ASAP.2013.6567600.

[33] Lei Yu. A formal model of ieee floating point arithmetic. *Archive of Formal Proofs*, July 2013. https://isa-afp.org/entries/IEEE_Floating_Point.html, Formal proof development.

[34] Érik Martin-Dorel and Guillaume Melquiond. Proving tight bounds on univariate expressions with elementary functions in coq. *Journal of Automated Reasoning*, 57, 2016. doi:10.1007/s10817-015-9350-4.