CR-LIBM
A library of correctly rounded elementary functions in double-precision

Catherine Daramy-Loirat, David Defour, Florent de Dinechin,
Matthieu Gallet, Nicolas Gast, Christoph Quirin Lauter, Jean-Michel Muller

February 10, 2017

Important warning

This report describes and proves version 1.0beta4 of the `crlibm` library. It may therefore not correspond to the latest version. An up-to-date version will always be distributed along with the code.

Many thanks to...

- Vincent Lefèvre, from INRIA-Lorraine, without whom this project would't have started, and who has since then lended a helpful hand in more than 17 occasions;
- The Arénaire project at ENS-Lyon, especially Marc Daumas, Sylvie Boldo, Guillaume Melquiond, Nathalie Revol and Arnaud Tisserand;
- Guillaume Melquiond for Gappa and its first-class hotline;
- The MPFR developers, and especially the INRIA SPACES project;
- The Intel Nizhniy-Novgorod Lab, especially Andrey Naraikin, Sergei Maidanov and Evgeny Gvozdev;
- The contributors of bits and pieces, Phil Defert and Eric McIntosh from CERN, Patrick Pelissier from INRIA;
- All the people who have reported bugs, hoping that they continue until they have to stop: Evgeny Gvozdev from Intel with special thanks, Christoph Lauter from TUMünchen (before he joined the project), Eric McIntosh from CERN, Patrick Pelissier and Paul Zimmermann from INRIA Lorraine;
- William Kahan at UCBerkeley, Peter Markstein at HP, Neil Toda at Sun, Shane Story at Intel, and many others for interesting and sometimes heated discussions;
- Serge Torres for LIPForge.

This work was partly funded by the INRIA, the ENS-Lyon and the Université de Perpignan.

Contents

Chapter 0

Getting started with crlibm

0.1 What is crlibm?

The crlibm project aims at developing a portable, proven, correctly rounded, and efficient mathematical library (libm) for double precision.

correctly rounded Current libm implementation do not always return the floating-point number that is closest to the exact mathematical result. As a consequence, different libm implementation will return different results for the same input, which prevents full portability of floating-point applications. In addition, few libraries support but the round-to-nearest mode of the IEEE754/IEC 60559 standard for floating-point arithmetic (hereafter usually referred to as the IEEE-754 standard). crlibm provides the four rounding modes: To nearest, to $+\infty$, to $-\infty$ and to zero.

portable crlibm is written in C and will be compiled by any compiler fulfilling basic requirements of the ISO/IEC 9899:1999 (hereafter referred to as C99) standard. This is the case of gcc version 3 and higher which is available on most computer systems. It also requires a floating-point implementation respecting the IEEE-754 standard, which is also available on most modern systems. crlibm has been tested on a large range of systems.

proven Other libraries attempt to provide correctly-rounded result. For theoretical and practical reasons, this behaviour is difficult to prove, and in extreme cases termination is not even guaranteed. crlibm intends to provide a comprehensive proof of the theoretical possibility of correct rounding, the algorithms used, and the implementation, assuming C99 and IEEE-754 compliance.

efficient performance and resource usage of crlibm should be comparable to existing libm implementations, both in average and in the worst case. In contrast, other correctly-rounded libraries have worst case performance and memory consumption several order of magnitude larger than standard libms.

The ultimate goal of the crlibm project is to push towards the standardization of correctly-rounded elementary functions.

0.2 Compilation and installation

See the INSTALL file in the main directory. This library is developed using the GNU autotools, and can therefore be compiled on most Unix-like systems by `./configure; make`.

The command `make check` will launch the selftest. For more advanced testing you will need to have MPFR installed (see www.mpfr.org) and to pass the `--enable-mpfr` flag to `configure`. For other flags, see `./configure --help` .

0.3 Using `crlibm` functions in your program

Currently `crlibm` functions have different names from the standard `math.h` functions. For example, for the sine function (`double sin(double)` in the standard `math.h`), you have four different functions in `crlibm` for the four different rounding modes. These functions are named `sin_rn`, `sin_ru`, `sin_rd` and `sin_rz` for round to the nearest, round up, round down and round to zero respectively. These functions are declared in the C header file `crlibm.h`.

The `crlibm` library relies on double-precision IEEE-754 compliant floating-point operations. For some processors and some operating systems (most notably IA32 and IA64 processors under GNU/Linux), the default precision is set to double-extended. On such systems you will need to call the `crlibm_init()` function before using any `crlibm` function to ensure such compliance. This has the effect of setting the processor flags to IEEE-754 double-precision with rounding to the nearest mode. This function returns the previous processor status, so that previous mode can be restored using the function `crlibm_exit()`. Note that you probably only need one call to `crlibm_init()` at the beginning of your program, not one call before each call to a mathematical function.

Here is a non-exhaustive list of systems on which `crlibm_init()` is NOT needed, and which can therefore use `crlibm` as a transparent replacement of the standard `libm`:

- Most Power/PowerPC based systems, including those from Apple or from IBM;
- All the 64-bit Linux versions: the reason is that all x86-compatible processors (by AMD and Intel) supporting 64-bit addressing also feature SSE2 FP instructions, which are cleaner and more efficient than the legacy x87 FPU. On such systems, SSE2 is therefore used by default by `gcc` for double-precision FP computing.
- On recent 32-bit x86 processors also featuring SSE2 extensions (including pentium 4 and later, and generally most processors produced after 2005), you can try to force the use of SSE2 instructions using `configure --enable-sse2`. Beware, the code produced will not run on older hardware.

Here's an example function named `compare.c` using the cosine function from `crlibm` library.

Listing 1: compare.c

```
#include<stdio.h>
#include<math.h>
#include<crlibm.h>

int main(void){
  double x, res_libm, res_crlibm;

  printf("Enter a floating point number: ");
  scanf("%lf", &x);
  res_libm = cos(x);
  crlibm_init(); /* no need here to save the old processor state returned by crlibm_init() */
  res_crlibm = cos_rn(x);
  printf("\n x=%.25e \n", x);
  printf("\n cos(x) with the system : %.25e \n", res_libm);
  printf("\n cos(x) with crlibm     : %.25e \n", res_crlibm);
  return 0;
}
```

This example will be compiled with `gcc compare.c -lm -lcrlibm -o compare`

0.4 Currently available functions

The currently available functions are summarized in Table 1.

Here are some comments on this table:

- Every function takes a double-precision number and returns a double-precision number.
- For trigonometric functions the angles are expressed in radian.
- The two last columns describe the state of the proof:

C99	crlibm name				State of the proof	
	to nearest	to $+\infty$	to $-\infty$	to zero	Worst cases	Proof of the code
exp	exp_rn	exp_ru	exp_rd	exp_rz	complete	complete (formal)
expm1	expm1_rn	expm1_ru	expm1_rd	expm1_rz	complete	partial
log	log_rn	log_ru	log_rd	log_rz	complete	complete
log1p	log1p_rn	log1p_ru	log1p_rd	log1p_rz	complete	partial
log2	log2_rn	log2_ru	log2_rd	log2_rz	complete	partial
log10	log10_rn	log10_ru	log10_rd	log10_rz	complete	partial
sin	sin_rn	sin_ru	sin_rd	sin_rz	$[-\pi, \pi]$	complete (paper+formal)
cos	cos_rn	cos_ru	cos_rd	cos_rz	$[-\pi/2, \pi/2]$	complete (paper+formal)
tan	tan_rn	tan_ru	tan_rd	tan_rz	$[-\pi/2, \pi/2]$	complete (paper+formal)
asin	asin_rn	asin_ru	asin_rd	asin_rz	complete	partial
acos	acos_rn	acos_ru	acos_rd	acos_rz	complete	partial
atan	atan_rn	atan_ru	atan_rd	atan_rz	complete	complete (paper)
sinh	sinh_rn	sinh_ru	sinh_rd	sinh_rz	complete	complete (paper)
cosh	cosh_rn	cosh_ru	cosh_rd	cosh_rz	complete	complete (paper)
sinpi	sinpi_rn	sinpi_ru	sinpi_rd	sinpi_rz	complete	complete (formal)
cospi	cospi_rn	cospi_ru	cospi_rd	cospi_rz	complete	complete (formal)
tanpi	tanpi_rn	tanpi_ru	tanpi_rd	tanpi_rz	$[2^{-25}, 2^{-5}]$	complete (formal)
atanpi	atanpi_rn	atanpi_ru	atanpi_rd	atanpi_rz	$[\tan(2^{-25}\pi), \tan(2^{-5}\pi)]$	complete (paper)
pow	pow_rn				see chapter 15	see chapter 15

Table 1: Current state of crlibm.

- The first indicates the state of the search for worst cases for correct rounding [26, 27]. If it indicates "complete", it means that the function is guaranteed to return correct rounding on its whole floating-point input range. Otherwise, it mentions the interval on which the function is guaranteed to return correct rounding. Note that crlibm is designed in such a way that there is a very high probability that it is correctly rounded everywhere, however this is not yet proven formally. This question is explained in details in section 1.3.
- The second indicates the state of the proof of the code itself. Some (older) functions have a lengthy paper proof in this document, some other have a partial or complete formal proof using the Gappa proof assistant [31, 13].

0.5 Writing portable floating-point programs

Here are some rules to help you design programs which have to produce exactly the same results on different architectures and different operating systems.

- Try to use the same compiler on all the systems.
- Demand C99 compliance (pass the `-C99`, `-std=c99`, or similar flag to the compiler). For Fortran, demand F90 compliance.
- Call `crlibm_init()` before you begin floating-point computation. This ensures that the computations will all be done in IEEE-754 double-precision with round to nearest mode, which is the largest precision well supported by most systems. On IA32 processors, problems may still occur for extremely large or extremely small values.

- Do not hesitate to rely heavily on parentheses (the compiler should respect them according to the standards, although of course some won't). Many times, wondering where the parentheses should go in an expression like `a+b+c+d` will even help you improve the accuracy of your code.
- Use `crlibm` functions in place of `math.h` functions.

Chapter 1

Introduction: Goals and methods

1.1 Correct rounding and elementary functions

The need for accurate elementary functions is important in many critical programs. Methods for computing these functions include table-based methods[17, 38], polynomial approximations and mixed methods[9]. See the books by Muller[34] or Markstein[30] for recent surveys on the subject.

The IEEE-754 standard for floating-point arithmetic[5] defines the usual floating-point formats (single and double precision). It also specifies the behavior of the four basic operators ($+, -, \times, \div$) and the square root in four rounding modes (to the nearest, towards $+\infty$, towards $-\infty$ and towards 0). Its adoption and widespread use have increased the numerical quality of, and confidence in floating-point code. In particular, it has improved *portability* of such code and allowed construction of *proofs* on its numerical behavior. Directed rounding modes (towards $+\infty$, $-\infty$ and 0) also enabled efficient *interval arithmetic*[32, 21].

However, the IEEE-754 standard specifies nothing about elementary functions, which limits these advances to code excluding such functions. Currently, several options exist: on one hand, one can use today's mathematical libraries that are efficient but without any warranty on the correctness of the results. To be fair, most modern libraries are *accurate-faithful*: trying to round to nearest, they return a number that is one of the two FP numbers surrounding the exact mathematical result, and indeed return the correctly rounded result most of the time. This behaviour is sometimes described using phrases like *99% correct rounding* or *0.501 ulp accuracy*.

When stricter guarantees are needed, some multiple-precision packages like MPFR [33] offer correct rounding in all rounding modes, but are several orders of magnitude slower than the usual mathematical libraries for the same precision. Finally, there are are currently three attempts to develop a correctly-rounded `libm`. The first was IBM's `libultim`[29] which is both portable and fast, if bulky, but lacks directed rounding modes needed for interval arithmetic. The second was Arénaire's `crlibm`, which was first distributed in 2003. The third is Sun correctly-rounded mathematical library called `libmcr`, whose first beta version appeared in 2004. These libraries are reviewed in 1.4.

The goal of the `crlibm` project is to build on a combination of several recent theoretical and algorithmic advances to design a proven correctly rounded mathematical library, with an overhead in terms of performance and resources acceptable enough to replace existing libraries transparently.

More generally, the `crlibm` project serves as an open-source framework for research on software elementary functions. As a side effect, it may be used as a tutorial on elementary function development.

1.2 A methodology for efficient correctly-rounded functions

1.2.1 The Table Maker's Dilemma

With a few exceptions, the image $\hat{y}$ of a floating-point number x by a transcendental function f is a transcendental number, and can therefore not be represented exactly in standard numeration systems. The only hope is to compute the floating-point number that is closest to (resp. immediately above or

immediately below) the mathematical value, which we call the result *correctly rounded* to the nearest (resp. towards $+\infty$ or towards $-\infty$).

It is only possible to compute an approximation y to the real number $\hat{y}$ with precision $\bar{\varepsilon}$. This ensures that the real value $\hat{y}$ belongs to the interval $[y(1-\bar{\varepsilon}), y(1+\bar{\varepsilon})]$. Sometimes however, this information is not enough to decide correct rounding. For example, if $[y(1-\bar{\varepsilon}), y(1+\bar{\varepsilon})]$ contains the middle of two consecutive floating-point numbers, it is impossible to decide which of these two numbers is the correctly rounded to the nearest of $\hat{y}$. This is known as the Table Maker's Dilemma (TMD). For example, if we consider a numeration system in radix 2 with n-bit mantissa floating point number and m the number of significant bit in y such that $\bar{\varepsilon} \leq 2^m$, then the TMD occurs:

- for rounding toward $+\infty$, $-\infty$, 0, when the result is of the form:

$$\underbrace{1.\underbrace{xxx...xx}_{n\ bits}\overbrace{111111...11}^{m\ bits}}\ xxx...$$

  or:

$$\underbrace{1.\underbrace{xxx...xx}_{n\ bits}\overbrace{000000...00}^{m\ bits}}\ xxx...$$

- for rounding to nearest, when the result is of the form:

$$\underbrace{1.\underbrace{xxx...xx}_{n\ bits}\overbrace{011111...11}^{m\ bits}}\ xxx...$$

  or :

$$\underbrace{1.\underbrace{xxx...xx}_{n\ bits}\overbrace{100000...00}^{m\ bits}}\ xxx...$$

1.2.2 The onion peeling strategy

A method described by Ziv [40] is to increase the precision $\bar{\varepsilon}$ of the approximation until the correctly rounded value can be decided. Given a function f and an argument x, the value of $f(x)$ is first evaluated using a quick approximation of precision $\bar{\varepsilon}_1$. Knowing $\bar{\varepsilon}_1$, it is possible to decide if rounding is possible, or if more precision is required, in which case the computation is restarted using a slower approximation of precision $\bar{\varepsilon}_2$ greater than $\bar{\varepsilon}_1$, and so on. This approach makes sense even in terms of average performance, as the slower steps are rarely taken.

However there was until recently no practical bound on the termination time of such an algorithm. This iteration has been proven to terminate, but the actual maximal precision required in the worst case is unknown. This might prevent using this method in critical application.

1.3 The Correctly Rounded Mathematical Library

Our own library, called `crlibm` for *correctly rounded mathematical library*, is based on the work of Lefèvre and Muller [26, 27] who computed the worst-case $\bar{\varepsilon}$ required for correctly rounding several functions in double-precision over selected intervals in the four IEEE-754 rounding modes. For example, they proved that 157 bits are enough to ensure correct rounding of the exponential function on all of its domain for the four IEEE-754 rounding modes.

1.3.1 Two steps are enough

Thanks to such results, we are able to guarantee correct rounding in two iterations only, which we may then optimize separately. The first of these iterations is relatively fast and provides between 60 and 80 bits of accuracy (depending on the function), which is sufficient in most cases. It will be referred

throughout this document as the quick phase of the algorithm. The second phase, referred to as the accurate phase, is dedicated to challenging cases. It is slower but has a reasonably bounded execution time, tightly targeted at Lefèvre's worst cases.

Having a proven worst-case execution time lifts the last obstacle to a generalization of correctly rounded transcendentals. Besides, having only two steps allows us to publish, along with each function, a proof of its correctly rounding behavior.

1.3.2 Portable IEEE-754 FP for a fast first step

The computation of a tight bound on the approximation error of the first step ($\bar{\varepsilon}_1$) is crucial for the efficiency of the onion peeling strategy: overestimating $\bar{\varepsilon}_1$ means going more often than needed through the second step, as will be detailed below in 1.3.6. As we want the proof to be portable as well as the code, our first steps are written in strict IEEE-754 arithmetic. On some systems, this means preventing the compiler/processor combination to use advanced floating-point features such as fused multiply-and-add or extended double precision. It also means that the performance of our portable library will be lower than optimized libraries using these features (see [12] for recent research on processor-specific correct-rounding).

To ease these proofs, our first steps make wide use of classical, well proven results like Sterbenz' lemma or other floating-point theorems. When a result is needed in a precision higher than double precision (as is the case of y_1, the result of the first step), it is represented as as the sum of two floating-point numbers, also called a *double-double* number. There are well-known algorithms for computing on double-doubles, and they are presented in the next chapter. An advantage of properly encapsulating double-double arithmetic is that we can actually exploit fused multiply-and-add operators in a transparent manner (this experimental feature is currently available for the Itanium and PowerPC platforms, when using the `gcc` compiler).

At the end of the quick phase, a sequence of simple tests on y_1 knowing $\bar{\varepsilon}_1$ allows to decide whether to go for the second step. The sequence corresponding to each rounding mode is shared by most functions and is also carefully proven in the next chapter.

1.3.3 Ad-hoc, fast multiple precision for accurate second step

For the second step, we may use two specific multiple-precision libraries:

- We first designed an ad-hoc multiple-precision library called Software Carry-Save library *(scslib)* which is lighter and faster than other available libraries for this specific application [14, 10]. This choice is motivated by considerations of code size and performance, but also by the need to be independent of other libraries: Again, we need a library on which we may rely at the proof level. This library is included in `crlibm`, but also distributed separately [2]. This library is described in more details in 2.8.

- More recently, we have been using redundant triple-double arithmetic for the second step. This approach is lighter, about ten times faster, and has the advantage of making it easier to reuse information from the fast step in the accurate one. The drawback is that it is more difficult to master. The basic triple-double procedures, and associated usage theorems, are described in a separate document (`tripledoubleprocedures.pdf`) also available in this distribution.

1.3.4 Relaxing portability

The `crlibm` framework has been used to study the performance advantage of using double-extended (DE) arithmetic when available. More specifically, the first case may be implemented fully in DE precision, and the second step may be implemented fully in double-DE arithmetic. Experiments have been performed on the logarithm and arctangent functions [12]. On some systems (mostly Linux on an IA32 processor) the logarithm will by default use this technology.

Another useful, non-portable hardware feature is the fused multiply-and-add available on Power/PowerPC and Itanium. The `crlibm` code does its best to use it when available.

1.3.5 Proving the correct rounding property

Throughout this document, what we call "proving" a function mostly means proving a tight bound on the total relative error of our evaluation scheme. The actual proof of the correct rounding property is then dependent on the availability of an actual worst-case accuracy for correctly rounding this function, as computed by Lefèvre and Muller. Three cases may happen:

- The worst case have been computed over the whole domain of the function. In this case the correct rounding property for this function is fully proven. The state of this search for worst cases is described in Table 1 page 9.
- The worst cases have been computed only over a subdomain of the function. Then the correct rounding property is proven on this subdomain. Outside of this domain `crlibm` offers "astronomical confidence" that the function is correctly rounded: to the best of current knowledge [18, 12], the probability of the existence of a misrounded value in the function's domain is lower than 2^{-40}. This is the case of the trigonometric functions, for instance. The actual domain on which the proof is complete is mentionned in the respective chapter of this document, and summed up in Table 1.
- The search for worst cases hasn't begun yet.

We acknowledge that the notion of astronomical confidence breaks the objective of guaranteed correct rounding, and we sidestep this problem by publishing along with the library (in this document) the domain of full confidence, which will only expand as time goes. Such behaviour has been proposed as a standard in [15]. The main advantage of this approach is that it ensures bounded and consistent worst-case execution time (within a factor 100 of that of the best available faithful `libms`), which we believe is crucial to the generalization of correctly rounded functions.

The alternative to our approach would be to implement a multi-layer onion-peeling strategy, as do GNU MPFR and Sun's `libmcr`. There are however many drawbacks to this approach, too:

- One may argue that, due to the finite nature of computers, it only pushes the bounds of astronomy a little bit further.
- The multilayer approach is only proven to terminate on elementary functions: the termination proof needs a theorem stating for example that the image of a rational by the function (with some well-known exceptions) will not be a rational. For other library functions like special functions, we have no such theorem. For these functions, we prefer take the risk of a misrounded value than the risk of an infinite loop.
- Similarly, the multilayer approach has potentially unbounded execution time and memory consumption which make it unsuitable for real-time or safety-critical applications, whereas crlibm will only be unsuitable if the safety depends on correct rounding, which is much less likely.
- Multilayer code is probably much more complex and error prone. One important problem is that it contains code that, according all probabilities, will never be run. Therefore, testing this code can not be done on the final production executable, but on a different executable in which previous layers have been disabled. This introduces the possibility of undetected bugs in the final production executable.

In the future, we will add, to those `crlibm` functions for which the required worst-case accuracy is unknown, a misround detection test at the end of the second step. This test will either print out on standard error a lengthy warning inviting to report this case, or launch MPFR computation, depending on a compilation switch.

1.3.6 Error analysis and the accuracy/performance tradeoff

As there are two steps on the evaluation, the proof also usually consists of two parts. The code of the second, accurate step is usually very simple and straightforward:

- Performance is not that much of an issue, since this step is rarely taken.

- All the special cases have already been filtered by the first step.
- The `scslib` library provides an overkill of precision.

Therefore, the error analysis of the second step, which ultimately proves the correct rounding property, is not very difficult.

For the first step, however, things are more complicated:

- We have to handle special cases (infinities, NaNs, signed zeroes, over- and underflows).
- Performance is a primary concern, sometimes leading to "dirty tricks" obfuscating the code.
- We have to compute a *tight* error bound, as explained below.

Why do we need a tight error bound? Because the decision to launch the second step is taken by a *rounding test* depending on

- the approximation $y_h + y_l$ computed in the first step, and
- this error bound $\bar{\varepsilon}_1$, which is computed statically.

The various rounding tests are detailed and proven in 2.7. The important notion here is that *the probability of launching the second, slower step will be proportional to the error bound $\bar{\varepsilon}_1$ computed for the first step*.

This defines the main performance tradeoff one has to manage when designing a correctly-rounded function: The average evaluation time will be

$$T_{\text{avg}} = T_1 + p_2 T_2 \tag{1.1}$$

where T_1 and T_2 are the execution time of the first and second phase respectively (with $T_2 \approx 100 T_1$ in `crlibm`), and p_2 is the probability of launching the second phase (typically we aim at $p_2 = 1/1000$ so that the average cost of the second step is less than 10% of the total.

As p_2 is almost proportional to $\bar{\varepsilon}_1$, to minimise the average time, we have to

- balance T_1 and p_2: this is a performance/precision tradeoff (the faster the first step, the less accurate)
- and compute a tight bound on the overall error $\bar{\varepsilon}_1$.

Computing this tight bound is the most time-consuming part in the design of a correctly-rounded elementary function. The proof of the correct rounding property only needs a proven bound, but a loose bound will mean a larger p_2 than strictly required, which directly impacts average performance. Compare $p_2 = 1/1000$ and $p_2 = 1/500$ for $T_2 = 100 T_1$, for instance. As a consequence, when there are multiple computation paths in the algorithm, it makes sense to precompute different values of $\bar{\varepsilon}_1$ on these different paths (see for instance the arctangent and the logarithm).

Apart from these considerations, computing the errors is mostly textbook science. Care must be taken that only *absolute* error terms (noted δ) can be added, although some error terms (like the rounding error of an IEEE operation) are best expressed as *relative* (noted ε). Remark also that the error needed for the theorems in 2.7 is a *relative* error. Managing the relative and absolute error terms is very dependent on the function, and usually involves keeping upper and lower bounds on the values manipulated along with the error terms.

Error terms to consider are the following:

- approximation errors (minimax or Taylor),
- rounding error, which fall into two categories:
  - roundoff errors in values tabulated as doubles or double-doubles (with the exception of roundoff errors on the coefficient of a polynomial, which are counted in the appproximation error),
  - roundoff errors in IEEE-compliant operations.

1.4 An overview of other available mathematical libraries

Many high-quality mathematical libraries are freely available and have been a source of inspiration for this work.

Most mathematical libraries do not offer correct rounding. They can be classified as

- portable libraries assuming IEEE-754 arithmetic, like *fdlibm*, written by Sun[3];
- Processor-specific libraries, by Intel[20, 1] and HP[30, 28] among other.

Operating systems often include several mathematical libraries, some of which are derivatives of one of the previous.

To our knowledge, three libraries currently offer correct rounding:

- The *libultim* library, also called MathLib, is developed at IBM by Ziv and others [29]. It provides correct rounding, under the assumption that 800 bits are enough in all case. This approach suffers two weaknesses. The first is the absence of proof that 800 bits are enough: all there is is a very high probability. The second is that, as we will see in the sequel, for challenging cases, 800 bits are much of an overkill, which can increase the execution time up to 20,000 times a normal execution. This will prevent such a library from being used in real-time applications. Besides, to prevent this worst case from degrading average performance, there is usually some intermediate levels of precision in MathLib's elementary functions, which makes the code larger, more complex, and more difficult to prove (and indeed this library is scarcely documented).

  In addition this library provides correct rounding only to nearest. This is the most used rounding mode, but it might not be the most important as far as correct rounding is concerned: correct rounding provides a precision improvement over current mathematical libraries of only a fraction of a unit in the last place *(ulp)*. Conversely, the three other rounding modes are needed to guarantee intervals in interval arithmetic. Without correct rounding in these directed rounding modes, interval arithmetic looses up to one *ulp* of precision in each computation.

- *MPFR* is a multiprecision package safer than *libultilm* as it uses arbitrary multiprecision. It provides most of elementary functions for the four rounding modes defined by the IEEE-754 standard. However this library is not optimized for double precision arithmetic. In addition, as its exponent range is much wider than that of IEEE-754, the subtleties of subnormal numbers are difficult to handle properly using such a multiprecision package.

- The `libmcr` library, by K.C. Ng, Neil Toda and others at Sun Microsystems, had its first beta version published in december 2004. Its purpose is to be a reference implementation for correctly rounded functions in double precision. It has very clean code, offers arbitrary multiple precision unlike `libultim`, at the expense of slow performance (due to, for example dynamic allocation of memory). It offers the directed rounding modes, and rounds in the mode read from the processor status flag.

1.5 Various policies in `crlibm`

1.5.1 Naming the functions

Current `crlibm` doesn't by default replace your existing `libm`: the functions in `crlibm` have the C99 name, suffixed with `_rn`, `_ru`, `_rd`, and `_rz` for rounding to the nearest, up, down and to zero respectively. They require the processor to be in round to nearest mode. Starting with version 0.9 we should provide a compile-time flag which will overload the default `libm` functions with the crlibm ones with rounding to nearest.

It is interesting to compare this to the behaviour of Sun's library: First, Sun's `libmcr` provides only one function for each C99 function instead of four in `crlibm`, and rounds according to the processor's current mode. This is probably closer to the expected long-term behaviour of a correctly-rounded mathematical library, but with current processors it may have a tremendous impact on performance. Besides,

the notion of "current processor rounding mode" is no longer relevant on recent processors like the Itanium family, which have up to four different modes at the same time. A second feature of `libmcr` is that it overloads by default the system `libm`.

The policy implemented in current `crlibm` intends to provide best performance to the two classes of users who will be requiring correct rounding: Those who want predictible, portable behaviour of floating-point code, and those who implement interval arithmetic. Of course, we appreciate any feedback on this subject.

1.5.2 Policy concerning IEEE-754 flags

Currently, the `crlibm` functions try to raise the Overflow and Underflow flags properly. Raising the other flags (especially the Inexact flag) is possible but considered too costly for the expected use, and will usually not be implemented. We also appreciate feedback on this subject.

1.5.3 Policy concerning conflicts between correct rounding and expected mathematical properties

As remarked in [15], it may happen that the requirement of correct rounding conflicts with a basic mathematical property of the function, such as its domain and range. A typical example is the arctangent of a very large number which, rounded up, will be a number larger than $\pi/2$ (fortunately, $\circ(\pi/2) < \pi/2$). The policy that will be implemented in `crlibm` will be

- to give priority to the mathematical property in round to nearest mode (so as not to hurt the innocent user who may expect such a property to be respected), and
- to give priority to correct rounding in the directed rounding modes, in order to provide trustful bounds to interval arithmetic.

Again, this policy is open to discussion.

1.6 Organization of the source code

For recent functions implemented using triple-double arithmetic, both quick and accurate phase are provided in a single source file, *e.g.* `exp-td.c`.

For older functions using the SCS library, each function is implemented as two files, one with the `_accurate` suffix (for instance `trigo_accurate.c`), the other named with the `_fast` suffix (for instance `trigo_fast.c`).

The *software carry-save* multiple-precision library is contained in a subdirectory called `scs_lib`.

The common C routines that are detailed in Chapter 2 of this document are defined in `crlibm_private.c` and `crlibm_private.h`.

Many of the constants used in the C code have been computed thanks to Maple procedures which are contained in the `maple` subdirectory. Some of these procedures are explained in Chapter 2. For some functions, a Maple procedure mimicking the C code, and used for debugging or optimization purpose, is also available.

The code also includes programs to test the `crlibm` functions against MPFR, `libultim` or `libmcr`, in terms of correctness and performance. They are located in the `tests` directory.

Gappa proof scripts are located in the `gappa` directory.

Chapter 2

Common notations, theorems and procedures

2.1 Notations

The following notations will be used throughout this document:

- $+$, $-$ and $\times$ denote the usual mathematical operations.
- $\oplus$, $\ominus$ and $\otimes$ denote the corresponding floating-point operations in IEEE-754 double precision, in the IEEE-754 *round to nearest* mode.
- $\circ(x)$, $\triangle(x)$ and $\triangledown(x)$ denote the value of x rounded to the nearest, resp. rounded up and down.
- ε (usually with some index) denotes a relative error, δ denotes an absolute error. Upper bounds on the absolute value of these errors will be denoted $\overline{\varepsilon}$ and $\overline{\delta}$.
- $\overline{\varepsilon}_{-k}$ – with a negative index – represents an error e such that $|e| \leq 2^{-k}$.
- For a floating-point number x, the value of the least significant bit of its mantissa is classically denoted $\mathrm{ulp}(x)$.

2.2 Common C procedures for double-precision numbers

2.2.1 Sterbenz Lemma

Theorem 1 (Sterbenz Lemma [37, 19])**.** *If x and y are floating-point numbers, and if $y/2 \leq x \leq 2y$ then $x \ominus y$ is computed exactly, without any rounding error.*

2.2.2 Double-precision numbers in memory

A double precision floating-point number uses 64 bits. The unit of memory in most current architectures is a 32-bit word. The order in which the two 32 bits words of a double are stored in memory depends on the architecture. An architecture is said *Little Endian* if the lower part of the number is stored in memory at the smallest address; It is the case of the x86 processors. Conversely, an architecture with the high part of the number stored in memory at the smallest address is said *Big Endian*; It is the case of the PowerPC processors.

In `crlibm`, we extract the higher and lower parts of a double by using an union in memory: the type `db_number`. The following code extracts the upper and lower part from a double precision number x.

Listing 2.1: Extract upper and lower part of a double precision number x

```
1  /* HI and LO are defined automatically by autoconf/automake. */
2
3  db_number xx;
```

```
int x_hi, x_lo;
xx.d = x;
x_hi = xx.i[HI]
x_lo = xx.i[LO]
```

2.2.3 Conversion from floating-point to integer

Theorem 2 (Conversion floating-point to integer)**.** *The following algorithm, taken from [4], converts a double-precision floating-point number d into a 32-bit integer i with rounding to nearest mode.*

It works for all the doubles whose nearest integer fits on a 32-bit machine signed integer.

Listing 2.2: Conversion from FP to int

```
#define DOUBLE2INT(i, d)    \
  {double t=(d+6755399441055744.0); i=LO(t);}
```

This algorithm adds the constant $2^{52} + 2^{51}$ to the floating-point number to put the integer part of x, in the lower part of the floating-point number. We use $2^{52} + 2^{51}$ and not 2^{52}, because the value 2^{51} is used to contain possible carry propagations with negative numbers.

2.2.4 Conversion from floating-point to 64-bit integer

Theorem 3 (Conversion floating-point to a long long integer)**.** *The following algorithm, is derived from the previous.*

It works for any double whose nearest integer is smaller than $2^{51} - 1$.

Listing 2.3: Conversion from FP to long long int

```
#define DOUBLE2LONGINT(i, d)                                          \
  {                                                                   \
    db_number t;                                                      \
    t.d = (d+6755399441055744.0);                                     \
    if (d >= 0) /* sign extend */                                     \
      i = t.l & 0x0007FFFFFFFFFFFFLL;                                 \
    else                                                              \
      i = (t.l & 0x0007FFFFFFFFFFFFLL) | (0xFFF8000000000000LL);      \
  }
```

2.2.5 Methods to raise IEEE-754 flags

The IEEE standard imposes, in certain cases, to raise flags and/or exceptions for the 4 operators ($+$, $\times$, $\div$, $\sqrt{}$). Therefore, it is legitimate to require the same for elementary functions.

In ISO C99, the following instructions raise exceptions and flags:

- **underflow** : the multiplication $\pm smallest \times smallest$ where *smallest* correspond to the smallest subnormal number,
- **overflow** : the multiplication $\pm largest \times largest$ where *largest* correspond to the largest normalized number,
- **division by zero** : the division $\pm 1.0/0.0$,
- **inexact** : the addition $(x + smallest) - smallest$ where x is the result and *smallest* the smallest subnormal number,
- **invalid** : the division $\pm 0.0/0.0$.

2.3 Common C procedures for double-double arithmetic

Hardware operators are usualy limited to double precision. To perform operations with more precision, then software solutions need to be used. One among them is to represent a floating point number as the sum of two non-overlapping floating-point numbers (or *double-double* numbers).

The algorithms are given as plain C functions, but it may be preferable, for performance issue, to implement them as macros, as in `libultim`. The code offers both versions, selected by the `DEKKER_AS_FUNCTIONS` constant which is set by default to 1 (functions).

A more recent proof is available in [25].

2.3.1 Exact sum algorithm Add12

This algorithm is also known as the Fast2Sum algorithm in the litterature.

Theorem 4 (Exact sum [22, 7])**.** *Let a and b be floating-point numbers, then the following method computes two floating-point numbers s and r, such that $s + r = a + b$ exactly, and s is the floating-point number which is closest to $a + b$.*

Listing 2.4: Add12Cond

```
void Add12Cond(double *s, double *r, a, b)
{
  double z;
  s = a + b;
  if (ABS(a) > ABS(b)){
    z = s - a;
    r = b - z;
  }else {
    z = s - b;
    r = a - z;
  }
}
```

Here ABS is a macro that returns the absolute value of a floating-point number. This algorithm requires 4 floating-point additions and 2 floating point tests (some of which are hidden in the ABS macro).

Note that if it is more efficient on a given architecture, the test can be replaced with a test on the exponents of a and b.

If we are able to prove that the exponent of a is always greater than that of b, then the previous algorithm to perform an exact addition of 2 floating-point numbers becomes :

Listing 2.5: Add12

```
void Add12(double *s, double *r, a, b)
{
  double z;
  s = a + b;
  z = s - a;
  r = b - z;
}
```

The cost of this algorithm is 3 floating-point additions.

2.3.2 Exact product algorithm Mul12

This algorithm is sometimes also known as the Dekker algorithm [16]. It was proven by Dekker but the proof predates the IEEE-754 standard and is difficult to read. An easier proof is available in [19] (see Th. 14).

Theorem 5 (Restricted exact product)**.** *Let a and b be two double-precision floating-point numbers, with 53 bits of mantissa. Let $c = 2^{\lceil \frac{53}{2} \rceil} + 1$. Assuming that $a < 2^{970}$ and $b < 2^{970}$, the following procedure computes the two floating-point numbers rh and rl such that $rh + rl = a + b$ with $rh = a \otimes b$:*

Listing 2.6: Mul12

```
void  Mul12(double *rh, double *rl, double u, double v){
```

```
  const double c = 134217729.;   /*  1+2^27 */
  double up, u1, u2, vp, v1, v2;

  up = u*c;         vp = v*c;
  u1 = (u-up)+up;   v1 = (v-vp)+vp;
  u2 = u-u1;        v2 = v-v1;

  *rh = u*v;
  *rl = (((u1*v1-*rh)+(u1*v2))+(u2*v1))+(u2*v2);
}
```

The cost of this algorithm is 10 floating-point additions and 7 floating-point multiplications.

The condition $a < 2^{970}$ and $b < 2^{970}$ prevents overflows when multiplying by c. If it cannot be proved statically, then we have to first test a and b, and prescale them so that the condition is true.

Theorem 6 (Exact product)**.** *Let a and b be two double-precision floating-point numbers, with 53 bits of mantissa. Let $c = 2^{\lceil \frac{53}{2} \rceil} + 1$. The following procedure computes the two floating-point numbers rh and rl such that $rh + rl = a + b$ with $rh = a \otimes b$:*

Listing 2.7: Mul12Cond

```
void Mul12Cond(double *rh, double *rl, double a, double b){
  const double two_970 = 0.99792015476735990582818635651841928 3e292;
  const double two_em53 = 0.11102230246251565404236316680908203125e-15;
  const double two_e53  = 9007199254740992.;
  double u, v;

  if (a>two_970)  u = a*two_em53;
  else            u = a;
  if (b>two_970)  v = b*two_em53;
  else            v = b;

  Mul12(rh, rl, u, v);

  if (a>two_970) {*rh *= two_e53; *rl *= two_e53;}
  if (b>two_970) {*rh *= two_e53; *rl *= two_e53;}
}
```

The cost in the worst case is then 4 tests over integers, 10 floating-point additions and 13 floating-point multiplications.

Finally, note that a fused multiply-and-add provides the Mul12 and Mul12Cond in only two instructions [8]. Here is the example code for the Itanium processor.

Listing 2.8: Mul12 on the Itanium

```
#define Mul12Cond(rh,rl,u,v)                                    \
{                                                               \
  *rh = u*v;                                                    \
  /* The following means: *rl = FMS(u*v-*rh) */                 \
  __asm__ __volatile__("fms %0 = %1, %2, %3\n ;;\n"             \
                       : "=f"(*rl)                              \
                       : "f"(u), "f"(v), "f"(*rh)               \
                       );                                       \
}
#define Mul12 Mul12Cond
```

The `crlibm` distribution attempts to use the FMA for systems on which it is availables (currently Itanium and PowerPC).

2.3.3 Double-double addition Add22

This algorithm, also due to Dekker [16], computes the sum of two double-double numbers as a double-double, with a relative error smaller than 2^{-103} (there is a proof in [16], a more recent one can be found in in [25]).

Listing 2.9: Add22Cond

```
void Add22Cond(double *zh, double *zl, double xh, double xl, double yh, double yl)
{
double r,s;
```

```
4
5 r = xh+yh;
6 s = (ABS(xh) > ABS(yh))? (xh-r+yh+yl+xl) : (yh-r+xh+xl+yl);
7 *zh = r+s;
8 *zl = r - (*zh) + s;
9 }
```

Here ABS is a macro that returns the absolute value of a floating-point number. Again, if this test can be resolved at compile-time, we get the faster `Add22` procedure:

Listing 2.10: Add22

```
1 void Add22(double *zh, double *zl, double xh, double xl, double yh, double yl)
2 {
3 double r,s;
4
5 r = xh+yh;
6 s = xh-r+yh+yl+xl;
7 *zh = r+s;
8 *zl = r - (*zh) + s;
9 }
```

2.3.4 Double-double multiplication Mul22

This algorithm, also due to Dekker [16], computes the product of two double-double numbers as a double-double, with a relative error smaller than 2^{-102}, under the condition $x_h < 2^{970}$ and $y_h < 2^{970}$ (there is a proof in [16], a more recent one can be found in in [25]).

Listing 2.11: Mul22

```
1 void Mul22(double *zh, double *zl, double xh, double xl, double yh, double yl)
2 {
3 double mh, ml;
4
5   const double c         = 134217729.;                /* 0x41A00000, 0x02000000 */
6   double up, u1, u2, vp, v1, v2;
7
8   up = xh*c;         vp = yh*c;
9   u1 = (xh-up)+up;   v1 = (yh-vp)+vp;
10  u2 = xh-u1;        v2 = yh-v1;
11
12  mh = xh*yh;
13  ml = (((u1*v1-mh)+(u1*v2))+(u2*v1))+(u2*v2);
14
15  ml += xh*yl + xl*yh;
16  *zh = mh+ml;
17  *zl = mh - (*zh) + ml;
18 }
```

Note that the bulk of this algorithm is a `Mul12(mh,ml,xh,yh)`. Of course there is a conditional version of this procedure but we have not needed it so far.

2.3.5 The multiplication procedure Mul122

Algorithm 1 (Mul122).
In: *a double precision number a and a double-double number* $b_h + b_l$
Out: *a double-double number* $r_h + r_l$
Preconditions on the arguments:

$$|b_l| \quad \leq \quad 2^{-53} \cdot |b_h|$$

Algorithm:

$$\begin{array}{l} (t_1, t_2) \leftarrow \textbf{Mul12}\,(a, b_h) \\ t_3 \leftarrow a \otimes b_l \\ t_4 \leftarrow t_2 \oplus t_3 \\ (r_h, r_l) \leftarrow \textbf{Add12}\,(t_1, t_4) \end{array}$$

Theorem 7 (Relative error of algorithm 1 **Mul122**).
Let be a and $b_h + b_l$ the values taken by the arguments of algorithm 1 **Mul122**
So the following holds for the values returned r_h and r_l:

$$r_h + r_l = (a \cdot (b_h + b_l)) \cdot (1 + \varepsilon)$$

where ε is bounded as follows:

$$|\varepsilon| \leq 2^{-102}$$

The values returned r_h and r_l will not overlap at all.

2.3.6 Double-double Horner step procedures

The multiply-and-add operator MulAdd212

Algorithm 2 (**MulAdd212**).
In: *a double-double number $c_h + c_l$, a double precision number a and a double-double number $b_h + b_l$*
Out: *a double-double number $r_h + r_l$*
Preconditions on the arguments:

$$\begin{aligned} |b_l| &\leq 2^{-53} \cdot |b_h| \\ |c_l| &\leq 2^{-53} \cdot |c_h| \\ |a \cdot (b_h + b_l)| &\leq 2^{-2} \cdot |c_h + c_l| \end{aligned}$$

Algorithm:

$$\begin{aligned} &(t_1, t_2) \leftarrow \mathbf{Mul12}\,(a, b_h) \\ &(t_3, t_4) \leftarrow \mathbf{Add12}\,(c_h, t_1) \\ &t_5 \leftarrow b_l \otimes a \\ &t_6 \leftarrow c_l \oplus t_2 \\ &t_7 \leftarrow t_5 \oplus t_6 \\ &t_8 \leftarrow t_7 \oplus t_4 \\ &(r_h, r_l) \leftarrow \mathbf{Add12}\,(t_3, t_8) \end{aligned}$$

Theorem 8 (Relative error of algorithm 2 **MulAdd212**).
Let be $c_h + c_l$, a and $b_h + b_l$ the arguments of algorithm 2 **MulAdd212** *verifying the given preconditions.*
So the following egality will hold for the returned values r_h and r_l

$$r_h + r_l = ((c_h + c_l) + a \cdot (b_h + b_l)) \cdot (1 + \varepsilon)$$

where ε is bounded by:

$$|\varepsilon| \leq 2^{-100}$$

The returned values r_h and r_l will not overlap at all.

The multiply-and-add operator MulAdd22

Algorithm 3 (**MulAdd22**).
In: *three double-double numbers $c_h + c_l$, $a_h + a_l$ and $b_h + b_l$*
Out: *a double-double number $r_h + r_l$*
Preconditions on the arguments:

$$\begin{aligned} |a_l| &\leq 2^{-53} \cdot |a_h| \\ |b_l| &\leq 2^{-53} \cdot |b_h| \\ |c_l| &\leq 2^{-53} \cdot |c_h| \\ |(a_h + a_l) \cdot (b_h + b_l)| &\leq 2^{-2} \cdot |c_h + c_l| \end{aligned}$$

Algorithm:

$$
\begin{array}{l}
(t_1, t_2) \leftarrow \mathbf{Mul12}\,(a_h, b_h) \\
(t_3, t_4) \leftarrow \mathbf{Add12}\,(c_h, t_1) \\
t_5 \leftarrow a_h \otimes b_l \\
t_6 \leftarrow a_l \otimes b_h \\
t_7 \leftarrow t_2 \oplus c_l \\
t_8 \leftarrow t_4 \oplus t_7 \\
t_9 \leftarrow t_5 \oplus t_6 \\
t_{10} \leftarrow t_8 \oplus t_9 \\
(r_h, r_l) \leftarrow \mathbf{Add12}\,(t_3, t_{10})
\end{array}
$$

Theorem 9 (Relative error of algorithm 3 **MulAdd22**).
Let be $c_h + c_l$, $a_h + a_l$ and $b_h + b_l$ the arguments of algorithm 3 **MulAdd22** *verifying the given preconditions. So the following egality will hold for the returned values r_h and r_l*

$$r_h + r_l = ((c_h + c_l) + (a_h + a_l) \cdot (b_h + b_l)) \cdot (1 + \varepsilon)$$

where ε is bounded by:

$$|\varepsilon| \leq 2^{-100}$$

The returned values r_h and r_l will not overlap at all.

2.3.7 Multiplication of a double-double by an integer

Use Cody and Waite algorithm. See for instance the log and the trigonometric argument reduction (chapter 3, p. 51).

2.4 Common C procedures for triple-double arithmetic

These procedures are used to reach accuracies of about 150 bits. They are detailed and proven in [25].

Algorithm 4 (Renormalization).
In: $a_h, a_m, a_l \in \mathbb{F}$ *verifying the following preconditions:*
Preconditions:

- *None of the numbers a_h, a_m, a_l is subnormal*
- *a_h et a_m do not overlap in more than* 51 *bits*
- *a_m et a_l do not overlap in more than* 51 *bits*

which means formally:

$$
\begin{array}{rcl}
|a_m| & \leq & 2^{-2} \cdot |a_h| \\
|a_l| & \leq & 2^{-2} \cdot |a_m| \\
|a_l| & \leq & 2^{-4} \cdot |a_h|
\end{array}
$$

Out: $r_h, r_m, r_l \in \mathbb{F}$

$$
\begin{array}{rcl}
(t_{1h}, t_{1l}) & \leftarrow & \mathbf{Add12}\,(a_m, a_l) \\
(r_h, t_{2l}) & \leftarrow & \mathbf{Add12}\,(a_h, t_{1h}) \\
(r_m, r_l) & \leftarrow & \mathbf{Add12}\,(t_{2l}, t_{1l})
\end{array}
$$

Theorem 10 (Correctness of the renormalization algorithm 4 **Renormalize3**).
For all arguments verifying the preconditions of procedure **Renormalize3**, *the values returned r_h, r_m and r_l will not overlap unless they are all equal to* 0 *and their sum will be exactly the sum of the values in argument a_h, a_m et a_l. This implies:*

$$|r_m| \leq 2^{-52} \cdot |r_h|$$
$$|r_l| \leq 2^{-53} \cdot |r_m|$$

2.4.1 The addition operator Add33

Algorithm 5 (**Add33**).
In: *two triple-double numbers,* $a_h + a_m + a_l$ *and* $b_h + b_m + b_l$
Out: *a triple-double number* $r_h + r_m + r_l$
Preconditions on the arguments:

$$\begin{aligned} |b_h| &\leq \frac{3}{4} \cdot |a_h| \\ |a_m| &\leq 2^{-\alpha_o} \cdot |a_h| \\ |a_l| &\leq 2^{-\alpha_u} \cdot |a_m| \\ |b_m| &\leq 2^{-\beta_o} \cdot |b_h| \\ |b_l| &\leq 2^{-\beta_u} \cdot |b_m| \\ \alpha_o &\geq 4 \\ \alpha_u &\geq 1 \\ \beta_o &\geq 4 \\ \beta_u &\geq 1 \end{aligned}$$

Algorithm:

$$\begin{aligned} &(r_h, t_1) \leftarrow \mathbf{Add12}\,(a_h, b_h) \\ &(t_2, t_3) \leftarrow \mathbf{Add12}\,(a_m, b_m) \\ &(t_7, t_4) \leftarrow \mathbf{Add12}\,(t_1, t_2) \\ &t_6 \leftarrow a_l \oplus b_l \\ &t_5 \leftarrow t_3 \oplus t_4 \\ &t_8 \leftarrow t_5 \oplus t_6 \\ &(r_m, r_l) \leftarrow \mathbf{Add12}\,(t_7, t_8) \end{aligned}$$

Theorem 11 (Relative error of algorithm 5 **Add33**).
Let be $a_h + a_m + a_l$ *and* $b_h + b_m + b_l$ *the triple-double arguments of algorithm 5* **Add33** *verifying the given preconditions.*
So the following egality will hold for the returned values r_h, r_m *and* r_l

$$r_h + r_m + r_l = ((a_h + a_m + a_l) + (b_h + b_m + b_l)) \cdot (1 + \varepsilon)$$

where ε *is bounded by:*

$$|\varepsilon| \leq 2^{-\min(\alpha_o+\alpha_u, \beta_o+\beta_u)-47} + 2^{-\min(\alpha_o, \beta_o)-98}$$

The returned values r_m *and* r_l *will not overlap at all and the overlap of* r_h *and* r_m *will be bounded by the following expression:*

$$|r_m| \leq 2^{-\min(\alpha_o, \beta_o)+5} \cdot |r_h|$$

2.4.2 The addition operator Add233

Algorithm 6 (**Add233**).
In: *a double-double number* $a_h + a_l$ *and a triple-double number* $b_h + b_m + b_l$
Out: *a triple-double number* $r_h + r_m + r_l$
Preconditions on the arguments:

$$\begin{aligned} |b_h| &\leq 2^{-2} \cdot |a_h| \\ |a_l| &\leq 2^{-53} \cdot |a_h| \\ |b_m| &\leq 2^{-\beta_o} \cdot |b_h| \\ |b_l| &\leq 2^{-\beta_u} \cdot |b_m| \end{aligned}$$

Algorithm:

$$
\begin{array}{l}
(r_h, t_1) \leftarrow \mathbf{Add12}\,(a_h, b_h) \\
(t_2, t_3) \leftarrow \mathbf{Add12}\,(a_l, b_m) \\
(t_4, t_5) \leftarrow \mathbf{Add12}\,(t_1, t_2) \\
t_6 \leftarrow t_3 \oplus b_l \\
t_7 \leftarrow t_6 \oplus t_5 \\
(r_m, r_l) \leftarrow \mathbf{Add12}\,(t_4, t_7)
\end{array}
$$

Theorem 12 (Relative error of algorithm 6 **Add233**).
Let be $a_h + a_l$ and $b_h + b_m + b_l$ the values taken in argument of algorithm 6 **Add233**. *Let the preconditions hold for this values.*
So the following holds for the values returned by the algorithm r_h, r_m and r_l

$$r_h + r_m + r_l = ((a_h + a_m + a_l) + (b_h + b_m + b_l)) \cdot (1 + \varepsilon)$$

where ε is bounded by

$$|\varepsilon| \leq 2^{-\beta_o - \beta_u - 52} + 2^{-\beta_o - 104} + 2^{-153}$$

The values r_m and r_l will not overlap at all and the overlap of r_h and r_m will be bounded by:

$$|r_m| \leq 2^{-\gamma} \cdot |r_h|$$

with

$$\gamma \geq \min\,(45, \beta_o - 4, \beta_o + \beta_u - 2)$$

2.4.3 The addition operator Add133

Algorithm 7 (**Add133**).
In: *a double precision number a and a triple-double number $b_h + b_m + b_l$*
Out: *a triple-double number $r_h + r_m + r_l$*
Preconditions on the arguments:

$$
\begin{array}{rcl}
|b_h| & \leq & 2^{-2} \cdot |a| \\
|b_m| & \leq & 2^{-\beta_o} \cdot |b_h| \\
|b_l| & \leq & 2^{-\beta_u} \cdot |b_m|
\end{array}
$$

Algorithm:

$$
\begin{array}{l}
(r_h, t_1) \leftarrow \mathbf{Add12}\,(a, b_h) \\
(t_2, t_3) \leftarrow \mathbf{Add12}\,(t_1, b_m) \\
t_4 \leftarrow t_3 \oplus b_l \\
(r_m, r_l) \leftarrow \mathbf{Add12}\,(t_2, t_4)
\end{array}
$$

Theorem 13 (Relative error of algorithm 7 **Add133**).
Let be a and $b_h + b_m + b_l$ the values taken in argument of algorithm 7 **Add133**. *Let the preconditions hold for this values.*
So the following holds for the values returned by the algorithm r_h, r_m and r_l

$$r_h + r_m + r_l = (a + (b_h + b_m + b_l)) \cdot (1 + \varepsilon)$$

where ε is bounded by

$$|\varepsilon| \leq 2^{-\beta_o - \beta_u - 52} + 2^{-154}$$

The values r_m and r_l will not overlap at all and the overlap of r_h and r_m will be bounded by:

$$|r_m| \leq 2^{-\gamma} \cdot |r_h|$$

with

$$\gamma \geq \min\,(47, \beta_o - 2, \beta_o + \beta_u - 1)$$

2.4.4 The multiplication procedure Mul33

Algorithm 8 (**Mul33**).
In: *two triple-double numbers* $a_h + a_m + a_l$ *and* $b_h + b_m + b_l$
Out: *a triple-double number* $r_h + r_m + r_l$
Preconditions on the arguments:

$$\begin{aligned} |a_m| &\leq 2^{-\alpha_o} \cdot |a_h| \\ |a_l| &\leq 2^{-\alpha_u} \cdot |a_m| \\ |b_m| &\leq 2^{-\beta_o} \cdot |b_h| \\ |b_l| &\leq 2^{-\beta_u} \cdot |b_m| \end{aligned}$$

with

$$\begin{aligned} \alpha_o &\geq 2 \\ \alpha_u &\geq 2 \\ \beta_o &\geq 2 \\ \beta_u &\geq 2 \end{aligned}$$

Algorithm:

$$\begin{aligned} &(r_h, t_1) \leftarrow \mathbf{Mul12}\,(a_h, b_h) \\ &(t_2, t_3) \leftarrow \mathbf{Mul12}\,(a_h, b_m) \\ &(t_4, t_5) \leftarrow \mathbf{Mul12}\,(a_m, b_h) \\ &(t_6, t_7) \leftarrow \mathbf{Mul12}\,(a_m, b_m) \\ &t_8 \leftarrow a_h \otimes b_l \\ &t_9 \leftarrow a_l \otimes b_h \\ &t_{10} \leftarrow a_m \otimes b_l \\ &t_{11} \leftarrow a_l \otimes b_m \\ &t_{12} \leftarrow t_8 \oplus t_9 \\ &t_{13} \leftarrow t_{10} \oplus t_{11} \\ &(t_{14}, t_{15}) \leftarrow \mathbf{Add12}\,(t_1, t_6) \\ &t_{16} \leftarrow t_7 \oplus t_{15} \\ &t_{17} \leftarrow t_{12} \oplus t_{13} \\ &t_{18} \leftarrow t_{16} \oplus t_{17} \\ &(t_{19}, t_{20}) \leftarrow \mathbf{Add12}\,(t_{14}, t_{18}) \\ &(t_{21}, t_{22}) \leftarrow \mathbf{Add22}\,(t_2, t_3, t_4, t_5) \\ &(r_m, r_l) \leftarrow \mathbf{Add22}\,(t_{21}, t_{22}, t_{19}, t_{20}) \end{aligned}$$

Theorem 14 (Relative error of algorithm 8 **Mul33**).
Let be $a_h + a_m + a_l$ *and* $b_h + b_m + b_l$ *the values taken by the arguments of algorithm 8* **Mul33**
So the following holds for the values returned r_h, r_m *and* r_l*:*

$$r_h + r_m + r_l = ((a_h + a_m + a_l) \cdot (b_h + b_m + b_l)) \cdot (1 + \varepsilon)$$

where ε *is bounded as follows:*

$$\begin{aligned} |\varepsilon| \leq\ & 2^{-151} \\ &+ 2^{-99-\alpha_o} \\ &+ 2^{-99-\beta_o} \\ &+ 2^{-49-\alpha_o-\alpha_u} \\ &+ 2^{-49-\beta_o-\beta_u} \\ &+ 2^{-50-\alpha_o-\beta_o-\beta_u} \\ &+ 2^{-50-\alpha_o-\alpha_u-\beta_o} \\ &+ 2^{-101-\alpha_o-\beta_o} \\ &+ 2^{-52-\alpha_o-\alpha_u-\beta_o-\beta_u} \end{aligned}$$

The values returned r_m and r_l will not overlap at all and the overlap of r_h and r_m will be bounded as follows:

$$|r_m| \leq 2^{-\gamma_o} \cdot |r_h|$$

with

$$\gamma_o \geq \min(48, \alpha_o - 4, \beta_o - 4, \alpha_o + \alpha_u - 4, \beta_o + \beta_u - 4, \alpha_o + \alpha_o - 4)$$

2.4.5 The multiplication procedure Mul23

Algorithm 9 (Mul23).
In: *two double-double numbers $a_h + a_l$ and $b_h + b_l$*
Out: *a triple-double number $r_h + r_m + r_l$*
Preconditions on the arguments:

$$\begin{aligned} |a_l| &\leq 2^{-53} \cdot |a_h| \\ |b_l| &\leq 2^{-53} \cdot |b_h| \end{aligned}$$

Algorithm:

$$\begin{aligned} &(r_h, t_1) \leftarrow \mathbf{Mul12}(a_h, b_h) \\ &(t_2, t_3) \leftarrow \mathbf{Mul12}(a_h, b_l) \\ &(t_4, t_5) \leftarrow \mathbf{Mul12}(a_l, b_h) \\ &t_6 \leftarrow a_l \otimes b_l \\ &(t_7, t_8) \leftarrow \mathbf{Add22}(t_2, t_3, t_4, t_5) \\ &(t_9, t_{10}) \leftarrow \mathbf{Add12}(t_1, t_6) \\ &(r_m, r_l) \leftarrow \mathbf{Add22}(t_7, t_8, t_9, t_{10}) \end{aligned}$$

Theorem 15 (Relative error of algorithm 9 **Mul23**).
Let be $a_h + a_l$ and $b_h + b_l$ the values taken by arguments of algorithm 9 **Mul23**
So the following holds for the values returned r_h, r_m and r_l:

$$r_h + r_m + r_l = ((a_h + a_l) \cdot (b_h + b_l)) \cdot (1 + \varepsilon)$$

where ε is bounded as follows:

$$|\varepsilon| \leq 2^{-149}$$

The values returned r_m and r_l will not overlap at all and the overlap of r_h and r_m will be bounded as follows:

$$|r_m| \leq 2^{-48} \cdot |r_h|$$

2.4.6 The multiplication procedure Mul233

Algorithm 10 (Mul233).
In: *a double-double number $a_h + a_l$ and a triple-double number $b_h + b_m + b_l$*
Out: *a triple-double number $r_h + r_m + r_l$*
Preconditions on the arguments:

$$\begin{aligned} |a_l| &\leq 2^{-53} \cdot |a_h| \\ |b_m| &\leq 2^{-\beta_o} \cdot |b_h| \\ |b_l| &\leq 2^{-\beta_u} \cdot |b_m| \end{aligned}$$

with

$$\begin{aligned} \beta_o &\geq 2 \\ \beta_u &\geq 1 \end{aligned}$$

Algorithm:

$$
\begin{array}{l}
(r_h, t_1) \leftarrow \textbf{Mul12}\,(a_h, b_h)\\
(t_2, t_3) \leftarrow \textbf{Mul12}\,(a_h, b_m)\\
(t_4, t_5) \leftarrow \textbf{Mul12}\,(a_h, b_l)\\
(t_6, t_7) \leftarrow \textbf{Mul12}\,(a_l, b_h)\\
(t_8, t_9) \leftarrow \textbf{Mul12}\,(a_l, b_m)\\
t_{10} \leftarrow a_l \otimes b_l\\
(t_{11}, t_{12}) \leftarrow \textbf{Add22}\,(t_2, t_3, t_4, t_5)\\
(t_{13}, t_{14}) \leftarrow \textbf{Add22}\,(t_6, t_7, t_8, t_9)\\
(t_{15}, t_{16}) \leftarrow \textbf{Add22}\,(t_{11}, t_{12}, t_{13}, t_{14})\\
(t_{17}, t_{18}) \leftarrow \textbf{Add12}\,(t_1, t_{10})\\
(r_m, r_l) \leftarrow \textbf{Add22}\,(t_{17}, t_{18}, t_{15}, t_{16})
\end{array}
$$

Theorem 16 (Relative error of algorithm 10 **Mul233**).
Let be $a_h + a_l$ and $b_h + b_m + b_l$ the values in argument of algorithm 10 **Mul233** *such that the given preconditions hold.*
So the following will hold for the values r_h, r_m and r_l returned

$$r_h + r_m + r_l = ((a_h + a_l) \cdot (b_h + b_m + b_l)) \cdot (1 + \varepsilon)$$

where ε is bounded as follows:

$$|\varepsilon| \leq \frac{2^{-99-\beta_o} + 2^{-99-\beta_o-\beta_u} + 2^{-152}}{1 - 2^{-53} - 2^{-\beta_o+1} - 2^{-\beta_o-\beta_u+1}} \leq 2^{-97-\beta_o} + 2^{-97-\beta_o-\beta_u} + 2^{-150}$$

The values r_m and r_l will not overlap at all and the following bound will be verified for the overlap of r_h and r_m:

$$|r_m| \leq 2^{-\gamma} \cdot |r_h|$$

where

$$\gamma \geq \min\,(48, \beta_o - 4, \beta_o + \beta_u - 4)$$

2.4.7 The multiplication procedure Mul133

Algorithm 11 (Mul133).
In: *a double number a and a triple-double number $b_h + b_m + b_l$*
Out: *a triple-double number $r_h + r_m + r_l$*
Preconditions on the arguments:

$$\begin{array}{rcl} |b_m| & \leq & 2^{-\beta_o} \cdot |b_h| \\ |b_l| & \leq & 2^{-\beta_u} \cdot |b_m| \end{array}$$

with

$$\begin{array}{rcl} \beta_o & \geq & 2 \\ \beta_u & \geq & 2 \end{array}$$

Algorithm:

$$
\begin{array}{l}
(r_h, t_2) \leftarrow \textbf{Mul12}\,(a, b_h)\\
(t_3, t_4) \leftarrow \textbf{Mul12}\,(a, b_m)\\
t_5 \leftarrow a \otimes b_l\\
(t_9, t_7) \leftarrow \textbf{Add12}\,(t_2, t_3)\\
t_8 \leftarrow t_4 \oplus t_5\\
t_{10} \leftarrow t_7 \oplus t_8\\
(r_m, r_l) \leftarrow \textbf{Add12}\,(t_9, t_{10})
\end{array}
$$

Theorem 17 (Relative error of algorithm 11 **Mul133**).
Let be a and $b_h + b_m + b_l$ the values in argument of algorithm 11 **Mul133** *such that the given preconditions hold.*
So the following will hold for the values r_h, r_m and r_l returned

$$r_h + r_m + r_l = (a \cdot (b_h + b_m + b_l)) \cdot (1 + \varepsilon)$$

where ε is bounded as follows:

$$|\varepsilon| \leq 2^{-101-\beta_o} + 2^{-49-\beta_o-\beta_u} + 2^{-156}$$

The values r_m and r_l will not overlap at all and the following bound will be verified for the overlap of r_h and r_m:

$$|r_m| \leq 2^{-\gamma} \cdot |r_h|$$

where

$$\gamma \geq \min(47, \beta_o - 5, \beta_o + \beta_u - 5)$$

2.4.8 The multiplication procedure Mul123

Algorithm 12 (**Mul123**).
In: *a double number a and a double-double number $b_h + b_l$*
Out: *a triple-double number $r_h + r_m + r_l$*
Preconditions on the arguments:

$$|b_l| \quad \leq \quad 2^{-53} \cdot |b_h|$$

Algorithm:

$$\begin{array}{l}(r_h, t_1) \leftarrow \textbf{Mul12}\,(a, b_h)\\ (t_2, t_3) \leftarrow \textbf{Mul12}\,(a, b_l)\\ (t_5, t_4) \leftarrow \textbf{Add12}\,(t_1, t_2)\\ t_6 \leftarrow t_3 \otimes t_4\\ (r_m, r_l) \leftarrow \textbf{Add12}\,(t_5, t_6)\end{array}$$

Theorem 18 (Relative error of algorithm 12 **Mul123**).
Let be a and $b_h + b_l$ the values in argument of algorithm 12 **Mul123** *such that the given preconditions hold.*
So the following will hold for the values r_h, r_m and r_l returned

$$r_h + r_m + r_l = (a \cdot (b_h + b_l)) \cdot (1 + \varepsilon)$$

where ε is bounded as follows:

$$|\varepsilon| \leq 2^{-154}$$

The values r_m and r_l will not overlap at all and the following bound will be verified for the overlap of r_h and r_m:

$$|r_m| \leq 2^{-\gamma} \cdot |r_h|$$

where

$$\gamma \geq 47$$

2.4.9 Final rounding to the nearest even

Algorithm 13 (Final rounding to the nearest (even)).
In: *a triple-double number $x_h + x_m + x_l$*
Out: *a double precision number x' returned by the algorithm*
Preconditions on the arguments:

- *x_h and x_m as well as x_m and x_l do not overlap*
- *$x_m = \circ(x_m + x_l)$*

- $x_h \neq 0$, $x_m \neq 0$ *and* $x_l \neq 0$
- $\circ(x_h + x_m) \notin \{x_h^-, x_h, x_h^+\} \Rightarrow |(x_h + x_m) - \circ(x_h + x_m)| \neq \frac{1}{2} \cdot \mathrm{ulp}(\circ(x_h + x_m))$

Algorithm:

$t_1 \leftarrow x_h^-$
$t_2 \leftarrow x_h \ominus t_1$
$t_3 \leftarrow t_2 \otimes \frac{1}{2}$
$t_4 \leftarrow x_h^+$
$t_5 \leftarrow t_4 \ominus x_h$
$t_6 \leftarrow t_5 \otimes \frac{1}{2}$

if $(x_m \neq -t_3)$ **and** $(x_m \neq t_6)$ **then**
  return $(x_h \oplus x_m)$
else
  if $(x_m \otimes x_l > 0.0)$ **then**
    if $(x_h \otimes x_l > 0.0)$ **then**
      return x_h^+
    else
      return x_h^-
    end if
  else
    return x_h
  end if
end if

Theorem 19 (Correctness of the final rounding procedure 13)**.**
Let be **A** *the algorithm 13 said " Final rounding to the nearest (even)". Let be $x_h + x_m + x_l$ triple-double number for which the preconditions of algorithm* **A** *hold. Let us notate x' the double precision number returned by the procedure.*
So

$$x' = \circ(x_h + x_m + x_l)$$

i.e. **A** *is a correct rounding procedure for round-to-nearest-ties-to-even mode.*

2.4.10 Final rounding for the directed modes

Theorem 20 (Directed final rounding of a triple-double number)**.**
Let be $x_h + x_m + x_l \in \mathbb{F} + \mathbb{F} + \mathbb{F}$ a non-overlapping triple-double number.
Let be $\diamond$ a directed rounding mode.
Let be **A** *the following instruction sequence:*

$(t_1, t_2) \leftarrow$ **Add12** (x_h, x_m)
$t_3 \leftarrow t_2 \oplus x_l$
return $\diamond(t_1 + t_3)$

So **A** *is a correct rounding procedure for the rounding mode $\diamond$.*

2.5 Horner polynomial approximations

Most function evaluation schemes include some kind of polynomial evaluation over a small interval. Classically, we use the Horner scheme, which is the best suited in this case.

For a polynomial of degree d, noting c_i its coefficients, the Horner scheme consists in computing S_0 using the following recursion:

$$\begin{cases} S_d(x) &= c_d \\ S_k(x) &= c_k + xS_{k+1}(x) \quad \text{for} \quad 0 \leq k < d \end{cases}$$

In the quick phase, the evaluation always begins in double-precision, but it may end with double-double arithmetic in order to compute the result as a double-double (from a performance point of view it is a less costly to begin the double-double part with a double-double addition rather than with a double-double multiplication). In this section only, $\oplus$ and $\otimes$ therefore denote either a double, or a double-double, or an SCS operation.

For fast and accurate function evaluation, we try to have $\overline{x}$ small with respect to the coefficients. In this case the error in one step is scaled down for the next step by the multiplication by x, allowing for an accumulated overall error which is actually close to that of the last operation.

In addition we note

- $\delta_\oplus$ and $\delta_\otimes$ the absolute error when performing an atomic $\oplus$ or $\otimes$, and $\varepsilon_\oplus$ and $\varepsilon_\otimes$ the corresponding relative error (we use whichever allows the finer error analysis, as detailed below). It can change during a Horner evaluation, typically if the evaluation begins in double-precision ($\overline{\varepsilon}_\oplus = \overline{\varepsilon}_\otimes = 2^{-53}$) and ends in double-double ($\overline{\varepsilon}_\oplus = \overline{\varepsilon}_\otimes = 2^{-102}$).

- c_j the coefficient of P of degree j, considered exactly representable (if c_j is an approximation to some exact value $\hat{c}_j$, the corresponding error is taken into account in the computation of the approximation error for this polynomial)

- $\overline{x}$ the maximum value of $|x|$ over the considered interval

- $\overline{\varepsilon}_x$ a bound on the relative error of the input x with respect to the exact mathematical value $\hat{x}$ it represents. Note that sometimes argument reduction is exact, and will yield $\overline{\varepsilon}_x = 0$ (see for instance the logarithm). Also note that $\overline{\varepsilon}_x$ may change during the evaluation: Typically, if $\hat{x}$ is approximated as a double-double $x_h + x_l = \hat{x}(1+\varepsilon)$, then the first iterations will be computed in double-precision only, and the error will be $\overline{\varepsilon}_x = 2^{-53}$ if one is able to prove that $x_h = \circ(\hat{x})$. For the last steps of the evaluation, using double-double arithmetic on $x_h + x_l$, the error will be improved to $\overline{\varepsilon}_x = \overline{\varepsilon}$.

- $p_k = x \otimes s_k$ the result of a multiplication step in the Horner scheme. We recursively evaluate its relative and absolute error $\overline{\varepsilon}_j^\times$ and $\overline{\delta}_j^\times$ with respect to the exact mathematical value $P_j(\hat{x}) = xS_{j+1}(\hat{x})$.

- $s_k = c_k \oplus p_{k+1}$ (with $s_d = c_d$) the result of an addition step in the Horner scheme. We recursively evaluate its absolute error $\overline{\varepsilon}_j^+$ with respect to the exact mathematical value $S_k(\hat{x})$.

- $\overline{s}_k$ the maximum value that s_k may take for $|x| \leq \overline{x}$.

- $||S_k||^\infty$ the infinite norm of S_k for $|x| \leq \overline{x}$.

Given $|x| \leq \overline{x}$, we want to compute by recurrence

$$\begin{cases} p_k &= x \otimes s_{k+1} \;=\; \hat{x}S_{k+1}(\hat{x})(1+\overline{\varepsilon}_k^\times) \\ s_k &= c_k \oplus p_k \;=\; S_k(\hat{x}) + \overline{\delta}_k^+ \end{cases}$$

The following computes tight bounds on $\varepsilon_k^\times$ and on δ_k^+.

- Initialization/degree 0 polynomial:

$$\begin{cases} s_d &= c_d \\ \overline{s}_d &= \underline{s}_d \;=\; |c_d| \\ \overline{\varepsilon}_d^+ &= 0 \end{cases}$$

- Horner steps:

– multiplication step:

$$\begin{aligned} p_k &= x \otimes s_{k+1} \\ &= \hat{x}(1+\varepsilon_x) \otimes (S_{k+1}(\hat{x}) + \delta^+_{k+1}) \\ &= \hat{x} S_{k+1}(\hat{x})(1+\varepsilon_x)(1+\frac{\delta^+_{k+1}}{S_{k+1}(\hat{x})})(1+\varepsilon_\otimes) \end{aligned}$$

We therefore get

$$p_k = P_k(\hat{x})(1+\varepsilon^\times_k) \tag{2.1}$$

with

$$\overline{\varepsilon}^\times_k = (1+\overline{\varepsilon}_x)(1+\overline{\varepsilon}^+_{k+1})(1+\overline{\varepsilon}_\otimes) - 1 \tag{2.2}$$

Here we will take $\overline{\varepsilon}'_\otimes = 2^{-53}$ or $\overline{\varepsilon}'_\otimes = 2^{-102}$ or $\overline{\varepsilon}'_\otimes = 2^{-205}$ respectively for double, double-double, or SCS operations.

– addition step

$$\begin{aligned} s_k &= c_k \oplus p_k \\ &= c_k + p_k + \delta_\oplus \\ &= c_k + P_k(\hat{x})(1+\varepsilon^\times_k) + \delta_\oplus \\ &= c_k + P_k(\hat{x}) + \varepsilon^\times_k P_k(\hat{x}) + \delta_\oplus \\ &= S_k(\hat{x}) + \varepsilon^\times_k P_k(\hat{x}) + \delta_\oplus \end{aligned}$$

We therefore get

$$s_k = S_k(\hat{x}) + \delta^+_k \tag{2.3}$$

$$\overline{\delta}^+_k = \overline{\varepsilon}^\times_k ||P_k||^\infty + \overline{\delta}_\oplus \tag{2.4}$$

Here $\overline{\delta}_\oplus$ will be computed for double-precision operations as

$$\overline{\delta}_\oplus = \frac{1}{2}\text{ulp}(||S_k||^\infty + \overline{\varepsilon}^\times_k ||P_k||^\infty) \quad .$$

For double-double or SCS operations, $\overline{\delta}_\oplus$ will be computed as

$$\overline{\delta}_\oplus = 2^{-\nu}(||S_k||^\infty + \overline{\varepsilon}^\times_k ||P_k||^\infty)$$

with $\nu = 102$ and $\nu = 205$ respectively.

To compute a relative error out of the absolute error $\overline{\delta}^+_0$, there are two cases to consider.

- If $c_0 \neq 0$, for small values of x, a good bound on the overall relative error is to divide δ_0 by the minimum of $|s_0|$, which – provided $\overline{x}$ is sufficiently small compared to c_0 – is well approximated by

  $$\underline{s}_0 = |c_0| - \overline{x}.\overline{s}_1$$

  where $\overline{s}_1 = ||S_1||^\infty + \delta_1$. An upper bound on the total relative error is then

  $$\rho = \frac{\delta^+_0}{|c_0| - \overline{x}.\overline{s}_1}$$

  When computing on double-precision numbers we want the error bound to be as tight as possible, as it directly impacts the performance as explained in Section 1.3.6. We may therefore check that $c_k \oplus p_k$ has a constant exponent for all the values of p_k. In which case, the above approximation is the tightest possible. If it is not the case (which is unlikely, as p_k is small w.r.t c_k), then the ulp above may take two different values. We divide the interval of p_k into two sub-intervals, and we compute δ^+_k, $\underline{s}_0$ and ρ on both to take the max.

  This is currently not implemented.

- If $c_0 = 0$, then the last addition is exact in double as well as double-double, and an efficient implementation will skip it anyway. The overall relative error is that of the last multiplication, and is given as $\bar{\varepsilon}'_0$.

2.6 Helper functions

2.6.1 High accuracy square roots

Some of `crlibm`'s functions need high precision square roots. They are not intended to be used outside `crlibm`. In particular, we do currently not guarantee the correct rounding of their results because this property is not needed for our purposes. Their implementation does not handle all possible special cases ($x < 0$, NaN, ∞ etc.) neither.

We currently provide two C macros computing the square root of a double precision argument either in double-double precision with at least 100 correct bits (in faithful rounding) or in triple-double precision with an accuracy of at least 146 bits (in faithful rounding). The corresponding macros are called **sqrt12** and **sqrt13**.

The implementation of these macros was guided by the following principles:

- no dependency on other `libms`, so avoidance of bootstrapping a Newton iteration by a double precision square root implemented elsewhere,
- high efficiency,
- a small memory footprint,
- the possible use of hardware support on some platforms in the future.

Overview of the algorithm

The algorithm uses a combination of polynomial approximation and Newton iteration.

After handling some special cases, the argument $x = 2^{E'} \cdot m'$ is reduced into its exponent E' stored in integer and its fractional part m' stored as a double precision number. This argument reduction is obviously exact. The two values are then adjusted as follows:

$$E = \begin{cases} E' & \text{if } \exists n \in \mathbb{N} \,.\, E' = 2n \\ E'+1 & \text{otherwise} \end{cases} \qquad m = \begin{cases} m' & \text{if } \exists n \in \mathbb{N} \,.\, E' = 2n \\ \frac{m'}{2} & \text{otherwise} \end{cases}$$

One easily checks that $\frac{1}{2} \leq m \leq 2$ and that E is always even. Thus

$$\sqrt{x} = \sqrt{2^E \cdot m} = 2^{\frac{E}{2}} \cdot \sqrt{m} = 2^{\frac{E}{2}} \cdot m \cdot \frac{1}{\sqrt{m}}$$

The algorithm therefore approximates $\hat{r} = \frac{1}{\sqrt{m}}$ and reconstructs the square root by multiplying by m and exactly by $2^{\frac{E}{2}}$.

The reciprocal square root $\hat{r}$ is approximated in two steps. First, a polynomial approximation yields to $r_0 = \hat{r} \cdot (1 + \varepsilon_1)$, which is exact to about 8 bits. In a second step, this approximation is refined by a Newton iteration that approximately doubles its accuracy at each step. So for a double-double result, 4 iterations are needed and for a triple-double result 5.

The initial polynomial approximation is less exact than the one provided by Itanium's operation, which allows for using this hardware assistance in the future.

Special case handling

The square root of a double precision number can never be subnormal. In fact, if $\sqrt{x} \leq 2^{-1021}$, $x = \sqrt{x}^2 \leq 2^{-1042441}$, a value that is is not representable in double precision.

Concerning subnormals in argument, it to be mentioned that still E' and m' can be found such that $x = 2^{E'} \cdot m$ exactly and $1 \leq m' \leq 2$. Only the extraction sequence must be modified: x is first multiplied

by 2^{52} where E' is set to -52. The double number x is thus no longer a subnormal an integer handling can extract its mantissa easily. The extraction of the exponent takes into account the preceeding bias of E'. The case $x = 0$ is filtered out before. Obviously $\sqrt{0} = 0$ is returned for this argument.

The special cases $x < 0$, $x = \pm\infty$ and $x = \text{NaN}$ are not handled since they can be easily excluded by the code using the square root macros.

Special case handling is implemented as follows:

Listing 2.12: Special case handling

```
/* Special case x = 0 */
if (x == 0) {
  *resh = x;
  *resl = 0;
} else {

  E = 0;

  /* Convert to integer format */
  xdb.d = x;

  /* Handle subnormal case */
  if (xdb.i[HI] < 0x00100000) {
    E = -52;
    xdb.d *= ((db_number) ((double) SQRTTWO52)).d;          /* make x a normal number */
  }

  /* Extract exponent E and mantissa m */
  E += (xdb.i[HI]>>20)-1023;
  xdb.i[HI] = (xdb.i[HI] & 0x000fffff) | 0x3ff00000;
  m = xdb.d;

  /* Make exponent even */
  if (E & 0x00000001) {
    E++;
    m *= 0.5;     /* Suppose now 1/2 <= m <= 2 */
  }

  /* Construct sqrt(2^E) = 2^(E/2) */
  xdb.i[HI] = (E/2 + 1023) << 20;
  xdb.i[LO] = 0;
```

Polynomial approximation

The reciprocal square root $\hat{r} = \frac{1}{\sqrt{m}}$ is approximated in the domain $m \in \left[\frac{1}{2}; 2\right]$ by a polynomial $p(m) = \sum_{i=0}^{4} c_i \cdot m^i$ of degree 4. The polynomial's coefficients c_0 through c_4 are stored in double precision. The following values are used:

$$
\begin{aligned}
c_0 &= 2.50385236695888790947606139525305479764938354492188 \\
c_1 &= -3.29763389114324168005509818613063544034957885742188 \\
c_2 &= 2.75726076139124520736345402838196605443954467773438 \\
c_3 &= -1.15233725777933848632983426796272397041320800781250 \\
c_4 &= 0.18690006667980096910497422868502326309680938720703 1
\end{aligned}
$$

The relative approximation error $\varepsilon_{\text{approx}} = \frac{p(m)-\hat{r}}{\hat{r}}$ is bounded by $|\varepsilon_{\text{approx}}| \leq 2^{-8.32}$ for $m \in \left[\frac{1}{2}; 2\right]$.

The polynomial is evaluated in double precision using Horner's scheme. There may be some cancellation in the different steps but the relative arithmetical error $\varepsilon_{\text{arithpoly}}$ is always less in magnitude than 2^{-30}. This will be shown in more detail below.

The code implementing the polynomial approximation reads:

Listing 2.13: Polynomial approximation

```
r0 = SQRTPOLYC0 + m * (SQRTPOLYC1 + m * (SQRTPOLYC2 + m * (SQRTPOLYC3 + m * SQRTPOLYC4)));
```

So 4 double precision multiplications and 4 additions are needed for computing the initial approximation. They can be replaced by 4 FMA instructions, if available.

Double and double-double Newton iteration

The polynomial approximation is then refined using the following iteration scheme:

$$r_{i+1} = \frac{1}{2} \cdot r_i \cdot (3 - m \cdot r_i^2)$$

If the arithmetic operations were exact, one would obtain the following error estimate:

$$\begin{aligned}
\varepsilon_{i+1} &= \frac{r_i - \hat{r}}{\hat{r}} \\
&= \frac{\frac{1}{2} \cdot r_i \cdot \left(3 - m \cdot r_i^2\right) - \hat{r}}{\hat{r}} \\
&= \frac{\frac{1}{2} \cdot \hat{r} \cdot (1 + \varepsilon_i) \cdot \left(3 - m \cdot \hat{r}^2 \cdot (1 + \varepsilon_i)^2\right) - \hat{r}}{\hat{r}} \\
&= \frac{1}{2} \cdot (1 + \varepsilon_i) \cdot \left(3 - m \cdot \frac{1}{m} \cdot (1 + \varepsilon_i)^2\right) - 1 \\
&= \frac{1}{2} \cdot (1 + \varepsilon_i) \cdot \left(3 - 1 - 2 \cdot \varepsilon_i - \varepsilon_i^2\right) - 1 \\
&= (1 + \varepsilon_i) \cdot \left(1 - \varepsilon_i - \frac{1}{2} \cdot \varepsilon_i^2\right) - 1 \\
&= 1 - \varepsilon_i - \frac{1}{2} \cdot \varepsilon_i^2 + \varepsilon_i - \varepsilon_i^2 - \frac{1}{2} \cdot \varepsilon_i^3 - 1 \\
&= -\frac{3}{2} \cdot \varepsilon_i^2 - \frac{1}{2} \cdot \varepsilon_i^3
\end{aligned}$$

So the accuracy of the approximation of the reciprocal square root is doubled at each step.

Since the initial accuracy is about 8 bits, it is possible to iterate two times on pure double precision without any considerable loss of accuracy. After the two iterations about 31 bits will be correct. The macro implements therefore:

Listing 2.14: Newton iteration - double precision steps

```
1 r1 = 0.5 * r0 * (3 - m * (r0 * r0));
2 r2 = 0.5 * r1 * (3 - m * (r1 * r1));
```

For these two iterations, 8 double precision multiplications and 2 additions are needed.

The next iteration steps must be performed in double-double precision because the 53 bit mantissa of a double cannot contain the about 60 bit exact value $m \cdot r_2^2 \approx 1$ before cancellation in the substraction with 3 and the multiplication by r_2.

In order to exploit maximally the parallelism in the iteration equation, we rewrite it as

$$\begin{aligned}
r_3 &= \frac{1}{2} \cdot r_2 \cdot \left(3 - m \cdot r_2^2\right) \\
&= \left(r_2 + \frac{1}{2} \cdot r_2\right) - \frac{1}{2} \cdot (m \cdot r_2) \cdot (r_2 \cdot r_2)
\end{aligned}$$

Since multiplications by integer powers of 2 are exact, it is possible to compute $r_2 + \frac{1}{2} \cdot r_2$ exactly as a double-double. Concurrently it is possible to compute $m \cdot r_2$ and $r_2 \cdot r_2$ exactly as double-doubles by means of an exact multiplication. The multiplication $(m \cdot r_2) \cdot (r_2 \cdot r_2)$ is then implemented as a double-double multiplication. The multiplication by $\frac{1}{2}$ of the value obtained is exact and can be performed pairwise on the double-double. A final double-double addition leads to $r_3 = \left(r_2 + \frac{1}{2} \cdot r_2\right) - \frac{1}{2} \cdot (m \cdot r_2) \cdot (r_2 \cdot r_2)$. Here, massive cancellation is no longer possible since the values added are approximately $\frac{3}{2} \cdot r_2$ and $\frac{1}{2} \cdot r_2$.

These steps are implemented as follows:

Listing 2.15: Newton iteration - first double-double step

```
Mul12(&r2Sqh, &r2Sql, r2, r2);     Add12(r2PHr2h, r2PHr2l, r2, 0.5 * r2);
Mul12(&mMr2h, &mMr2l, m, r2);
Mul22(&mMr2Ch, &mMr2Cl, mMr2h, mMr2l, r2Sqh, r2Sql);

MHmMr2Ch = -0.5 * mMr2Ch;
MHmMr2Cl = -0.5 * mMr2Cl;

Add22(&r3h, &r3l, r2PHr2h, r2PHr2l, MHmMr2Ch, MHmMr2Cl);
```

The next iteration step provides enough accuracy for a double-double result. We rewrite the basic iteration equation once again as:

$$\begin{aligned} r_4 &= \frac{1}{2} \cdot r_3 \cdot \left(3 - m \cdot r_3^2\right) \\ &= r_3 \cdot \left(\frac{3}{2} - \frac{1}{2} \cdot m \cdot r_3^2\right) \\ &= r_3 \cdot \left(\frac{3}{2} - \frac{1}{2} \cdot \left(\left(m \cdot r_3^2 - 1\right) + 1\right)\right) \\ &= r_3 \cdot \left(1 - \frac{1}{2} \cdot \left(m \cdot r_3^2 - 1\right)\right) \end{aligned}$$

Further, we know that r_3, stored as a double-double, verifies $r_3 = \hat{r} \cdot (1 + \varepsilon_3)$ with $|\varepsilon_3| \leq 2^{-60}$. So we check that

$$m \cdot r_3^2 = m \cdot \hat{r}^2 \cdot (1 + \varepsilon_3)^2 = 1 + 2 \cdot \varepsilon_3 + \varepsilon_3^2$$

Clearly, $\left|2 \cdot \varepsilon_3 + \varepsilon_3^2\right| < \frac{1}{2}\text{ulp}(1)$. So when squaring $r_{3h} + r_{3l}$ in double-double precision and multiplying it in double-double precision by m produces a double-double $mMr3Sq_h + mMr3Sq_l = m \cdot (r_{3h} + r_{3l})^2 \cdot (1 + \varepsilon)$, $|\varepsilon| \leq 2^{-100}$ such that $mMr3Sq_h = 1$ in all cases.

So we can implement the iteration equation

$$r_4 = r_3 \cdot \left(1 - \frac{1}{2} \cdot \left(m \cdot r_3^2 - 1\right)\right)$$

as follows:

Listing 2.16: Newton iteration - second double-double step

```
Mul22(&r3Sqh, &r3Sql, r3h, r3l, r3h, r3l);
Mul22(&mMr3Sqh, &mMr3Sql, m, 0, r3Sqh, r3Sql);

Mul22(&r4h, &r4l, r3h, r3l, 1, -0.5 * mMr3Sql);
```

We since get $r_{4h} + r_{4l} = \hat{r} \cdot (1 + \varepsilon_4)$ with $|\varepsilon_4| \leq 2^{-102}$, the accuracy being limited by the accuracy of the last double-double multiplication operator.

This approximation is than multiplied by m in double-double precision, leading to an approximation $srtm_h + srtm_l = \sqrt{m} \cdot (1 + \varepsilon)$ with $|\varepsilon| \leq 2^{-100}$.

Out of this value, the square root of the initial argument can be reconstructed by multiplying by $2^{\frac{E}{2}}$, which has already been stored in *xdb.d*. This multiplication is exact because it cannot produce a subnormal.

These two steps are implemented as shown below:

Listing 2.17: Multiplication $m \cdot \hat{r}$, reconstruction

```
Mul22(&srtmh,&srtml,m,0,r4h,r4l);

/* Multiply componentwise by sqrt(2^E), which is an integer power of 2 that may not produce a
   subnormal */

*resh = xdb.d * srtmh;
*resl = xdb.d * srtml;
```

Triple-double Newton iteration

For producing a triple-double approximate to $\hat{r}$ with an accuracy of at least 147 bits, one more Newton iteration is needed. We apply the same equation as in the last double-double step, which reads:

$$r_5 = r_4 \cdot \left(1 - \frac{1}{2} \cdot \left(m \cdot r_4^2 - 1\right)\right)$$

Once again, the first component of the triple-double number holding an approximation to $m \cdot r_4^2$ is exactly equal to 1. So by neglecting this component, we substract 1 from it. Unfortunately, a renormalization step is needed after the multiplications for squaring r_4 and by m because the values computed might be overlapped which would prevent us form substracting 1 by neglecting a component.

We implement thus:

Listing 2.18: Newton iteration - triple-double step

```
Mul23(&r4Sqh, &r4Sqm, &r4Sql, r4h, r4l, r4h, r4l);
Mul133(&mMr4Sqhover, &mMr4Sqmover, &mMr4Sqlover, m, r4Sqh, r4Sqm, r4Sql);
Renormalize3(&mMr4Sqh, &mMr4Sqm, &mMr4Sql, mMr4Sqhover, mMr4Sqmover, mMr4Sqlover);

HmMr4Sqm = -0.5 * mMr4Sqm;
HmMr4Sql = -0.5 * mMr4Sql;

Mul233(&r5h,&r5m,&r5l,r4h,r4l,1,HmMr4Sqm,HmMr4Sql);
```

This approximation $r_{5h} + r_{5m} + r_{5l} = \hat{r} \cdot (1 + \varepsilon_5)$, where $|\varepsilon_5| \leq 2^{-147}$ is then multiplied by m in order to obtain a triple-double approximation of $\sqrt{m}$. Once renormalized result is exactly multiplied by $2^{\frac{E}{2}}$ stored in $xdb.d$. We implement:

Listing 2.19: Newton iteration - triple-double step

```
Mul133(&srtmhover, &srtmmover, &srtmlover,m,r5h,r5m,r5l);

Renormalize3(&srtmh,&srtmm,&srtml,srtmhover,srtmmover,srtmlover);

(*(resh)) = xdb.d * srtmh;
(*(resm)) = xdb.d * srtmm;
(*(resl)) = xdb.d * srtml;
```

Accuracy bounds

TODO: see possibly available Gappa files meanwhile

2.7 Test if rounding is possible

We assume here that an evaluation of $y = f(x)$ has been computed with a total relative error smaller than $\bar{\varepsilon}$, and that the result is available as the sum of two non-overlapping floating-point numbers y_h and y_l (as is the case if computed by the previous algorithms). This section gives and proves algorithms for testing if y_h is the correctly rounded value of y according to the relative error $\bar{\varepsilon}$. This correspond to detect whether we are in a hard to round case.

2.7.1 Rounding to the nearest

Theorem 21 (Correct rounding of a double-double to the nearest double, avoiding subnormals)**.**
Let y be a real number, and $\bar{\varepsilon}$, e, y_h and y_l be double-precision floating-point numbers such that

- $y_h = y_h \oplus y_l$
- *none of y_h and y_l is a NaN or $\pm\infty$,*
- $|y_h| \geq 2^{-1022+54}$ *(i.e. $\frac{1}{4}ulp(y_h)$ is not subnormal),*
- $|y_h + y_l - y| < \bar{\varepsilon}.|y|$ *(i.e. the total relative error of $y_h + y_l$ with respect to y is bounded by $\bar{\varepsilon}$),*

- $0 < \bar{\varepsilon} \leq 2^{-53-k}$ *with* $k \geq 3$ *integer,*

- $e \geq (1-2^{-53})^{-1}\left(1+\dfrac{2^{54}\bar{\varepsilon}}{1-\bar{\varepsilon}-2^{-k+1}}\right)$ *and* $e \leq 2$.

The following test determines whether y_h is the correctly rounded value of y in round to nearest mode.

Listing 2.20: Test for rounding to the nearest

```
if( y_h == (y_h + (y_l*e)) )
  return y_h;
else /* more accuracy is needed, launch accurate phase */
```

Proof. Remark that the condition $|y_h| \geq 2^{-1022+54}$ implies that y_h is a normal number.

The implication we need to prove is: if the test is true, then $y_h = \circ(y)$ (failure of the test does not necessary mean that $y_h \neq \circ(y)$).

Let us note $u = \text{ulp}(y_h)$ and consider only the case when y_h is positive (as the other case is symmetrical).

We have to consider separately the following two cases.

If y_h is not a power of two or $y_l \geq 0$

In this case we will always assume that $y_l \geq 0$, as the case $y_l \leq 0$ is symmetrical when y_h is not a power of two.

To prove that $y_h = \circ(y)$, it is enough to prove that $|y_h - y| \leq u/2$. As $|y_h + y_l - y| < \bar{\varepsilon}.|y|$ (fourth hypothesis) it is enough to prove that $u/2 - y_l > \bar{\varepsilon}y$.

By definition of the ulp of a positive normal number, we have $y_h \in [2^{52}u, (2^{53}-1)u]$.

From the first hypothesis we have

$$y_l \leq \frac{1}{2}u \tag{2.5}$$

Therefore $y_h + y_l \leq (2^{53}-1)u + \frac{1}{2}u$, and $y < (y_h + y_l)/(1-\bar{\varepsilon})$. Hence

$$y < \frac{2^{53}-\frac{1}{2}}{1-\bar{\varepsilon}}u$$

As a consequence, since $\bar{\varepsilon} \leq 2^{-56}$,

$$y < 2^{53}u \tag{2.6}$$

The easy case is when we have $y_h = \circ(y)$ regardless of the result of the test. This is true as soon as y_l is sufficiently distant from $u/2$. More specifically, if $0 \leq y_l < \left(\frac{1}{2} - 2^{-k}\right)u$, we combine (2.6) with the fifth hypothesis to get $\bar{\varepsilon}y < 2^{-k}u$. From $y_l < \left(\frac{1}{2} - 2^{-k}\right)u$ we deduce $u/2 - y_l > 2^{-k}u > \bar{\varepsilon}y$, which proves that $y_h = \circ(y)$.

If y_h is a power of two and $y_l < 0$

To prove that $y_h = \circ(y)$, it is enough to prove that $|y_h - y| \leq u/4$. As $y_l \leq 0$, we have $y \leq y_h$ and $|y_h - y| = y - y_h$.

From fourth hypothesis, it is enough to prove that $u/4 + y_l > \bar{\varepsilon}y$.

By our definition of the ulp of a normal number, we have $y_h = 2^{52}u$ in this case.

We have $y_h = 2^{52}u$ and $y_l \leq 0$, therefore $y_h + y_l \leq 2^{52}u$, and

$$y < \frac{2^{52}u}{1-\bar{\varepsilon}} \tag{2.7}$$

The easy case is when we have $y_h = \circ(y)$ regardless of the result of the test. This is true as soon as y_l is sufficiently distant from $-u/4$. More specifically, if $-\left(\frac{1}{4} - \frac{2^{-k-1}}{1-\bar{\varepsilon}}\right)u < y_l \leq 0$, after combining (2.7) with the fifth hypothesis to get $\bar{\varepsilon}y < \frac{2^{-k-1}u}{1-\bar{\varepsilon}}$, we deduce $y_l + \frac{u}{4} > \frac{2^{-k-1}}{1-\bar{\varepsilon}}u > \bar{\varepsilon}y$, which proves that $y_h = \circ(y)$.

Now consider the case when $y_l \geq \left(\frac{1}{2} - 2^{-k}\right) u$. The condition $|y_h| \geq 2^{-1022+54}$ ensures that $u/4$ is a normal number, and now $y_l > u/4$, so in this case y_l is a normal number. As $1 < e \leq 2$, the result is also normal, therefore

$$y_l \times e(1 - 2^{-53}) \leq y_l \otimes e \leq y_l \times e(1 + 2^{-53})$$

Suppose that the test is true ($y_h \oplus y_l \otimes e = y_h$). With IEEE-54 compliant rounding to nearest, this implies $|y_l \otimes e| \leq \frac{u}{2}$, which in turn implies $y_l \times e(1 - 2^{-53}) \leq \frac{u}{2}$ (as y_l is a normal number and $1 < e \leq 2$). This is rewritten

$$\frac{u}{2} - y_l \geq y_l \left(e\left(1 - 2^{-53}\right) - 1\right)$$

Using $y_l \geq (\frac{1}{2} - 2^{-k})u$, we get

$$\frac{u}{2} - y_l \geq \left(\frac{1}{2} - 2^{-k}\right) u \left(e\left(1 - 2^{-53}\right) - 1\right)$$

We want to ensure that $\frac{u}{2} - y_l \geq \bar{\varepsilon} y$, we will again use (2.6) and ensure that $\frac{u}{2} - y_l \geq 2^{53}\bar{\varepsilon}u$. This provides the condition that must be fullfilled by e for the theorem to hold in this case: we need

$$\left(\frac{1}{2} - 2^{-k}\right) u \left(e(1 - 2^{-53}) - 1\right) \geq 2^{53}\bar{\varepsilon}u$$

rewritten as:

$$e \geq (1 - 2^{-53})^{-1} \left(1 + \frac{2^{54}\bar{\varepsilon}}{1 - 2^{-k+1}}\right)$$

Now consider the case when $-y_l \geq \left(\frac{1}{4} - \frac{2^{-k-1}}{1-\bar{\varepsilon}}\right) u$. The condition $|y_h| \geq 2^{-1022+54}$ ensures that $u/8$ is a normal number, and now $y_l > u/8$, so in this case y_l is a normal number. As $1 < e \leq 2$, the result is also normal, therefore

$$-y_l \times e(1 - 2^{-53}) \leq -y_l \otimes e \leq -y_l \times e(1 + 2^{-53})$$

Suppose that the test is true ($y_h \oplus y_l \otimes e = y_h$). For this value of y_h and this sign of y_l, this implies $|y_l \otimes e| \leq \frac{u}{4}$, which in turn implies $-y_l \times e(1 - 2^{-53}) \leq \frac{u}{4}$.
This is rewritten

$$\frac{u}{4} + y_l \geq -y_l \left(e\left(1 - 2^{-53}\right) - 1\right)$$

Using $-y_l \geq \left(\frac{1}{4} - \frac{2^{-k-1}}{1-\bar{\varepsilon}}\right) u$, we get

$$\frac{u}{4} + y_l \geq \left(\frac{1}{4} - \frac{2^{-k-1}}{1-\bar{\varepsilon}}\right) u \left(e\left(1 - 2^{-53}\right) - 1\right)$$

To ensure that $\frac{u}{4} + y_l \geq \bar{\varepsilon} y$, we again use (2.7) and ensure that $\frac{u}{4} + y_l \geq \frac{2^{52}u}{1-\bar{\varepsilon}}\bar{\varepsilon}$. This provides the condition that must be fullfilled by e for the theorem to hold in this case: we need

$$\left(\frac{1}{4} - \frac{2^{-k-1}}{1-\bar{\varepsilon}}\right) u \left(e(1 - 2^{-53}) - 1\right) \geq \frac{2^{52}u}{1-\bar{\varepsilon}}\bar{\varepsilon}$$

rewritten as:

$$e \geq (1 - 2^{-53})^{-1} \left(1 + \frac{2^{54}\bar{\varepsilon}}{1 - \bar{\varepsilon} - 2^{-k+1}}\right)$$

Taking for contraint on e the max of these values completes the proof of the theorem.

□

Notes

- In general we will target values of $\bar{\varepsilon}$ in the order of 2^{-53-10} to balance the execution times of the quick and accurate phases.
- A similar theorem could be written for y_h subnormal. In most cases, there will be a property such as $\circ(f(x)) = x$, deduced from the Taylor theorem. For the rare functions that come close to zero without such a property (an example is `exp`), it is simpler and safer to launch the accurate phase systematically in this case.
- These theorems are not proven for $y_h = \pm\infty$ (an implementation would depend on the correct behaviour of the double-double arithmetic in the neighborhood of $\pm\infty$ anyway). This is not a problem in practice, because an implementation will fall into one of the following cases:
  - It can be proven statically that the function is bounded well below the largest representable double-precision number. This will be the case of the logarithm and tangent functions in their respective chapters.
  - The function comes close to infinity, but monotonicity or another mathematical property allows to prove that $\pm\infty$ should be returned for x above or below some statically-defined

threshold, and never otherwise. This will be the case of exponential and hyperbolic functions, for instance.

In both cases, returning a value close to infinity won't require a rounding test.

- A fused multiply-and-add should probably not be used for the computation of $y_h + y_l \times e$. Studying this question is on the TODO list.

2.7.2 Directed rounding modes

Directed rounding is much easier to achieve than round to the nearest: The difficult cases are the cases when the exact value y is very close to a machine number, and we have in $y_h + y_l$ an approximation to y with a relative error smaller than the half-ulp of y_h. Therefore we have in $|y_l|$ an approximation to the distance of y_h to the closest machine number, with a known approximation error.

We use in `crlibm` the following macro which does the test and then the rounding. It should be used as follows (example taken from `atan_fast.c`):

Listing 2.21: An occurence of the test for rounding up

```
1  TEST_AND_RETURN_RU(atanhi, atanlo, maxepsilon);
2  /* if the previous block didn't return a value, launch accurate phase */
3  return scs_atan_ru(x);
```

Theorem 22 (Test for correct rounding up of a double-double to a double)**.**
Let y be a real number, and y_h, y_l and $\bar{\varepsilon}$ be floating-point numbers such that

- $y_h = y_h \oplus y_l$,
- *y_h is neither a NaN, a subnormal, ± 0 or $\pm\infty$.*
- *y_l is neither a NaN or $\pm\infty$.*
- $|y_h + y_l - y| < \bar{\varepsilon}.|y|$

The following test determines whether y_h is the correctly rounded value of y in round up mode.

Listing 2.22: Test for directed rounding

```
1  #define TEST_AND_RETURN_RU(__yh__, __yl__, __eps__)                          \
2  {                                                                            \
3    db_number yh, yl, u53;  int yh_neg, yl_neg;                                \
4    yh.d = __yh__;    yl.d = __yl__;                                           \
5    yh_neg = (yh.i[HI] & 0x80000000);                                          \
6    yl_neg = (yl.i[HI] & 0x80000000);                                          \
7    yh.l = yh.l & 0x7fffffffffffffffLL;  /* compute the absolute value*/       \
8    yl.l = yl.l & 0x7fffffffffffffffLL;  /* compute the absolute value*/       \
9    u53.l     = (yh.l & 0x7ff0000000000000LL) +  0x0010000000000000LL;        \
10   if(yl.d > __eps__ * u53.d){                                                \
11     if(!yl_neg) {  /* The case yl==0 is filtered by the above test*/        \
12       /* return next up */                                                  \
13       yh.d = __yh__;                                                        \
14       if(yh_neg) yh.l--;  else yh.l++; /* Beware: Fails for zero */         \
15       return yh.d ;                                                         \
16     }                                                                       \
17     else  return __yh__;                                                    \
18   }                                                                         \
19 }
```

Proof. The first lines compute $|y_h|$, $|y_l|$, boolean values holding the sign information of y_h and y_l, and $u_{53} = 2^{53}\text{ulp}(y_h)$. Here we use integer 64-bit arithmetic for readability, but other implementations may be more efficient on some systems. Note that these computations don't work for infinities, zeroes or subnormals.

As previously, by definition of the ulp, we have $y < 2^{53}u$.

The main test which determines whether correct rounding is possible is line 10. If this test is true, then $y_l > (2^{53}u) \otimes \bar{\varepsilon} = 2^{53}\bar{\varepsilon}u$ (the multiplication by u_{53}, a power of two, is exact), hence $y_l > \bar{\varepsilon}y$ so we are in an easy case for directed rounding.

The remaining computations and tests (lines 11 and following) compute `nextafter(yh,inf)` in an efficient way since an integer representation of y_h is already available. For the other directed rounding modes, only these lines change in a straightforward way. □

Notes

- Rounding down and to zero are identical to the previous, except for the computation of the rounded value itself.
- These tests launch the accurate phase when yl=0, in particular in the exceptional cases when the image of a double is a double. See the chapter 3 for an example where it may introduce a misround.
- These tests don't work if y_h is a subnormal. If one cannot prove statically that this case doesn't appear, a sensible solution is to test for subnormals and launch the accurate phase.
- Finally, remark that for some functions, the tests on the sign of y_h are statically predictable to be true because the function is always positive. We shall use this macro anyway for safety. Tanks to branch predictor logic in modern processors, it will make little difference from a performance point of view.

2.8 The Software Carry Save library

The software carry-save internal representation of multiple-precision numbers was designed specifically for simple and fast implementations of addition and multiplication in the 100-500 bit precision range, as required by the accurate phase of our algorithms. More details on software carry-save are available in [14, 10].

The parameters of `scslib` are set up so that all the operators offer a relative error better than 2^{-208}. This is a large overkill for all the functions in `crlibm`, as the worst cases computed by Lefevre never require more than 158 bits of accuracy. This enables simple proofs for the second steps, assuming the operators in `scslib` are correct.

Another feature that makes accuracy proofs simple when using `scslib` is the following: The range of SCS numbers includes the range of IEEE double-precision numbers, including subnormals and exceptional cases. Conversions between SCS format and IEEE-754 doubles, as well as arithmetic operations, follow the IEEE rules concerning the exceptional cases. SCS doesn't ensure correct rounding, but provides conversions to doubles in the four IEEE-754 rounding modes, which is enough for the purpose of `crlibm`.

However, a formal proof of correctness of the `scslib` operators remains to be done. Currently there is nothing more than good confidence based on the simplicity of the code.

2.8.1 The SCS format

A MP number is represented in the proposed format as a *Software Carry Save* (SCS) structure R, depicted on Figure 2.1 and composed of the following fields:

R.digits[n_r] A table of n_r digits with m_r bits of precision. These digits can in principle be either integer or FP machine numbers, however integer is always faster and simpler. We will not mention FP digits anymore here, the interested reader is referred to [14, 10].

R.index An integer storing the index of the first digit in the range of representable numbers, as depicted on Figure 2.1;

R.sign A sign information.

In other words, the value x of a representation R is:

$$x = R.sign \times \sum_{j=1}^{n_r} R.digits[j] \times 2^{m_r*(R.index-j)} \tag{2.8}$$

In such a *normal* SCS number R, the bits from m_I to m_r of the $R.digits$ fields are thus set to zero. They will be exploited by the algorithms to store temporary *carry* information, and are therefore called *carry-save* bits. An SCS number where these bits are non-zero is said to be non-normal.

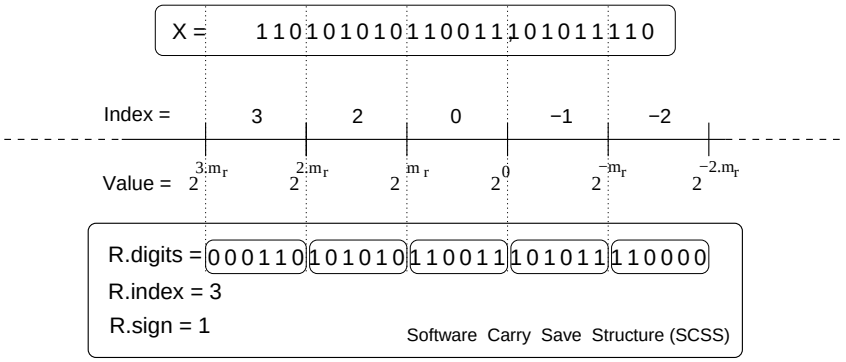


Figure 2.1: The proposed format

The values of the parameters for use in `crlibm` is $n_r = 8$ digits of $m_i = 30$ bits stored on $m_r = 32$-bit words. The worst-case precision that this format may hold is when the most significant digit is equal to 1, meaning that an SCS numbers holds only $1 + 7 \times 30 = 211$ significant digits.

2.8.2 Arithmetic operations

Conversion from double to SCS

A first method for converting a double precision floating point number d into an SCS representation is to extract the exponent d_{exp} from d, and then determine the corresponding $R.index$ as the integer part of $\frac{d_{exp}}{2^{m_r}}$.

Another method uses a variable number of multiplications by 2^{m_r} or 2^{-m_r}. This method is faster than the previous one when the exponent of d is close to 0.

After testing both methods in `crlibm`, the first method was preferred.

Addition and subtraction

The addition of two SCS numbers of the same sign consists in aligning, then adding digits of the same order. Thanks to the carry-save bits, all these additions will be *exact* and *independent*. However the result will usually not be a normal SCS number: the sums will have overflown in the carry-save bits. A *renormalization* procedure is presented in section 2.8.2 to propagate these carry bits and get again a normal SCS number. However, the advantage of SCS representation is that many SCS numbers can be summed before needing to perform this expensive step (up to 7 with the choice of parameters made in `crlibm`).

The subtraction (addition of two numbers of opposite signs) is very similar to the addition algorithm. It may also classically lead to a cancellation, which may need an update of the index of the result. However, as in other floating-point formats, a subtraction involving a a cancellation is exact.

Although all the digit operations are exact, the addition or subtraction of two numbers also classically involves a rounding error, due to aligning the digits of same magnitude. For performance reason this rounding is a truncation, so the worst-case relative error is one ulp of the least accurate representable number, or 2^{-211}.

Multiplication

The multiplication of two normal SCS numbers involves the operations depicted on the Figure 2.2: The partial products are computed (in parallel) and summed in columns. The parameters are set up so that none of these operation overflow. Again, the result is not a normal SCS number, and a renormalization procedure (described below) has to be applied to empty the carry bits. However, a few additions may follow a multiplication before this renormalization, which allows for further optimization of algorithms using SCS arithmetic. For instance, a polynomial evaluation can be implemented with a renormalization after one multiplication and one addition.

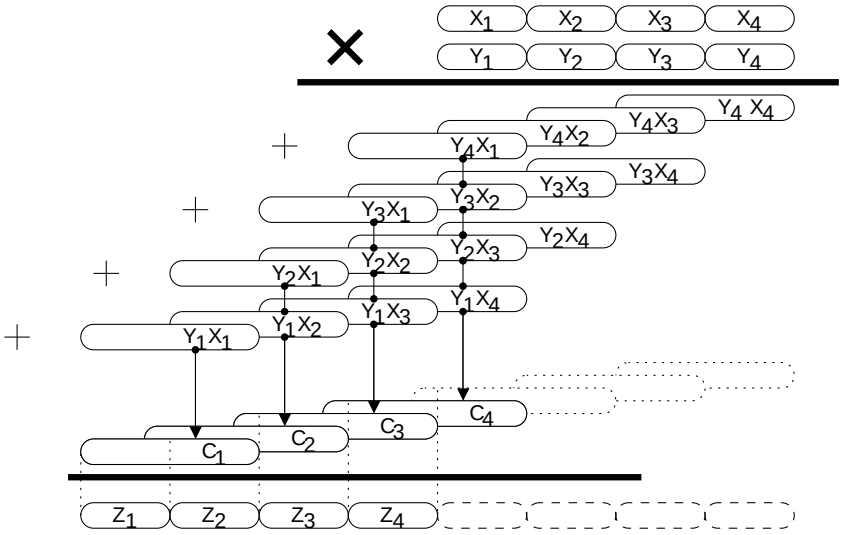


Figure 2.2: SCS multiplication

Here also, a rounding error is involved when two n_r-digit numbers are multiplied if the result is to fit on n_r digits. The actual implementation tests if the most significant digit (z_1 on Figure 2.2) is null, in which case the index of the result is that of z_2.

If the whole of the computations of Figure 2.2 are implemented, the worst case for relative accuracy is again 2^{-211}. However a further optimization is to avoid computing the columns of lower magnitude, at the expense of an increase in the rounding error. More specifically, we compute 9 columns instead of 16. The wors case is now when z_1 is null, in which case the relative error correspond to the truncation of the 8 leftmost columns, whose maximum value is smaller than 3 ulps of the SCS result. Therefore the relative error of the multiplication is bounded by 2^{-208} with this optimization, which is still a large overkill for the purpose of `crlibm`.

This optimization is therefore implemented if the loop are hand-unrolled. If they are not, the increased control complexity actually degrades performance.

Renormalization (carry propagation)

Renormalization is a carry propagation from the low order to high order digits: Starting with an initially null carry, at each step, the previous carry is added to the current digit, and this sum is then split into two parts using masks. The low m_r bits are a digit of the normalized result, and the upper part is the next carry.

The actual algorithm is a little bit more complex. The initial non-normal number may not be representable exactly as a normal SCS number, therefore the index of the normalized result may have to be increased by one or two. Normalization thus again involves a rounding error. Note that this error was already taken into account in the previous discussions of addition and multiplication.

Conversion from SCS to floating-point

A few (4 in the worst case) multiplications and additions suffice to get the FP number closest to a SCS number. For instance, for $m_I = 53$ and $m_r = 26$, we need to compute $d = A.sign \times 2^{A.index \times m_r} \times (A.digits[0] + 2^{-m_r} \times A.digits[1] + 2^{-2.m_r} \times A.digits[2] + 2^{-3.m_r} \times A.digits[3])$. The number $2^{A.index \times m_r}$ is build using integer masks. The actual implementation of this formula is slightly less simple, but this conversion is still very fast.

Mixed 32- and 64-bit arithmetics

An improvement implemented in `scslib` was the combined use of integer 32- and 64-bit arithmetics as follows:

- MP digits are stored as 32-bit numbers where only a few bits are reserved for carries. This removes the main problem of the initial implementation [14], namely its memory inefficiency.
- Addition uses 32-bit arithmetic.
- In the MP multiplication, the partial products are products of two 32-bit digits, which are 64-bit numbers. The column sums need thus to be computed using 64-bit arithmetic. This can be expressed in the C language in a non-ISO-C99, but de-facto standard way, as follows: 32-bit numbers have the `unsigned int` type; 64-bit numbers have the `unsigned long long int` type. When multiplying two digits, one is first cast into this 64-bit type.

  For UltraSPARC architectures (detected at build time) the conversion is to floating-point, but we will not detail this peculiarity further.

This works well because all modern processors either have 64-bit integer units, or offer instructions which store the 64-bit product of two 32-bit integers into two 32-bit registers. The compiler does the rest well, because it is conceptually simple: casting unsigned 32-bit into unsigned 64-bit is trivial; 64-bit addition is translated straightforwardly into one 32-bit *add* followed by one 32-bit *add-with carry*.

Implementation considerations

For portability purposes, the implemention uses C as defined by the ISO C99 standard, and tries to use a recent version of `gcc`. We could not exhibit a case where a native compiler from the processor vendor (Intel or Sun) gave significantly better results than `gcc`, which is probably a consequence of the simplicity of our code.

However, when tuning for performance, we observed that the same code which was efficient on one processor could lead to very poor results on another. Usually, this difference can be traced down to the capabilities of the processor itself. The typical example is the knowingly poor integer multiplication on UltraSPARC II. Sometimes however, the processor should be able to perform well, and it is the processor-specific backend of the compiler which is to blame, which can be checked by observing the assembly code produced. A typical example is the casting of 32-bits digits to 64-bit arithmetic (or to an FP number in the case of the UltraSPARC) in the multiplication algorithm. In these cases we tried to change the programming style in a way that works well on all processors. Sometimes it wasn't possible, in which case the code contains, along with a generic version, several processor-specific tricky versions of the problematic operation, selected at compile time thanks to the GNU `automake/autoconf` tools.

More surprisingly, we were disappointed by the higher-level capabilities of the compilers, especially at unrolling loops. Our code exhibits many small `for` loops whose size is known at compile-time (usually n). This is the ideal situation for loop unrolling, a technique well known and described in most textbooks on compiler design. Options exist in most compilers to turn on this optimisation. Unfortunately, leaving loop unrolling to the compiler gives very poor results, even when compared to the non-unrolled case. Since unrolling the loops by hand in the C code takes a few minutes, we did it for the version of the library which we use ($m = 30$, $n = 8$). It marginally increases the code sizes for this small n, and sometimes provides a twofold improvement on speed, depending of the processor. Of course, this is not satisfactory: We don't want to do it for all values of n, nor do we want to study for each processor the tradeoffs involved as n increase. We expect however future compilers to handle unrolling better, and we were surprised that no compiler had a clear edge on the other in this respect. Some argue, however, that this issue is pointless, as superscalarity, along with register renaming and branch prediction inside modern processors, sum up to the equivalent of dynamic unrolling of the code. In our tests (in 2003), it doesn't: unrolling does bring a speed-up.

2.9 Common Maple procedures

2.9.1 Conversions

Procedure `ieeedouble` returns the sign, the exponent and the mantissa of the IEEE-754 double-precision number closest to input value `x`.

Listing 2.23: ieeedouble

```
ieeedouble:=proc(xx)
local x, sgn, logabsx, exponent, mantissa, infmantissa,powermin,powermax,expmin,expmax,
    expmiddle,powermiddle;
Digits := 100;
x := evalf(xx);
if (x=0) then sgn, exponent, mantissa := 1, -1022, 0
else
  if (x < 0) then sgn := -1
  else sgn := 1
  fi:
  x := abs(x);
  if x >=  2^(1023)*(2-2^(-53)) then mantissa := infinity; exponent := 1023
  else if x <= 2^(-1075) then mantissa := 0; exponent := -1022
      else
         if x <= 2^(-1022) then exponent := -1022
         else
# x is between 2^(-1022) and 2^(1024)
         powermin := 2^(-1022); expmin := -1022;
         powermax := 2^1024; expmax := 1024;
         while (expmax-expmin > 1) do
            expmiddle := round((expmax+expmin)/2);
            powermiddle := 2^expmiddle;
            if x >= powermiddle then
                powermin := powermiddle;
                expmin := expmiddle
            else
                powermax := powermiddle;
                expmax := expmiddle
            fi
          od;
# now, expmax - expmin = 1 and powermin <= x < powermax,
# powermin = 2^expmin and powermax = 2^expmax, so expmin is the exponent of x
         exponent := expmin;
         fi;
         infmantissa := x*2^(52-exponent);
         if frac(infmantissa) <> 0.5 then mantissa := round(infmantissa)
            else
              mantissa := floor(infmantissa);
               if type(mantissa,odd) then mantissa := mantissa+1 fi
            fi;
         mantissa := mantissa*2^(-52);
      fi;
  fi;
fi;
sgn,exponent,mantissa;
end:
```

Procedure `ieeehexa` returns the hexadecimal representation of the nearest double to its input `x`.

Listing 2.24: ieeehexa

```
ieeehexa:= proc(x)
local  hex2, xx, longint, expo, sgn, frac, resultat;
    if(x=0) then resultat:=["00000000","00000000"];
    elif(x=-0) then resultat:=["80000000","00000000"]; # nice try
    else
        xx:=ieeedouble(x);
        sgn:=xx[1]:
        expo:=xx[2]:
        frac:=xx[3]:
        if (expo = -1023) then
            longint := (frac)*2^51 ;   # subnormal
        else
            longint := (frac-1)*2^52 +   (expo+1023)*2^52;
        fi:
        if (sgn=-1) then
            longint := longint + 2^63;
        fi:
        longint := longint + 2^64:  # to get all the hexadecimal digits when we'll convert to
            string
        hex2:=convert(longint, hex);
        hex2:=convert(hex2, string):

        resultat:=[substring(hex2,2..9), substring(hex2,10..18)]:
    fi:
    resultat;
end proc:
```

Procedure `hexa2ieee` performs the reciprocal conversion.
Procedure `hi_lo` returns two IEEE-double numbers *x_hi* and *x_lo* so that $x = x\_hi + x\_lo + \varepsilon_{-103}$.

Listing 2.25: hi_lo

```
hi_lo:= proc(x)
local x_hi, x_lo, res:
x_hi:= nearest(evalf(x)):
res:=x-x_hi:
if (res = 0) then
  x_lo:=0:
else
  x_lo:=nearest(evalf(res)):
end if;
x_hi,x_lo;
end:
```

Procedure `showHowDifficultToRound` takes a real number, and prints the bits after the 53th of its nearest IEEE floating-point number.

Listing 2.26: showHowDifficultToRound

```
showHowDifficultToRound:=proc(x)
local xb,xs,s,e,m:
    Digits:=200:
    s,e,m := ieeedouble(x):
    xb:=convert(evalf(x*2^(-e)),binary):
    xs:=convert(xb, string):
    substring(xs,55..153)
end proc:
```

2.9.2 Procedures for polynomial approximation

Procedure `Poly_exact2` takes in arguments a polynomial `P` and a integer `n`. It returns a truncated polynomial, of which coefficients are exactly IEEE-double numbers. The `n` first coefficients are written over 2 IEEE-double numbers.

Listing 2.27: poly_exact2

```
poly_exact2:=proc(P,n)
local deg,i, coef, coef_hi, coef_lo, Q:
Q:= 0:
convert(Q, polynom):
deg:=degree(P,x):
  for i from 0 to deg do
    coef :=coeff(P,x,i):
    coef_hi, coef_lo:=hi_lo(coef):
    Q:= Q + coef_hi*x^i:
    if(i<n) then
        Q := Q + coef_lo*x^i:
    fi:
  od:
  return(Q);
end:
```

We also have procedures for computing good truncated polynomial approximation for a function. As they are useless to the proof, we do not describe them here, the interested reader is referred to file `maple/common-procedures.mpl` for more details.

2.9.3 Accumulated rounding error in Horner evaluation

The following Maple procedures implement the error analysis described in Section 2.5.

Procedure `compute_abs_rounding_error` computes a bound on the accumulated rounding error caused by the Horner evaluation of a truncated polynomial. `poly` is the polynomial, `xmax` is the max value of $|x|$, `nn` is the degree when `poly` is computed in double double, and the first double-double operation is an addition.

This procedure returns the maximum absolute error, and safe bounds on the minimum and maximum values of the function. It also checks on the fly that the fast (test-free) versions of the double-double addition can be used, and prints warnings if it not the case.

Listing 2.28: compute_abs_rounding_error

```
compute_abs_rounding_error:=proc(poly,xmax, nn)
local n, deg, delta, deltap, i, S, P, Snorm, Smin, Smax, prec:
deltap:=0:
delta:=0:
deg:=degree(poly):

prec:=53; # precision of the first iterations

S:=coeff(poly, x, deg):
Smax:=abs(S):
Smin:=Smax:

if nn<0 then n:=0: else n:=nn: fi:# sometimes called by compute_rel_rounding_error with n=-1

for i from (deg-1) to 0 by -1 do
  P:= convert(S*x, polynom):
  Smin := abs(coeff(poly,x,i)) - xmax*Smax :
  if(Smin<=0) then
    printf("Warning! in compute_abs_rounding_error, Smin<=0 at iteration %d, consider
        decreasing xmax\n",i);
  fi:
  delta:= evalf(xmax*deltap + 2**(-prec)*xmax*Smax):
  if i<n then
    # fast Add22 ?
    if abs(coeff(poly,x,i)) < xmax*Smax  # may be improved to xmax*Smax/2
    then printf("WARNING Add22 cannot be used at step %d, use Add22Cond\n" , i  );
         printf("    coeff=%1.20e,  xmax*Smax=%1.20e"  ,  abs(coeff(poly,x,i)),  xmax*Smax );
    fi:
  fi:
  S:=convert(P+coeff(poly,x,i), polynom):
  Snorm:=evalf(infnorm(S, x=-xmax..xmax)):
  if i=n-1 then prec:=100: fi:  # from the addition of the n-1-th iteration
  deltap:= evalf(delta + 2**(-prec)*(delta + Snorm)):
  Smax := Snorm + deltap:
od:
deltap, Smin, Smax;
end proc:
```

Procedure `compute_rel_rounding_error` computes a bound on the total relative rounding error of a Horner polynomial evaluation, in the same condition as the previous procedure.

Listing 2.29: compute_abs_rounding_error

```
compute_rel_rounding_error:=proc(poly,xmax, n)
local deg, p, rho, deltap, Smin, Smax:

deg:=degree(poly):
if(n>0) then p:=100: else p:=53: fi:

if coeff(poly,x, 0) = 0 then
   deltap, Smin, Smax := compute_abs_rounding_error(poly/x,xmax, n-1):
   rho :=  (2^(-p)*(Smax+deltap) +deltap ) / Smin :
else
   deltap, Smin, Smax := compute_abs_rounding_error(poly,xmax, n):
   rho := deltap /  Smin:
fi:
rho;
end proc:
```

Procedures `compute_abs_rounding_error_firstmult` and `compute_abs_rounding_error_firstmult` are similar to the previous, but in the case when the first double-double operation is a multiplication.

2.9.4 Rounding

Procedure `compute_rn_constant` computes a good constant for the round-to-nearest test of Theorem 21. Its input is a bound of the overall relative error of the approximation scheme.

2.9.5 Using double-extended

The file `maple/double-extended.mpl` contains procedures similar to those previously described to handle double-extended precision (64 bits of mantissa and 15 bits of exponent). This is currently used in experimental code only: The `crlibm` CVS repository at `http://lipforge.ens-lyon.fr/` contains such code for exponential and arctangent on Itanium, and arctangent on IA32 processors. For more details see [11, 12].

Chapter 3

The natural logarithm

There are two versions of the logarithm.

- The first relies on 80-bit double-extended arithmetic, and is well suited to IA32 and IA64 architectures which have hardware support for such arithmetic. It computes the quick step in double-extended arithmetic, and the accurate step in double-double-extended arithmetic.
- The second relies only on double-precision arithmetic, and is portable. It uses double-double for the quick step, and triple-double for the accurate step.

Both implementations use the same algorithm, which is detailed in 3.1. Sections 3.3 and 3.2 detail the proof of both implementations, and 3.4 give some performance results.

3.1 General outline of the algorithm

The algorithm used is mainly due to Wong and Goto[39] and has been discussed further in [34]. In the case we are given here, both quick and accurate phase use principally the same algorithm however optimized for different accuracies.

The function's argument $x \in \mathbb{F}$ is first checked for special cases, such as $x \leq 0$, $+\infty$, NaN etc. These checks are mainly implemented using integer arithmetics and will be further explained in section 3.2.1. Then, the argument is reduced using integer arithmetics as follows:

$$x = 2^{E'} \cdot m$$

where E' is the exponent of x and m a double corresponding to the mantissa of x. This decomposition is done such that in any case, i.e. even if x is subnormal, $1 \leq m < 2$. In the subnormal case, the exponent of x is adjusted accordingly. This first argument reduction corresponds to the equality

$$\log(x) = E' \cdot \log(2) + \log(m)$$

Using this term directly would lead to catastrophic cancellation in the case where $E' = -1$ and $m \approx 2$. To overcome this difficulty, a second adjustment is done as follows:

$$E = \begin{cases} E' & \text{if} \quad m \leq \sqrt{2} \\ E'+1 & \text{if} \quad m > \sqrt{2} \end{cases} \qquad y = \begin{cases} m & \text{if} \quad m \leq \sqrt{2} \\ \frac{m}{2} & \text{if} \quad m > \sqrt{2} \end{cases}$$

The decision whether $m \leq \sqrt{2}$ or not is performed using integer arithmetics on the high order bits of the mantissa m. The test is therefore not completely exact which is no disadvantage since, in any case, the bound $\sqrt{2}$ is somewhat arbitrary.

All the previous reduction steps can be implemented exactly as they consist mainly in decompositions of a floating point number, multiplications by powers of 2 and integer additions on the corresponding exponent value. All this leads to the following equation

$$\log(x) = E \cdot \log(2) + \log(y)$$

where

$$-\frac{1}{2}\cdot\log(2) \leq \log(y) \leq \frac{1}{2}\cdot\log(2)$$

The magnitude of y is thus still too great for allowing for a direct polynomial approximation of $\log(y)$. Therefore, a second argument reduction step is performed using a table of 128 entries as follows: using the high order bits of y as an index i, a tabulated value r_i is looked up which approximated very well $\frac{1}{y}$. Setting $z = y \cdot r_i - 1$, one obtains

$$\log(y) = \log(1+z) - \log(r_i)$$

Since $y = \frac{1}{r_i} + \delta$ the magnitude of z is finally small enough (typically $|z| < 2^{-8}$) for approximating $\log(1+z)$ by a Remez polynomial $p(z)$. The values for $\log(r_i)$ are of course also tabulated.

It is important to notice that the reduction step

$$z = y \cdot r_i - 1$$

can be implemented exactly which eases the correctness proof of the algorithm. This property will be proven in section 3.2.1. The reduced argument z will be represented as a double-double number $z_h + z_l$ that will be fed into the polynomial approximation algorithms of both quick and accurate phase. Each of these phases will take into account the lower significant value z_l for more or less higher monomial degrees.

Both phases will finally reconstruct the function's value as follows:

$$\log(x) \approx E \cdot \log(2) + p(z) - \log(r_i)$$

using a double (respectively a triple for the accurate phase) double value for each $\log(2)$ and $-\log(r_i)$. The computations necessary for performing this reconstruction are carried out in double-double arithmetics for the quick phase and triple-double for the accurate phase.

The quick phase uses a modified Remez polynomial of degree 7 of the form

$$p(z) = z - \frac{1}{2}\cdot z^2 + z^3 \cdot (c_3 + z\cdot(c_4 + z\cdot(c_5 + z\cdot(c_6 + z\cdot c_7))))$$

with $c_i \in \mathbb{F}$. This polynomial is evaluated as indicated by the parenthesis in the following term:

$$p(z_h + z_l) \approx \left((z_h + z_l) - \frac{1}{2}\cdot z_h^2\right) + \left((-z_h \cdot z_l) + \left(z_h^2 \cdot z_h\right)\cdot(c_3 + z_h\cdot(c_4 + z_h\cdot(c_5 + z_h\cdot(c_6 + z_h\cdot c_7))))\right)$$

The mathematical relative approximation error of the polynomial $p(z)$ defined as

$$\varepsilon_{\text{meth}} = \frac{p(z) - \log(1+z)}{\log(1+z)}$$

is bounded by

$$|\varepsilon_{\text{meth}}| \leq 2^{-62.99}$$

This methodical error is joined by the arithmetical error induced by the evaluation of $p(z)$ and by the rounding of the constants $\log(2)$ and $\log(r_i)$. As will be shown in section 3.2.2, the overall error of the quick phase defined as

$$\varepsilon_{\text{quick}} = \frac{(log_h + log_l) - \log(x)}{\log(x)}$$

is bounded by

$$|\varepsilon_{\text{quick}}| \leq 5 \cdot 2^{-65} \leq 2^{-62.6}$$

After the computation of the quick phase double-double value $(log_h + log_l)$ a rounding test is performed using the rounding constants according to 21. If the rounding cannot be decided, the accurate phase is launched.

The accurate phase performs all its computations on the same reduced argument $z = z_h + z_l$ which will be shown to be exact. An approximation polynomial of degree 14 is used. It is once again a modified Remez polynomial and has the following form:

$$p(z) = z + \frac{1}{2} \cdot z + z^3 \cdot q(z)$$

where

$$q(z) = c'_3 + z \cdot \left(c'_4 + z \cdot \left(c'_5 + z \cdot \left(c'_6 + z \cdot \left(c'_7 + z \cdot \left(c'_8 + z \cdot \left(c'_9 + z \cdot r(z)\right)\right)\right)\right)\right)\right)$$

with $c'_i = c_{ih} + c_{il} \in \mathbb{F} + \mathbb{F}$ and

$$r(z) = c_{10} + z \cdot (c_{11} + z \cdot (c_{12} + z \cdot (c_{13} + z \cdot c_{14})))$$

with $c_i \in \mathbb{F}$. The mathematical relative error

$$\varepsilon_{\text{meth}} = \frac{p(z) - \log(1+z)}{\log(1+z)}$$

is bounded by

$$|\varepsilon_{\text{meth}}| \leq 2^{-125}$$

The polynomial is evaluated using double precision for $r(z)$, double-double arithmetic for $q(z)$ and a triple-double representation for $p(z)$ and the final reconstruction.

The overall error

$$\varepsilon_{\text{accurate}} = \frac{(log_h + log_m + log_l) - \log(x)}{\log(x)}$$

is bounded by

$$|\varepsilon_{\text{accurate}}| \leq 5735 \cdot 2^{-132} \leq 2^{-119.5}$$

as will be shown in section 3.2.3. Here $(log_h + log_m + log_l)$ are obtained by reconstructing the logarithm as indicated by the parenthesis in the following term:

$$log_h + log_m + log_l = (E \cdot (log2_h + log2_m + log2_l)) + ((p_h + p_m + p_l) + (logi_h + logi_m + logi_l))$$

where $log2_h + log2_m + log2_l \approx \log(2)$ and $logi_h + logi_m + logi_l \approx -\log(r_i)$.

Since the critical accuracy of the double precision log function is 118 bits according to [11], rounding $\log_h + log_m + log_l \approx \log(x)$ to double precision is equivalent to rounding the infinite precision value $\log(x)$ to double precision. Using the final rounding sequences presented in [25], which are supposed to be correct, the double precision value returned by the function is the correctly rounded double precision value of $\log(x)$.

3.2 Proof of correctness of the triple-double implementation

Proving that an implementation of an elementary function is correctly rounded means mainly proving two bounds on the relative error $\varepsilon_{\text{quick}}$ and $\varepsilon_{\text{accurate}}$, using the appropriate lemma for proving the correctness of the rounding test and conluding by means of the theorem stating the critical accuracy of the function considered. The computation of the error bounds will be done mainly using the Gappa tool[31] but some parts of the proof are still based on paper or Maple computations. These parts will be shown in sections 3.2.1, 3.2.2 and 3.2.3 and mainly comprise the following:

- the demonstration that all special cases are handled correctly,
- a proof that $z_h + z_l = r_i \cdot y - 1$ exactly,
- the bounds for the mathematical approximation errors for the polynoms,
- a proof of the exactness of some multiplications in the code,
- the proof for the accuracy of all basic addition and multiplication code sequences on double-double and triple-double numbers,

- the correctness proof of the final rounding sequences for rounding triple-double numbers to double precision and
- the mathematical equality of the term rewriting hints in the Gappa code.

The proofs for the accuracy of the basic operation bricks and the correctness proof of the final rounding sequences are somewhat lengthy and are not given here; they can be found in [25].

3.2.1 Exactness of the argument reduction

In this section, we will show that all special cases are handled correctly and that the reduced argument consisting in E and $z_h + z_l$ is exact, which means that we have the mathematically exact equation

$$\log(x) = E \cdot \log(2) + \log(1 + (z_h + z_l)) - \log(r_i)$$

This part of the algorithm is performed by the following code sequences which we will analyse line by line:

Listing 3.1: Handling of special cases and table access

```
1  E=0;
2  xdb.d=x;
3
4  /* Filter cases */
5  if (xdb.i[HI] < 0x00100000){                    /* x < 2^(-1022)     */
6    if (((xdb.i[HI] & 0x7fffffff)|xdb.i[LO])==0){
7      return -1.0/0.0;
8    }                                             /* log(+/-0) = -Inf  */
9    if (xdb.i[HI] < 0){
10     return (x-x)/0;                             /* log(-x) = Nan     */
11   }
12   /* Subnormal number */
13   E = -52;
14   xdb.d *= ((db_number) ((double) two52)).d;    /* make x a normal number    */
15 }
16
17 if (xdb.i[HI] >= 0x7ff00000){
18   return  x+x;                                  /* Inf or Nan        */
19 }
20
21
22 /* Do argument reduction */
23 E += (xdb.i[HI]>>20)-1023;                      /* extract the exponent */
24 index = (xdb.i[HI] & 0x000fffff);
25 xdb.i[HI] =  index | 0x3ff00000;                /* do exponent = 0 */
26 index = (index + (1<<(20-L-1))) >> (20-L);
27
28 /* reduce  such that sqrt(2)/2 < xdb.d < sqrt(2) */
29 if (index >= MAXINDEX){                         /* corresponds to xdb>sqrt(2)*/
30   xdb.i[HI] -= 0x00100000;
31   E++;
32 }
33 y = xdb.d;
34 index = index & INDEXMASK;
35
36 ed = (double) E;
37
38 ri = argredtable[index].ri;
39
40 logih = argredtable[index].logih;
41 logim = argredtable[index].logim;
```

Analysis of the code:

line 1 and 2: Initialization of integer E and `db_number` `xdb` which is now equal to x.

line 5: As the integer ordering and the ordering on floating point numbers are compatible, $x < +2^{-1022}$, i.e. negative, negative infinite, equal to zero or a subnormal.

line 6: `xdb.i[HI] & 0x7fffffff` is the high order word of x without the sign bit. If the test is true, $|x| = 0$. As the logarithm of 0 is not defined but as the limit $-\infty$ is known, returning $-1.0/0.0$ is correct.

line 9: Since the integer ordering and the ordering on floating point numbers are compatible, `xdb.i[HI] < 0` implies $x < 0$. The logarithm is not defined for negative numbers, so the result must be NaN. $0.0/0.0$ leads to a NaN; one uses $(x - x)\ /0.0$ in order to overcome the static tests of the compiler.

line 13 and 14: if this code lines are reached, x is a subnormal. Since E equals 0 at this point, setting it to -52 and multipliying `xdb` by 2^{-52} means bringing `xdb` to the normal number range and rescaling the internal representation $x = 2^E \cdot m = 2^E \cdot$`xdb` in consequence.

line 17: As the integer ordering and the ordering on floating point numbers are compatible and as `0x7feffffff ffffffff` is the greatest normal, the test being true implies that x is equal to $+\infty$ or NaN. In the case of $x = +\infty$, $+\infty$ must be returned which is done. In the other case, NaN must be returned which is still assured by $x + x$.

line 23: At this point of the code, the most significant bit of the high order word of `xdb` must be 0 as the case where $x < 0$ is already filtered out. So `xdb.i[HI] > > 20` is equal to the biased exponent of `xdb` because a double number consists in 1 sign bit, 11 exponent bits and the word bit length is supposed to be 32. Subtracting 1023 yields to the unbiased exponent which is written to E.

line 24 and 25: Since a double number consists in 1 sign bit and 11 exponent bits, the operation `xdb.i[HI] & 0x000fffff` masks out the mantissa bits in the higher order word of `xdb`. Rewriting `xdb.i[HI] = index | 0x3ff00000` means setting the exponent of `xdb` to 0 because `0x3ff`$-1023 = 0$.

line 26: Before execution of this line of code, `index` contains the high order bits of the normalized mantissa of x stored as a double in `xdb.d` and verifying thus $1 \leq m < 2$. The second argument reduction step will slice this intervall in 128 intervalls for each of which we dispose of a table entry. For reasons of possible cancellation in the reconstruction step on the operation $p(z) - \log(r_i)$, we want the small intervalls to be centered around 1. That means e.g. for the intervall around 1 and a table indexed by 7 bits that mantissas (as doubles) with the high order word `0x3feffffff` through `0x3ff00fff` must be mapped to 0. The decision is therefore made at the $7 + 1$th bit of the mantissa part of the double depending on whether this bit is 0 – in which case the value falls in the lower intervall – or 1 – in which case the value goes to the next higher intervall. So adding 1 to the $(20 - 7 - 1)$ rightmost bit ($L = 7$) increases the index value by 1 iff this bit is 1. So after execution of the line, `index` contains the number of the intervall for the second argument reduction step centered in 1.

line 29 through 31: The second adjustment to be made on E' and m is the decision whether $m > \sqrt{2}$ as indicated in section 3.1. The high order word of $\sqrt{2}$ rounded to a double is `0x3ff6a09e`. As one can simply verify, the value for `index` calculated for this value is 53. As the integer ordering and the ordering of floating point numbers are compatible and as the computations for computing `index` are monotone, `index` being greater or equal than 53 implies that the (normalized) mantissa of x is greater than $\sqrt{2} + \delta$ with a neglectable error δ. As `MAXINDEX` is equal to 53, the test will be true iff the adjustment on E' leading to E and m yielding y is to be made. It is trivial to see that the code in the `if`'s body implements the adjustment correctly.

lines 33 and 34: the final value of the reduced argument y – still stored in `xdb.d` – is copied to a `double` variable (or register) named `y`. The final index value is masked out by means of an `INDEXMASK` which is equal to $127 = 2^7 - 1$.

lines 36: The integer value of the exponent E stored in `E` is cast to a `double ed`.

lines 38 through 41: The table is indexed by `index` and values `ri`$= r_i$ and `logih`$= logi_h$ and `logim`$= logi_m$ are read. Since the latter form a double-double precision value, we know that $logi_h + logi_m = \log(r_i) \cdot (1 + \varepsilon)$ with $|\varepsilon| \leq 2^{-106}$. The value `ri` is stored as a single precision variable and a Maple procedure assures that for each value y the following inegality is verified:

$$|z| = |y \cdot r_i - 1| \leq 2^{-8}$$

Let us show now that the following line calculate z_h and z_l such that for each y and corresponding r_i, we obtain exactly

$$z_h + z_l = y \cdot r_i - 1$$

Listing 3.2: Argument reduction

```
42 Mul12(&yrih, &yril, y, ri);
43 th = yrih - 1.0;
44 Add12Cond(zh, zl, th, yril);
```

We know that we can suppose that the multiplication and addition sequences **Mul12** and **Add12** used at lines 42 and 44 are exact. Thus, it suffices to show that

$$yri_h - 1.0 = yri_h \ominus 1.0$$

because in that case, we can note

$$z_h + z_l = th + yri_l = yri_h \ominus 1.0 + yri_l = y \cdot r_i - 1.0$$

We will show this property using Sterbenz' lemma. It suffices thus to prove that

$$\frac{1}{2} \leq yri_h \leq 2$$

We know that

$$\begin{aligned} yri_h &= \circ(y \cdot r_i) \\ &\leq \circ\left(1 + 2^{-8}\right) \\ &= 1 + 2^{-8} \\ &< 2 \end{aligned}$$

since the rounding function $\circ$ is monotonic and the accuracy of the format is greater than 9 bits.

The other way round, we get

$$\begin{aligned} yri_h &= \circ(y \cdot r_i) \\ &\geq \circ\left(1 - 2^{-8}\right) \\ &= 1 - 2^{-8} \\ &> \frac{1}{2} \end{aligned}$$

for the same reasons.

Thus $z_h + z_l = y \cdot r_i$ exactly. Since the previous phases of the argument reduction were all exact, the reduced argument verifies $x = 2^E \cdot y$ exactly.

Still in this section, let us show that neither the reduced argument of the logarithm function nor its result may be a sub-normal double number. The first property has already been assured by special case handling as shown above. The latter can be proven as follows: the $\log(x)$ function has one zero for $x = 1$ and only one. As it is monotone, for $x = 1 \pm 1\text{ulp} = 1 \pm 2^{-52}$ we will obtain $\log\left(1 \pm 2^{-52}\right) = 0 \pm 2^{-52} + \delta$ with $|\delta| \leq 2^{-103}$. As 0 ± 2^{-1022} is the least normal, the result of the logarithm function will always be a normal. Further, in both double-double and triple-double representations for the final intermediate result for the function, as its critical accuracy is 118, the least significant double in the representation will still be a normal as $52 + 106 = 158 < 1022$.

3.2.2 Accuracy proof of the quick phase

As already mentionned, the accuracy proof of the quick phase is mainly based on the Gappa tool. To prove the desired accuracy bound defined as

$$\varepsilon_{\text{quick}} = \frac{(log_h + log_l) - \log(x)}{\log(x)}$$

and given by

$$|\varepsilon_{\text{quick}}| \leq 5 \cdot 2^{-65} \leq 2^{-62.6}$$

three different Gappa proof files are necessary depending on the following cases:

- for $E \geq 1$ and all indexes to the table $0 \leq i \leq 127$, a general proof file named `log-td.gappa` is used
- for $E = 0$ and all indexes to the table except 0, i.e. $1 \leq i \leq 127$, a proof file named `log-td-E0.gappa` comes to hand and
- for $E = 0$ and the table index $i = 0$, a proof file called `log-td-E0-logir0.gappa` is employed. This latter file uses relative error computations in opposition to the other two cases where absolute error estimates suffice. This is necessary because in this case and in this one only, the logarithm function has a zero in the intervall considered.

In each of the three proof files, we will ask the Gappa tool to verify the accuracy bound expressed in its syntax as follows:

Listing 3.3: Accuracy bound to prove

```
109 ->
110 ((logh + logm) - Log) / Log in [-5b-65,5b-65]
```

Still in any proof file, some hypothesis are made on the correctness of one multiplication sequence and the accuracy of the constants and resting operations carried out in double-double arithmetic. These hypothesis are the following:

- The operations in the following code sequence are exact since the constants are stored with enough trailing zeros:

  Listing 3.4: Multiplication by E

  ```
  50 Add12(log2edh, log2edl, log2h * ed, log2m * ed);
  ```

  This means that $log2ed_h + log2ed_l = E \cdot (log2_h + log2_l)$ exactly.

- The operations in the following code sequence are exact since multiplications with a power of 2 are exact as long as the result is not underflowed:

  Listing 3.5: Multiplication by -0.5

  ```
  60 zhSquareHalfh = zhSquareh * -0.5;
  61 zhSquareHalfl = zhSquarel * -0.5;
  ```

  i.e. $zhSquareHalf_h + zhSquareHalf_l = -0.5 \cdot (zhSquare_h + zhSquare_l)$.

- The following hypothesis on the accuracy bounds, expressed here in Gappa syntax, are verified:

  Listing 3.6: Gappa hypothesis

  ```
  100 (T2hl - T2) / T2 in [-1b-103,1b-103]
  101 /\ (Phl - PE) / PE in [-1b-103,1b-103]
  102 /\ (LogTabPolyhl - LogTabPoly) / LogTabPoly in [-1b-103,1b-103]
  103 /\ (Loghm - LogE) / LogE in [-1b-103,1b-103]
  104 /\ (Log2hm - Log2) / Log2 in [-1b-84,1b-84]
  105 /\ (Logihm - Logir) / Logir in [-1b-106,1b-106]
  106 /\ Z in [_zmin,_zmax]
  107 /\ (P - Log1pZ) / Log1pZ in [-_epsilonApproxQuick,_epsilonApproxQuick]
  108 /\ ((logh + logm) - Loghm) / Loghm in [-1b-106,1b-106]
  ```

  Here, `_zmin`, `_zmax` and `_epsilonApproxQuick` are replaced by Maple calculated values, typically $-zmin = zmax = 2^{-8}$ and $epsilonApproxQuick = 2^{-62.99}$.

Let us now show each of this hypothesises.

1. The operations yielding `log2edh` and `log2edl` are all exact because the **Add12** sequence is supposed to be exact in any case and because the constants `log2h` and `log2m` are calculated by the following Maple code and have in consequence at least 11 trailing zeros and `ed` $= E$ is less than 1024 in magnitude since 1024 is the maximum exponent value for double precision.

Listing 3.7: Maple code for computing `log2h` and `log2m`

```
log2acc := log(2):
log2h := round(log2acc * 2**(floor(-log[2](abs(log2acc))) + (53 - 11))) /
         2**(floor(-log[2](abs(log2acc))) + (53 - 11)):
log2m := round((log2acc - log2h) * 2**(floor(-log[2](abs((log2acc - log2h)))) +
         (53 - 11))) / 2**(floor(-log[2](abs((log2acc - log2h)))) + (53 - 11)):
```

2. To show that $zhSquareHalf_h + zhSquareHalf_l = -0.5 \cdot (zhSquare_h + zhSquare_l)$ we just have to show that both values $zhSquare_h$ and $zhSquare_l$ are either equal to 0 or greater than 2 times the smallest normal. Let us first give the definitions of both values:

$$\begin{aligned} zhSquare_h &= \circ(z_h \cdot z_h) \\ zhSquare_l &= z_h \cdot z_h - zhSquare_h \end{aligned}$$

where $z_h = \circ(z)$. Let us suppose that $z \neq 0$. Otherwise all values are equal to 0 and we can conclude.

Let us first show that $|zhSquare_h|$ is greater than 2^{54} times the smallest normal. Let us therefore suppose that this is not the case, i.e. $|zhSquare_h| < 2^{-948}$. Since the rounding function is monotonic, this implies that $|z_h| \leq 2^{-424}$. For the same reason, we can note that $|z| \leq 2^{-424}$. As we have $z = y \cdot r_i - 1$, clearly neither y nor r_i can be exactly 1. If this were the case for both, we would obtain $z = 0$ which we excluded; if there were one of them only that was exactly 1, the other being a floating point number in the intervall $[0.5; 1.5]$, the resulting inegality $|z| \geq 2^{-53}$ which would be contradictory.

Otherwise, since we know that $1 - 2^{-8} \leq y \cdot r_i \leq 1 + 2^{-8}$ and since the precision of all formats used is greater than 9, the hypothesis that $1 - 2^{-424} \leq y \cdot r_i \leq 1 + 2^{-424}$ and $y \cdot r_i \neq 0$ would imply that the infinite precision mantissa of $y \cdot r_i$ contains a 1 weighted with 2^0 and a 1 weighted with less than 2^{-424}. So its length would be greater than 423 bits. As it is the product of two floating point numbers which have 52 and 23 significant bits, there cannot be a 1 weighted with less than 76 if there is a 1 weighted with 2^0 which is the case. Contradiction.

So $-0.5 \cdot zhSquare_h$ is not underflowed. Additionally, with a similar argument, since `zh` is a double precision number, $zhSquare_l$ is either 0 or greater in magnitude than $2^{-53} \cdot |zhSquare_h|$ which is 2^{52} times greater in magnitude than the smallest normal. So $zhSquare_l$ is either 0 or 2 times greater in magnitude than the smallest normal.

So, the floating point multiplication of $zhSquare_h$ and $zhSquare_l$ with -0.5 can be considered to be exact.

3. `(T2hl - T2) / T2 in [-1b-103,1b-103]` which means that

$$\left| \frac{T2hl - T2}{T2} \right| \leq 2^{-103}$$

is verified as $T2hl$ and $T2$ are defined as follows:

$$T2hl = t2_h + t2_l \leftarrow \mathbf{Add22}\,(z_h, z_l, zhSquareHalf_h, zhSquareHalf_l)$$

$$T2 = (z_h + z_l) + (zhSquareHalf_h + zhSquareHalf_l)$$

The given bound is thus just the accuracy bound of the **Add22** sequence for which a proof can be found in [25].

4. `(Phl - PE) / PE in [-1b-103,1b-103]` is verified for the same reason; let us just recall the definitions

$$Phl = p_h + p_l \leftarrow \mathbf{Add22}\,(t2_h, t2_l, t1_h, t1_l)$$

$$PE = (t2_h + t2_l) + (t1_h + t1_l)$$

5. `(LogTabPolyhl - LogTabPoly) / LogTabPoly in [-1b-103,1b-103]` falls still into the same case with

$$LogTabPolyhl = logTabPoly_h + logTabPoly_l \leftarrow \mathbf{Add22}\,(logi_h, logi_m, p_h, p_l)$$

$$LogTabPoly = (logi_h, +logi_m) + (p_h + p_l)$$

6. And finally, `(Loghm - LogE) / LogE in [-1b-103,1b-103]` which is also just the accuracy bound of the **Add22** sequence for

$$Loghm = log_h + log_m \leftarrow \mathbf{Add22}\,(log2ed_h, log2ed_l, logTabPoly_h, logTabPoly_l)$$

$$LogE = (log2ed_h + log2ed_l) + (logTabPoly_h + logTabPoly_l)$$

7. `(Log2hm - Log2) / Log2 in [-1b-84,1b-84]` is verified since $log2_h$ and $log2_m$ are computed as already indicated in listing 3.7. This means that at least 11 trailing zeros are stored in each in the doubles in this (pseudo-)double-double number, so it is exact to $2^{-106-2\cdot 11} = 2^{-84}$.

8. `(Logihm - Logir) / Logir in [-1b-106,1b-106]` which means

$$\left|\frac{(logi_h + logi_m) - \log(r_i)}{\log(r_i)}\right| \leq 2^{-106}$$

is verified by construction as $logi_h$ and $logi_m$ are computed by the following Maple code:

Listing 3.8: Maple code for computing $logi_h$ and $logi_m$

```
35 (logih[i], logim[i], logil[i]) := hi_mi_lo(evalf(-log(r[i]))):
```

where `hi_mi_lo` is the procedure for rounding an arbitrary precision number to a triple-double number the higher significant numbers of which form a double-double number.

9. The hypothesis `Z in [_zmin,_zmax]` simply recalls the bounds for z as calculated by Maple.

10. The same can be said on the hypothesis
`(P - Log1pZ) / Log1pZ in [-_epsilonApproxQuick,_epsilonApproxQuick]`
which gives the mathematical approximation error of the polynomial. This bound is computed by Maple using the following instructions:

Listing 3.9: Maple code for computing the relative error of the polynomial

```
129 epsilonApproxQuick := numapprox[infnorm]( 1-polyQuick/log(1+x), x=zminmin..zmaxmax)
```

11. Finally, Gappa's hypothesis `((logh + logm) - Loghm) / Loghm in [-1b-106,1b-106]` simply restates the fact that a double-double precision number is exact to at least 2^{-106} in terms of its relative error.

The Gappa tool itself is not capable of proving the final accuracy bound it is asked for a complex algorithm as the one given here. Its user must provide hints to help it to rewrite the interval arithmetics terms it encounters in the program. These hints are generally given in the form α `->` β where β is an expression we want the tool to rewrite the expression α by. Generally speaking, the idea behind each hint is one of the following:

- For computing intervall bounds on differences like $\alpha = a - A$ where both a and A are sums of terms like $a = c + C$ and $B = d + D$, it is often useful to rewrite α by $\beta = (c - d) + (C - D)$.
- An intervall bound can often be easier found for a term A representing an exact mathematical value that for a which is its arithmetical equivalent. So it is useful to rewrite a by $A \cdot \left(1 + \frac{a-A}{A}\right)$ when an intervall for $\frac{a-A}{A}$ is known.

- Fractional left hand sides like $\frac{a}{b}$ where both expressions a and b are functions in a common argument x that can be written like $a = a(x) = x^n \cdot a'(x)$ and $b = b(x) = x^m \cdot b'(x)$ should usually be rewritten as follows:
$$\frac{a(x)}{b(x)} = \frac{x^n \cdot a'(x)}{x^m \cdot b'(x)} = x^{n-m} \cdot \frac{a'(x)}{b'(x)}$$
In particular, this kind of hint is needed when an intervall for the denominator of a fractional left-hand-side comprises 0.

- Fractional left-hand-sides of the form $\frac{a-A}{A}$ with an unknown A can easily be written like
$$\frac{a-A}{A} = \frac{a-B}{B} + \frac{B-A}{A} + \frac{a-B}{B} \cdot \frac{B-A}{A}$$
We can show this equivalence like this
$$\begin{aligned}
\frac{a-A}{A} &= \frac{a-B+B-A}{A} \\
&= \frac{a-B}{A} + \frac{B-A}{A} \\
&= \frac{a-B}{B} \cdot \frac{B}{A} + \frac{B-A}{A} \\
&= \frac{a-B}{B} \cdot \left(1 + \frac{B-A}{A}\right) + \frac{B-A}{A} \\
&= \frac{a-B}{B} + \frac{B-A}{A} + \frac{a-B}{B} \cdot \frac{B-A}{A}
\end{aligned}$$
This is particularly useful when a bound on the relative error of some term a with regard to B should be extended to the next approximation level.

Clearly, the left-hand-side A and right-hand-side B of an hint must be mathematically equivalent to provide a correct result. The Gappa tool checks for this equivalence and sometimes is able to prove it. If not, it emits a warning indicating that the formal proof it is generating for the accuracy bound computations is valid only under the hypothesis that both sides of the rewriting hint are mathematically equivalent. Further, it prints out the difference $A - B$ of both sides A and B which it has already reduced using the equivalences given in the Gappa code. It is relatively simple to verify that all this differences are equal to 0 modulo the definitions given in the Gappa code by means of Maple-scripts. This work can even been done automatically. Thus, we refrain from giving a paper proof of each hint in the Gappa files used for proving the logarithm function but just give the exhaustive list of the hints in files `log-td.gappa` and `log-td-E0-logir0.gappa`:

Listing 3.10: Gappa term rewriting hints in file `log-td.gappa`

```
115 T2hl - T2 -> ((T2hl - T2) / T2) * T2;
116 T2hl -> (T2hl - T2) + T2;
117
118 Phl - PE -> ((Phl - PE) / PE) * PE;
119 Phl -> (Phl - PE) + PE;
120
121
122 LogTabPolyhl -> (LogTabPolyhl - LogTabPoly) + LogTabPoly;
123
124 Loghm -> (Loghm - LogE) + LogE;
125
126 Log2 -> Log2hm * (1 / (((Log2hm - Log2) / Log2) + 1));
127
128 Logir -> Logihm * (1 / (((Logihm - Logir) / Logir) + 1));
129
130
131 LogTabPolyhl - LogTabPoly -> ((LogTabPolyhl - LogTabPoly) / LogTabPoly) * LogTabPoly;
132
133 HZZsimp -> (-0.5 * zh * zh) - (0.5 * zl * zl);
134
135 T2hl - ZpHZZsimp -> (0.5 * zl * zl) + delta1;
136
137 zhCube - ZZZ -> (Z * (zhSquareh - Z * Z)) - (zl * zhSquareh);
```

```
138
139 polyUpper - ZZZPhigher -> ZZZ * (polyHorner - Phigher) + polyHorner * delta3 + delta2;
140
141 ZpHZZ + ZZZPhigher -> ZpHZZsimp + ZZZPhigherPzhzl;
142
143 Phl - P -> (T2hl - ZpHZZsimp) + (T1hl - ZZZPhigherPzhzl) + delta4;
144
145 Log1pZ -> P * (1 / (((P - Log1pZ) / Log1pZ) + 1));
146 P - Log1pZ -> ((P - Log1pZ) / Log1pZ) * Log1pZ;
147
148 Phl - Log1pZ -> (Phl - P) + delta6;
149
150 LogTabPolyhl - Log1pZpTab -> (Logihm - Logir) + (Phl - Log1pZ) + delta7;
151
152 Loghm - Log -> (Log2edhm - Log2E) + (LogTabPolyhl - Log1pZpTab) + delta5;
153
154 (logh + logm) - Loghm -> (((logh + logm) - Loghm) / Loghm) * Loghm;
155
156 (logh + logm) - Log -> ((logh + logm) - Loghm) + (Loghm - Log);
```

Listing 3.11: Gappa term rewriting hints in file `log-td-E0-logir0.gappa`

```
81 T2hl - T2 -> ((T2hl - T2) / T2) * T2;
82 T2hl -> (T2hl - T2) + T2;
83
84 Phl - PE -> ((Phl - PE) / PE) * PE;
85 Phl -> (Phl - PE) + PE;
86
87
88 (ZhSquarehl - ZZ) / ZZ -> 2 * ((zh - Z) / Z) + ((zh - Z) / Z) * ((zh - Z) / Z);
89
90 (zhSquareh - ZZ) / ZZ -> ((ZhSquarehl - ZZ) / ZZ) + ((zhSquareh - ZhSquarehl) / ZZ);
91
92 (zhSquareh - ZhSquarehl) / ZZ -> ((zhSquareh - ZhSquarehl) / ZhSquarehl) * (ZhSquarehl / ZZ);
93
94 ZhSquarehl / ZZ -> ((ZhSquarehl - ZZ) / ZZ) + 1;
95
96 (ZhCube - ZZZ) / ZZZ -> (((zh * zhSquareh) - ZZZ) / ZZZ) + ((ZhCube - (zh * zhSquareh)) / ZZZ)
       ;
97
98 ((zh * zhSquareh) - ZZZ) / ZZZ -> (1 + ((zh - Z) / Z)) * (1 + ((zhSquareh - ZZ) / ZZ)) - 1;
99
100 ((ZhCube - (zh * zhSquareh)) / ZZZ) -> ((ZhCube - (zh * zhSquareh)) / (zh * zhSquareh)) * (((
       zh - Z) / Z) + 1) * (((zhSquareh - ZZ) / ZZ) + 1);
101
102 polyHorner / Phigher -> ((polyHorner - Phigher) / Phigher) + 1;
103
104 (polyUpper - ZZZPhigher) / ZZZPhigher -> ((polyHorner - Phigher) / Phigher) + ((ZhCube - ZZZ)
       / ZZZ) * (polyHorner / Phigher) +
105                                          + ((polyUpper - (polyHorner * ZhCube)) / (polyHorner
                                               * ZhCube)) * (polyHorner / Phigher) +
106                                          + ((ZhCube - ZZZ) / ZZZ) * ((polyUpper - (polyHorner
                                               * ZhCube)) / (polyHorner * ZhCube)) *
107                                            (polyHorner / Phigher);
108
109
110 ((ZhSquareHalfhl - (zh * zl)) - HZZ) / HZZ -> - ((zh - Z) / Z) * ((zh - Z) / Z);
111
112 (ZhSquareHalfhl - HZZ) / HZZ -> (ZhSquarehl - ZZ) / ZZ;
113
114 ((T2hl - (zh * zl)) - ZpHZZ) / ZpHZZ -> ((HZ * (((ZhSquareHalfhl - (zh * zl)) - HZZ) / HZZ)) +
       ((T2hl - T2) / T2)
115                                          + (HZ * ((T2hl - T2) / T2))
116                                          + (HZ * ((ZhSquareHalfhl - HZZ) / HZZ) * ((T2hl - T2)
                                               / T2))) / (1 + HZ);
117
118 (PE - P) / P -> (((1 + HZ) * (((T2hl - (zh * zl)) - ZpHZZ) / ZpHZZ)) +
119                  ((1 + ((zh - Z) / Z)) * (Z * ((zh - Z) / Z)) * ((Flzhzl - (zh * zl)) / (zh *
                       zl)))
120                  + (ZZ * Phigher * ((polyUpper - ZZZPhigher) / ZZZPhigher))) / (1 + HZ + ZZ *
                       Phigher);
121
122 (Phl - P) / P -> ((PE - P) / P) + ((((PE - P) / P) + 1) * ((Phl - PE) / PE));
123
124 (Loghm - Log) / Log -> ((Loghm - P) / P) + ((P - Log) / Log) + ((Loghm - P) / P) * ((P - Log)
       / Log);
125
126 (((logh + logm) - Log) / Log) -> (((logh + logm) - Loghm) / Loghm) + ((Loghm - Log) / Log) +
```

```
(((logh + logm) - Loghm) / Loghm) * ((Loghm - Log) / Log);
```

For the reasons mentionned, we can consider the accuracy proof of the quick phase to be correct.

3.2.3 Accuracy proof of the accurate phase

The accuracy proof of the accurate phase is also based mainly on the use of the Gappa tool. Nevertheless, since the tool is currently not directly supporting triple-double representations, some additional hand-proven accuracy bound results for the main addition and multiplication operators are needed. They can be found in [25]. Since all these accuracy bounds are parameterized by the maximal overlap bound for the triple-double numbers along the computations, before being able to give a numerical value for these error bounds understood by the Gappa tool, it is necessary to do a maximal overlap bound analysis using the theorems given in [25].

Eventually, since not an overlapped triple-double intermediate result is to be returned by the logarithm function but a double precision number that is the correct rounding according to the rounding mode chosen, the algorithm effectuates a renormalizing operation on the final result and rounds this non-overlapped result down to a double using an appropriate rounding sequence. All this renormalization and rounding sequences are exact and have been shown to be correct in [25]. The same way, all properties shown in section 3.2.1 concerning the special case handling and exactness argument reduction can be reused because the algorithm implemented in the accurate phase uses the same reduced argument and is substantially the same as for the quick phase.

We will thus rely on all these properties and simply show the following accuracy bound

$$\varepsilon_{\text{accurate}} = \frac{(log_h + log_m + log_l) - \log(x)}{\log(x)}$$

is bounded by

$$|\varepsilon_{\text{accurate}}| \leq 5735 \cdot 2^{-132} \leq 2^{-119.5}$$

which will be expressed in Gappa syntax as follows:

Listing 3.12: Accuracy bound to prove for the accurate phase

```
165 ->
166 ((logh + logm + logl) - MLog) / MLog in [-5735b-132,5735b-132]
```

The Gappa proof files still make the hypothesis that two of the multiplications in the accurate phase code can be considered to be exact. This property must therefore be shown in a paper proof in the following.

The first of these multiplications is the following sequence:

Listing 3.13: Multiplication of triple-double $\circ(Z \cdot Z)$ by $-\frac{1}{2}$

```
99  zSquareHalfh = zSquareh * -0.5;
100 zSquareHalfm = zSquarem * -0.5;
101 zSquareHalfl = zSquarel * -0.5;
```

As it will be shown below, the relative error $\varepsilon_{ZSquare}$ defined as

$$\varepsilon_{ZSquare} = \frac{(zSquare_h + zSquare_m + zSquare_l) - Z^2}{Z^2}$$

is bounded by $\left|\varepsilon_{ZSquare}\right| \leq 2^{-149}$. Using the same argument as the one given in section 3.2.2, one can show that Z is either 0 or greater in magnitude than at least 2^{-77}. So the following is true

$$Z^2 = 0 \vee \left|Z^2\right| \geq 2^{-154}$$

If $Z^2 = 0$, $ZSquarehml = zSquare_h + zSquare_m + zSquare_l$ trivially is 0, too, and the multiplication is with $-\frac{1}{2}$ is therefore exact. Since we can note $ZSquarehml = Z^2 \cdot (1 + \varepsilon_{ZSquare})$, we know that in the other case,

$$|ZSquarehml| \geq 2^{-155}$$

We can suppose that in the triple-double number $zSquare_h + zSquare_m + zSquare_l$, $zSquare_m$ and $zSquare_l$ are not overlapped at all (since $zSquare_m = \circ(zSquare_m + zSquare_l)$) and that $zSquare_h$ and $zSquare_m$ are not fully overlapped. So we can note $|zSquare_m| \leq 2^{-\beta_o} \cdot |zSquare_h|$ and $|zSquare_l| \leq 2^{-\beta_u} \cdot |zSquare_m|$ with $\beta_o \geq 1$ and $\beta_u \geq 53$. We will show this property below we are just supposing here. So we can verify the following

$$\begin{aligned} |ZSquarehml| &= |zSquare_h + zSquare_m + zSquare_l| \\ &\leq |zSquare_h| + |zSquare_m| + |zSquare_l| \\ &\leq |zSquare_h| + 2^{-\beta_o} \cdot |zSquare_h| + 2^{-\beta_o} \cdot 2^{-\beta_u} \cdot |zSquare_h| \\ &\leq 2 \cdot |zSquare_h| \end{aligned}$$

In consequence, we obtain

$$|zSquare_h| \geq \frac{1}{2} \cdot |ZSquarehml|$$

and thus

$$|zSquare_h| \geq 2^{-156}$$

under the hypothesis that it is not exactly zero. So $zSquareHalf_h = -\frac{1}{2} \cdot zSquare_h$ will never be underflowed.

Let us now show first that the operations for computing $zSquareHalf_m$ and $zSquareHalf_l$ cannot both be inexact. We will use the fact that $|zSquare_l| \leq 2^{-53} \cdot |zSquare_m|$. Suppose first that

$$zSquareHalf_m \leftarrow -\frac{1}{2} \otimes zSquare_m$$

is inexact. So $|zSquare_m| < 2^{-1022}$ and in consequence $|zSquare_l| < 2^{-1022-53}$. Note that the inegality is strict. Since the least (in magnitude) representable denormalized double precision floating point number is $2^{-52} \cdot 2^{-1023}$, $zSquare_l = 0$ in this case. So

$$zSquareHalf_l \leftarrow -\frac{1}{2} \otimes zSquare_l$$

is exact because trivially, a multiplication with 0 is exact.

Suppose now that

$$zSquareHalf_l \leftarrow -\frac{1}{2} \otimes zSquare_l$$

is inexact. So $|zSquare_l| < 2^{-1022}$. Further, the least significant bit of the mantissa of $zSquare_l$ is 1 because otherwise, a bit-shift in its mantissa by 1 would be an exact operation. Thus $|zSquare_l| \geq 2^{-52} \cdot 2^{-1023}$ and $|zSquare_m| \geq 2^{-1022}$. So

$$zSquareHalf_m \leftarrow -\frac{1}{2} \otimes zSquare_m$$

cannot be inexact because in this case we would have $|zSquare_m| < 2^{-1022}$.

So, in any case, if ever $zSquareHalf_m + zSquareHalf_l$ are not exactly $-\frac{1}{2} \cdot (zSquare_m + zSquare_l)$, the error made will be $\frac{1}{2} \cdot d$ in magnitude, where $d = 0^+$ is the smallest representable denormalized non-zero double. So we can note down in this case

$$zSquareHalf_h + zSquareHalf_m + zSquareHalf_l = -\frac{1}{2} \cdot (zSquare_h + zSquare_m + zSquare_l) + \delta$$

with $|\delta| \leq 2^{-1075}$. Since we know that $\left|-\frac{1}{2} \cdot (zSquare_h + zSquare_m + zSquare_l)\right| \geq 2^{-156}$, we can give the following bound

$$\left|\frac{\delta}{-\frac{1}{2} \cdot (zSquare_h + zSquare_m + zSquare_l)}\right| \leq \frac{2^{-1075}}{2^{-156}} = 2^{-919}$$

So we get

$$ZSquareHalfhml = -\frac{1}{2} \cdot ZSquarehml \cdot (1 + \varepsilon)$$

with $|\varepsilon| \leq 2^{-919}$

In contrast, since we know that $|Z| \leq 2^{-8}$ thus that $\left|Z^2\right| \leq 2^{-16}$ but that $\left|Z^2\right| \geq 2^{-154}$, we can assume that the infinite precision mantissa of Z^2 can always be written exactly with at most $154 - 16 = 138 < 149$ bits. As we can show that $\frac{1}{2} \cdot |ZSquarehml| \leq |zSquare_h| \leq 2 \cdot |ZSquarehml|$ we know that if ever one of $zSquare_m$ or $zSquare_l$ is such that the multiplication with $-\frac{1}{2}$ is not exact, the error made has already been accounted for in the error bound for $ZSquarehml$ with regard to Z^2. So the operation computing $ZSquareHalfhml$ out of $ZSquarehml$ can be considered to be exact.

Let us now analyse the following sequence

Listing 3.14: Multiplication of triple-double $Log2hml$ by E

```
126   log2edhover = log2h * ed;
127   log2edmover = log2m * ed;
128   log2edlover = log2l * ed;
```

Similar to the argumentation that has been given in section 3.2.2, since $E = ed$ is bound in magnitude by $1024 = 2^{10}$ and since $log2_h$, $log2_m$ are stored with at least 11 trailing bits at zero, the multiplications in these components are exact. The constant $log2_l$ is not stored with 11 trailing bits at zero but it could be because we will be just supposing the bound $\left|\varepsilon_{Log2hml}\right| \leq 2^{-3 \cdot 53 + 33} = 2^{-126}$ for

$$\varepsilon_{Log2hml} = \frac{log2_h + log2_m + log2_l - \log(2)}{\log(2)}$$

So the multiplication is not exact in itself but the final result is exacter than the bound we are using for it.

Let us finally just recall the Maple code for computing the constants:

Listing 3.15: Maple code for computing $Log2hml$

```
21 log2acc := log(2):
22 log2h := round(log2acc * 2**(floor(-log[2](abs(log2acc))) + (53 - 11))) / 2**(floor(-log[2](
       abs(log2acc))) + (53 - 11)):
23 log2m := round((log2acc - log2h) * 2**(floor(-log[2](abs((log2acc - log2h)))) + (53 - 11))) /
       2**(floor(-log[2](abs((log2acc - log2h)))) + (53 - 11)):
24 log2l := log2acc - (log2h + log2m):
```

So the multiplication can be considered to be exact as long the less accurate bound for $\varepsilon_{Log2hml}$ is used.

Let us know analyse the bounds that we can give for the maximal overlap of the components of the triple-double numbers in the logarithm implementation. For doing this, we will assign each triple-double number in the code an overlap bound as follows. Call the number in consideration e.g. $a_h + a_m + a_l$. So we will give the bounds expressed like this:

$$\begin{aligned} |a_m| &\leq 2^{-\alpha_o} \cdot |a_h| \\ |a_l| &\leq 2^{-\alpha_u} \cdot |a_m| \end{aligned}$$

where $\alpha_o, \alpha_u \geq 2$. We will then propagate this information following the flow of control in the implementation and using the overlap bound theorems given in [25]. Here, we understand by "propagating" checking a system of constraints of the bounds under the limitations provided by the theorems. As the control-flow-graph of our implementation is completely linear, this check is linear, too. The theorems mentionned can be summarized as follows:

Operation	1st arg.	2nd arg.	result high	result low
Add33	$\alpha_o \geq 4, \alpha_u \geq 1$	$\beta_o \geq 4, \beta_u \geq 1$	$\gamma_o \geq \min(\alpha_o, \beta_o) - 5$	$\gamma_u \geq 53$
Add233	-	$\beta_o \geq 2, \beta_u \geq 1$	$\gamma_o \geq \min(45, \beta_o - 4, \beta_o + \beta_u - 2)$	$\gamma_u \geq 53$
Mul23	-	-	$\gamma_o \geq 48$	$\gamma_u \geq 53$
Mul233	-	$\beta_o \geq 2, \beta_u \geq 1$	$\gamma_o \geq \min(48, \beta_o - 4, \beta_o + \beta_u - 4)$	$\gamma_u \geq 53$

So let us analyse the following code:

Listing 3.16: Triple-double computations

```
90 Mul23(&zSquareh, &zSquarem, &zSquarel, zh, zl, zh, zl);
91 Mul233(&zCubeh, &zCubem, &zCubel, zh, zl, zSquareh, zSquarem, zSquarel);
```

```
92 Mul233(&higherPolyMultZh, &higherPolyMultZm, &higherPolyMultZl, t14h, t14l, zCubeh, zCubem,
       zCubel);
93 zSquareHalfh = zSquareh * -0.5;
94 zSquareHalfm = zSquarem * -0.5;
95 zSquareHalfl = zSquarel * -0.5;
96 Add33(&polyWithSquareh, &polyWithSquarem, &polyWithSquarel,
97       zSquareHalfh, zSquareHalfm, zSquareHalfl,
98       higherPolyMultZh, higherPolyMultZm, higherPolyMultZl);
99 Add233(&polyh, &polym, &polyl, zh, zl, polyWithSquareh, polyWithSquarem, polyWithSquarel);
100 Add33(&logyh, &logym, &logyl, logih, logim, logil, polyh, polym, polyl);
101 log2edhover = log2h * ed;
102 log2edmover = log2m * ed;
103 log2edlover = log2l * ed;
104 log2edh = log2edhover;
105 log2edm = log2edmover;
106 log2edl = log2edlover;
107 Add33(&loghover, &logmover, &loglover, log2edh, log2edm, log2edl, logyh, logym, logyl);
```

This code will finally generate triple-double numbers respecting the following overlap bounds as will be shown below:

Variable	Line(s)	$\alpha_o \geq$	$\alpha_u \geq$
ZSquarehml	90	48	53
ZCubehml	91	44	53
HigherPolyMultZhml	92	40	53
ZSquareHalfhml	93-95	48	53
PolyWithSquarehml	96-98	35	53
Polyhml	99	31	53
Logyhml	100	26	53
Log2edhml	101-106	40	40
Logoverhml	107	21	53

So let us verify exemplarily some of these bounds:

- At line 90, *ZSquarehml* is computed out of the double-double number $z_h + z_l$ by use of the **Mul23** sequence. Since the inputs of this function are not triple-double, the overlap bound is just the bound provided by the sequence itself, i.e. $\alpha_o \geq 48$, $\alpha_u \geq 53$.
- *ZCubehml* is the result of a **Mul233** sequence at line 91. Its overlap bound depends therefore on the one for *ZSquarehml*, which is the second argument of the function. Since we know the bound for this variable, we easily verify the one for *ZCubehml* which is $\alpha_o \geq 44$ and $\alpha_u \geq 53$.
- *Log2edhml* is the exact pairwise product of the triple-double constant *Log2hml* and double *E*. Since *E* may be as small as 0 in magnitude and further, since the multiplication is pairwise, the overlap bound we dispose of for *Log2edhml* is the same as for *Log2hml* which is stored with at least 11 bit trailing zeros. So an appropriate bound is $\alpha_o \geq 52 - 11 \geq 40$ and $\alpha_u \geq 40$.

All other bounds can be verified the same way using the theorems given in [25] and indicated above.

Since we have computed the overlap bounds for the different triple-double operands in the code, we can now calculate the accuracy bounds for the operations. Doing this is only possible with the knowledge of the overlap of the operations because all accuracy bound theorems given in [25] are parameterized with this overlap expressions.

Let us first give a list of the accuracy of the different basic operations which is not exhaustive with regard to its lack of listing almost all preconditions on the sequences required for theorems to hold. We refrain from explicitely verifying each of this preconditions in this document as this is only fastidious work but not of special interest.

Operation	Overlap 1st arg.	Overlap 2nd arg.	Relative error ε
Add22	-	-	$\lvert\varepsilon\rvert \leq 2^{-103.5} \leq 2^{-103}$
Mul22	-	-	$\lvert\varepsilon\rvert \leq 2^{-102}$
Add33	$\alpha_o \geq 4$, $\alpha_u \geq 1$	$\beta_o \geq 4$, $\beta_u \geq 1$	$\lvert\varepsilon\rvert \leq 2^{-\min(\alpha_o+\alpha_u,\beta_o+\beta_u)-47} + 2^{-\min(\alpha_o,\beta_o)-98}$
Add233	-	$\beta_o \geq 2$, $\beta_u \geq 1$	$\lvert\varepsilon\rvert \leq 2^{-\beta_o-\beta_u-52} + 2^{-\beta_o-104} + 2^{-153}$
Mul23	-	-	$\lvert\varepsilon\rvert \leq 2^{-149}$
Mul233	-	$\beta_o \geq 2$, $\beta_u \geq 1$	$\lvert\varepsilon\rvert \leq 2^{-97-\beta_o} + 2^{-97-\beta_o-\beta_u} + 2^{-150}$

Still analyzing the following double-double computations code and the code given at listing 3.16, one can now easily check the bounds for the relative error of the different operations listed in the table below. We define here the relative error of an operation $*$ and its arithmetical equivalent $\circledast$ as follows:

$$\varepsilon = \frac{(a \circledast b) - (a * b)}{(a * b)}$$

Listing 3.17: Double-double computations in accurate phase

```
73  Mul12(&t1h, &t1l, zh, highPoly);
74  Add22(&t2h, &t2l, accPolyC9h, accPolyC9l, t1h, t1l);
75  Mul22(&t3h, &t3l, zh, zl, t2h, t2l);
76  Add22(&t4h, &t4l, accPolyC8h, accPolyC8l, t3h, t3l);
77  Mul22(&t5h, &t5l, zh, zl, t4h, t4l);
78  Add22(&t6h, &t6l, accPolyC7h, accPolyC7l, t5h, t5l);
79  Mul22(&t7h, &t7l, zh, zl, t6h, t6l);
80  Add22(&t8h, &t8l, accPolyC6h, accPolyC6l, t7h, t7l);
81  Mul22(&t9h, &t9l, zh, zl, t8h, t8l);
82  Add22(&t10h, &t10l, accPolyC5h, accPolyC5l, t9h, t9l);
83  Mul22(&t11h, &t11l, zh, zl, t10h, t10l);
84  Add22(&t12h, &t12l, accPolyC4h, accPolyC4l, t11h, t11l);
85  Mul22(&t13h, &t13l, zh, zl, t12h, t12l);
86  Add22(&t14h, &t14l, accPolyC3h, accPolyC3l, t13h, t13l);
```

Result	Line(s)	Operation	Relative error ε
T1hl through *T14hl*	73 - 86	**Add22** / **Mul22**	$\lvert\varepsilon\rvert \leq 2^{-103}$ / $\lvert\varepsilon\rvert \leq 2^{-102}$
ZSquarehml	90	**Mul23**	$\lvert\varepsilon\rvert \leq 2^{-149}$
ZCubehml	91	**Mul233**	$\lvert\varepsilon\rvert \leq 2^{-144}$
HigherPolyMultZhml	92	**Mul233**	$\lvert\varepsilon\rvert \leq 2^{-141}$
PolyWithSquarehml	96-98	**Add33**	$\lvert\varepsilon\rvert \leq 2^{-137}$
Polyhml	99	**Add233**	$\lvert\varepsilon\rvert \leq 2^{-134}$
Logyhml	100	**Add33**	$\lvert\varepsilon\rvert \leq 2^{-128}$
Logoverhml	107	**Add33**	$\lvert\varepsilon\rvert \leq 2^{-123}$

Let us just explicitely check the bound for one of the operations for sake of an example. Let us take therefore the **Add33** sequence at lines 96-98 computing *PolyWithSquarehml* out of *ZSquareHalfhml* and *HigherPolyMultZhml*. We have already obtained to following overlap bounds:

$$\begin{aligned}
|zSquareHalf_m| &\leq 2^{-48} \cdot |zSquareHalf_h| \\
|zSquareHalf_l| &\leq 2^{-53} \cdot |zSquareHalf_m| \\
|higherPolyMultZ_m| &\leq 2^{-40} \cdot |higherPolyMultZ_h| \\
|higherPolyMultZ_l| &\leq 2^{-53} \cdot |higherPolyMultZ_m|
\end{aligned}$$

Feeding now this bounds into the theorem on the accuracy of **Add33**, we get

$$|\varepsilon| \leq 2^{-\min(48+53,40+53)-47} + 2^{-\min(48,40)-98} \leq 2^{-140} + 2^{-138} \leq 2^{-137}$$

All other error bounds can be verified in a similar way. They are finally expressed in Gappa syntax as follows:

Listing 3.18: Relative error bounds in Gappa code

```
139 (T2hl - T2) / T2 in [-1b-103,1b-103]
140 /\ (T3hl - T3) / T3 in [-1b-102,1b-102]
141 /\ (T4hl - T4) / T4 in [-1b-103,1b-103]
142 /\ (T5hl - T5) / T5 in [-1b-102,1b-102]
143 /\ (T6hl - T6) / T6 in [-1b-103,1b-103]
144 /\ (T7hl - T7) / T7 in [-1b-102,1b-102]
145 /\ (T8hl - T8) / T8 in [-1b-103,1b-103]
146 /\ (T9hl - T9) / T9 in [-1b-102,1b-102]
147 /\ (T10hl - T10) / T10 in [-1b-103,1b-103]
148 /\ (T11hl - T11) / T11 in [-1b-102,1b-102]
149 /\ (T12hl - T12) / T12 in [-1b-103,1b-103]
```

```
/\ (T13hl - T13) / T13 in [-1b-102,1b-102]
/\ (T14hl - T14) / T14 in [-1b-103,1b-103]
/\ (ZSquarehml - ZSquare) / ZSquare in [-1b-149,1b-149]
/\ (ZCubehml - ZCube) / ZCube in [-1b-144,1b-144]
/\ (HigherPolyMultZhml - HigherPolyMultZ) / HigherPolyMultZ in [-1b-141,1b-141]
/\ (PolyWithSquarehml - PolyWithSquare) / PolyWithSquare in [-1b-137,1b-137]
/\ (Polyhml - Poly) / Poly in [-1b-134,1b-134]
/\ (Logyhml - Logy) / Logy in [-1b-128,1b-128]
/\ (Loghml - Logover) / Logover in [-1b-123,1b-123]
/\ (Log2hml - MLog2) / MLog2 in [-1b-126,1b-126]
/\ (Logihml - MLogi) / MLogi in [-1b-159,1b-159]
/\ (MPoly - MLog1pZ) / MLog1pZ in [-_epsilonApproxAccurate,_epsilonApproxAccurate]
/\ Z in [_zmin,_zmax]
/\ ((logh + logm + logl) - Loghml) / Loghml in [-1b-159,1b-159]
```

Concerning the Gappa proofs for accurate phase, in a similar way as for the quick phase, three different proof files are used. They reflect once again the three main cases for the argument of the logarithm function:

- For cases where after argument reduction $|E| \geq 1$, the file `log-td-accurate.gappa` is used. In this case, absolute error computations are sufficient for the final relative error bound to be calculable because $|\log(x)| \geq \frac{1}{2}\log(2)$ in this case.
- For the case where after argument reduction, $E = 0$ and $i \neq 0$, the file `log-td-accurate-E0.gappa` is used. The same way here, we have a preponderant constant term so absolute error computations suffice.
- For the other case, where $E = 0$ and $i = 0$ the file `log-td-accurate-E0-logir0.gappa` provides the accuracy bound proof. In contrast to the other cases, we obliged to relative error estimations since the beginning since the function $\log(x)$ has a zero in this intervall.

Once again, several term rewriting hints are needed in the Gappa proof files for enabling the Gappa tool to generate a proof for the accuracy bounds. In a similar way, the hints which cannot directly be checked for their mathematical correctness by the tool itself are verified by semi-automatic Maple scripts.

By the existence of an accuracy proof for a final relative error of $|\varepsilon_{\text{accurate}}| \leq 2^{-119.5}$ and by the use of the critical accuracy of the double precision natural logarithm function which is 118 bits[11], we can consider the implementation to be correctly rounding under the hypothesis that the final rounding sequence used is exact and correct. Since we suppose this – a correctness proof can be found in [25] – the correctly rounding property is verified.

3.3 Proof of correctness of the double-extended implementation

3.4 Performance results

The given implementation of the natural logarithm function aims at being both portable and more performant than the previous implementations using the SCS libary for the accurate phase. This goal is acheived in terms of memory consumption (if the code sizes for `scslib` are taken into account) and in terms of speed performance. The reason for this is mainly the possibility of reusing all values computed in the argument reduction phase and the tables for the accurate phase directly.

3.4.1 Memory requirements

3.4.2 Timings

Chapter 4

The logarithm in base 2

Todo. In between, see files log2-td.{h,c,mpl,gappa}.

Chapter 5

The logarithm in base 10

This chapter is contributed by Ch. Q. Lauter.

5.1 Main considerations, critical accuracy bounds

If one wants to guarantee that an implementation of the logarithm in base 10, $\log_{10}(x)$, in double precision is correctly rounded, one has to ensure that the final intermediate approximation before rounding to double precision has a relative error of less than 2^{-122}.

An implementation of $\log_{10}(x)$ can also be derived from an implementation of the natural logarithm $\ln(x)$ using the formula:

$$\log_{10}(x) = \frac{1}{\ln(10)} \cdot \ln(x) \tag{5.1}$$

When doing so, one must ensure that the constant $log10inv = \frac{1}{\ln(10)}$ is stored with enough accuracy, that the approximation for $\ln(x)$ is exact enough and that the multiplication sequence does not introduce too great an error. As we will see in the next section 5.2, this implies slight changes to the code for the natural logarithm with regard to what has been presented in Chapter 3.

With regard to final rounding, the elementary function $\log_{10}$ presents a particular issue that is somewhat singular amongst all considered elementary functions: There exist a large set of input double-precision numbers x such that $\log_{10}(x)$ is a rational and is representable as a double-precision number.

For such cases, a final directed rounding will be correct only if the approximation error is exactly 0. Indeed, the rounding $\diamond(f(x))$ of the exactly representably value $f(x) \in \mathbb{F}$ is trivially $\diamond(f(x)) = f(x)$ [5]. In contrast, $\diamond(f(x) + \delta) \neq f(x) \in \mathbb{F}$ holds for all $|\delta| > 0$.

As it is impossible to achieve an approximation error exactly equal to zero, it is preferable to filter out such cases and handle them separately. Other functions described so far had only one such argument ($x = 1$ for log, x=0 for the trigs). $\log_2$ has a set of such cases ($x = 2^k$, $k \in \mathbb{Z}$) which is equally trivial to handle in binary floating point.

For $f = \log_{10}$, filtering much more difficult. In fact, $y = \log(x)$ is algebraic for exactly all $x = 10^k$, $k \in \mathbb{Z}$ [6]. Filtering means thus testing whether an input x can be written $x = 10^k$ with an integer $k \in \mathbb{Z}$. This is equivalent to testing if $\log_{10}(x)$ an integer, i.e. $\log_{10}(x) \in \mathbb{Z}$. However, since $\log_{10}(x)$ can only be approximated, filtering this way is impossible.

One possibility is the following approach. In floating point arithmetic, in order to be in a situation of difficult rounding, not only $\log_{10}(x)$ must be algebraic but also the corresponding $x = 10^k, k \in \mathbb{Z}$, must be representable in floating point. To start with eliminating cases, we can argue that this impossible for all $k < 0$. Indeed, since $2 \nmid 5$, there exist no $m \in \mathbb{N}$ and $e \in \mathbb{Z}$ for any $k \in \mathbb{Z}^-$ such that $10^k = 2^e \cdot m$ [34]. So we have reduced the range of cases to filter to all $x = 10^k$, $k \in \mathbb{N} \cup \{0\}$. Further in double precision, the mantissa's length is 53. So $10^k = 2^k \cdot 5^k = 2^e \cdot m$ is exactly representable in double precision only for values $k \in \mathbb{N} \cup \{0\}$ such that $5^k \leq 2^{53}$. This yields to $k \leq 53 \cdot \frac{\ln(2)}{\ln(5)} \approx 22.82$; hence $0 \leq k \leq 22$. In consequence, it would be possible to filter out the 23 arguments $x = 10^k$ with $k \in \{0 \ldots 22\}$.

Nevertheless, this approach would be relatively costly. It is not the way that has been chosen for the implementation presented here.

Our approach uses the critical worst case accuracy of the elementary function $\log_{10}(x)$. As already mentioned, it is 2^{-122}. Under the condition that we can provide an approximation to the function that is exact to at least 2^{-123}, we can decide the directed rounding using a modified final rounding sequence: We know that a 1 after a long series of 0s (respectively a 0 after a long series of 1s) must be present at least at the 122th bit of the intermediate mantissa. If it is not, we can consider a potentially present 1 after the 122th bit to be an approximation error artefact. In fact this means neglecting δs relatively less than 2^{-122} when rounding $\diamond(f(x)+\delta)$ instead of $\diamond(f(x))$.

One shortcoming of this approach is that the accurate phase is launched for arguments where the quick phase's accuracy would suffice to return the correct result. As such arguments are extremely rare ($p = \frac{23}{2^{63}} \approx 2.5 \cdot 10^{-18}$!), this is not of an issue. The modification of the final rounding sequence is relatively lightweight: merely one floating point multiplication, two integer masks and one integer comparison have to be added to handle the case.

One remarks this approach is only possible because the critical worst case accuracy of the function is known by Lefèvre's works. Ziv's oignon peeling strategy without the filtering of the 23 possible cases in input and without any accuracy limitation for intermediate computations yields to nontermination of the implementation of the function on such arguments $x = 10^k$.

An earlier `crlibm` implementation of the $\log_{10}(x)$ function based on the SCS format did not handle the problem and returned incorrectly rounded results for inputs $x = 10^k$ in the directed rounding modes.

5.2 General outline of the algorithm and accuracy estimates

The quick phase of the implementation of the $\log_{10}(x)$ follows exactly the scheme depicted by equation (5.1) above. Similarly to the logarithm in base 2, the natural logarithm's intermediate double-double result is multiplied by a double-double precision approximation of *log10inv*. The rounding test is slightly modified in order to ensure safe rounding or launching the accurate phase.

Concerning the accurate phase, some modifications in the natural logarithm's code are necessary because of the tighter accuracy bound needed for the worst case. The natural logarithms accurate phase polynomial approximation relative error has already been less than 2^{-125} which is exact enough for $\log_{10}(x)$. The fact that the complete triple-double implementation is exact to only 119 bits, is mainly due to the inexactness of the operators used in reconstruction phase. In turn, this inexactness is caused by the relatively high overlap in the triple-double numbers handled. By adding two additional renormalisations the triple-double operators become exact enough.

The constant *log10inv* cannot be stored in double-double precision with an accuracy of 124 bits. A triple-double approximation is therefore used. Its relative approximation error is smaller than 2^{-159}. The final multiplication of the triple-double constant representing *log10inv* and the triple-double natural logarithm result is performed by a **Mul33**. The relative error of this operator on non-overlapping triple-doubles is not greater than 2^{-140}. This last operation therefore offers a large accuracy overkill.

TODO The combination of the previous errors should be verified in Gappa.

5.3 Timings

We compare `crlibm`'s portable triple-double implementation for $\log_{10}(x)$ to other correctly rounded and not-correctly rounded implementations. "`crlibm` portable using `scslib`" is the timing for the earlier implementation in `crlibm`, which has been superseded by the one depicted here since version 0.10β. This earlier implementation was completely based on the SCS format and did not contain a quick phase implemented in double precision arithmetic. The values are given in arbitrary units and obtained on a IBM Power 5 processor with gcc 3.3.3 on a Linux Kernel 2.6.5.

On average, our triple-double based implementation is even 10% faster than its incorrectly rounding counterpart on Power. On Pentium, we observe the usual factor 2 with respect to an implementation using double-extended arithmetic. Worst case timings are acceptable in both cases.

Library	avg time	max time
Power5 / Linux-2.6 / gcc-3.3		
`MPFR`	9490	84478
`crlibm` portable using `scslib`	2624	2744
`crlibm` portable using triple-double	60	311
default `libm` (not correctly rounded)	66	71
PentiumM / Linux-2.6 / gcc-4.0		
`crlibm` portable using triple-double	304	1529
default `libm` (not correctly rounded)	153	1904

Table 5.1: Log10 timings on Power5 and PentiumM architectures

Chapter 6

The exponential

This chapter is contributed by Ch. Q. Lauter.

6.1 Overview of the algorithm

The exponential function allows for additive argument reduction with multiplicative reconstruction: $e^{a+b} = e^a \cdot e^b$. In particular, the following equation is useful in developing an argument reduction:

$$e^x = e^{E\cdot\ln(2)+z} = \left(e^{\ln(2)}\right)^E \cdot e^z = 2^E \cdot e^z$$

Here, E can be considered to be a signed integer, $E \in \mathbb{Z}$ and z to be a reduced argument such that $|z| < \ln(2)$. One remarks that the use of such an argument reduction implies a multiplication with a transcendental constant, $\ln(2)$. This means that the reduced argument will not be exact. The corresponding error bound will be given in section 6.3.

A reduced argument obtained by the reduction shown above is generally still to great for polynomial approximation. By use of tabulation methods, the following argument reduction can be employed and yields to smaller reduced arguments.

$$\begin{aligned} k &= \left\lfloor x \cdot \frac{2^l}{\ln(2)} \right\rceil \\ \hat{r} &= x - k \cdot \frac{\ln(2)}{2^l} \\ k &= 2^l \cdot M + 2^{w_1} \cdot i_2 + i_1 \end{aligned}$$

where $\hat{r}$ is the reduced argument, $k, M \in \mathbb{N}$ are intermediate integers, and $w_1, w_2 \in \mathbb{N}$, $l = w_1 + w_2$, are the widths of the indices to the two tables. The corresponding reconstruction phase is

$$e^x = 2^M \cdot 2^{\frac{i_2}{2^{w_1}}} \cdot 2^{\frac{i_1}{2^l}} \cdot e^{\hat{r}} = 2^M \cdot t_1 \cdot t_2 \cdot e^{\hat{r}}$$

with the table values $t_1 = 2^{\frac{i_2}{2^{w_1}}}$ and $t_2 = 2^{\frac{i_1}{2^l}}$. The argument reduction ensures that $|\hat{r}| \leq \frac{\ln(2)}{2^l} < 2^{-l}$. This magnitude is small enough for allowing for polynomial approximation.

In the case of the given algorithm, we use $l = 12$ and $w_1 = w_2 = 6$.

The subtraction $\hat{r} = x - k \cdot \circ\left(\frac{\ln(2)}{2^l}\right)$ can be implemented exactly but leads to catastrophic cancellation that amplifies the absolute error of the potentially exact multiplication of k by the approximated $\frac{\ln(2)}{2^l}$. Is is nevertheless not of such an issue, as will be shown in section 6.6.

6.2 Special case handling

The exponential function e^x, which is monotone increasing, produces results that are representable in a floating point format for arguments

$$x \in [underflowBound; overflowBound]$$

Herein the following values are observed for double precision:

$$underflowBound = \circ\left(\ln\left(2^{-1075}\right)\right) \approx -745.13$$

and

$$overflowBound = \circ\left(\ln\left(2^{1024} \cdot \left(1 - 2^{-53}\right)\right)\right) \approx 709.78$$

Its double precision result is gradually underflowed in the argument domain

$$x \in [underflowBound; denormBound]$$

where

$$denormBound = \circ\left(\ln\left(2^{-1022}\right)\right) \approx -708.40$$

for double precision. No special case can therefore occur for arguments x such that

$$|x| \leq \min\left(|underflowBound|, |overflowBound|, |denormBound|\right) \approx 708.40$$

This provides us a very efficient filter for the greatest part of the definition domain. Since also some arguments are filtered out that actually do not represent a special case, additional tests are made in the body of the main special handling sequence launched by the filter. Particular cases like $x = \pm\infty$, $x = \text{NaN}$ are handled as follows:

- The result for $x = \text{NaN}$ is NaN.
- The result for $x = +\infty$ is $+\infty$.
- The result for $x = -\infty$ is 0, even in round-upwards mode.

If the result is clearly underflowed, 0 is returned with the inexact flag set for round-to-nearest and round-downwards, 0^+ is returned for round-upwards.

The ordering of double precision numbers is compatible with the integer ordering of the signed 64 bit integers the floating point numbers can be read in memory as. Further the ordering on numbers is equal to the lexicographic ordering of its digits. So if the higher order word of $|x|$ is less than the higher order word of $overUnderDenormBound = \min\left(|underflowBound|, |overflowBound|, |denormBound|\right)$, no underflow, gradual underflow or overflow can occur. In the other cases, special handling in the reconstruction step may be necessary. A flag `mightBeDenorm` is set for possible subnormals. Cases where intermediate results are not representable because of overflow but where the final result is representable can be overcome by replacing the corresponding floating point multiplication by an integer manipulation. This will be shown below.

Since $e^x = 1 + x + \mathcal{O}\left(x^2\right)$ for small $|x|$, underflowed arguments x ($|x| \leq 2^{-1022}$) can be handled as follows:

- In round-to-nearest mode, 1 can be returned since $\circ(1 + x) = 1$ for $|x| \leq 2^{-54}$.
- In round-upwards mode, 1 must be returned for $x = 0$. For $x < 0$, 1 can be returned, too, because $\triangle(1 - |x|) = 1$. If $x > 0$, 1^+ must be returned.
- Round-downwards mode is analogous to round-upwards mode, but the signs are inverted.
- Round-towards-zero mode is equivalent to round-downwards mode since $e^x > 0 \ \ \forall x \in \mathbb{R}$.

The following sequence realizes the special case handling for round-to-nearest. The sequences for the other rounding modes are straightforward and will not be shown here. The constants `OVRUDRFLWSMPLBOUND`, `OVRFLWBOUND` and `DENORMBOUND` are computed by the corresponding Maple script that realizes the equations given above.

Listing 6.1: Handling special cases

```
/* Special cases tests */
xIntHi = xdb.i[HI];
mightBeDenorm = 0;
/* Test if argument is a denormal or zero */
if ((xIntHi & 0x7ff00000) == 0) {
  /* We are in the RN case, return 1.0 in all cases */
  return 1.0;
}

/* Test if argument is greater than approx. 709 in magnitude */
if ((xIntHi & 0x7fffffff) >= OVRUDRFLWSMPLBOUND) {
  /* If we are here, the result might be overflowed, underflowed, inf, or NaN */

  /* Test if +/- Inf or NaN */
  if ((xIntHi & 0x7fffffff) >= 0x7ff00000) {
    /* Either NaN or Inf in this case since exponent is maximal */

    /* Test if NaN: mantissa is not 0 */
    if (((xIntHi & 0x000fffff) | xdb.i[LO]) != 0) {
        /* x = NaN, return NaN */
        return x + x;
    } else {
        /* +/- Inf */

        /* Test sign */
        if ((xIntHi & 0x80000000)==0)
          /* x = +Inf, return +Inf */
          return x;
        else
          /* x = -Inf, return 0 */
          return 0;
    } /* End which in NaN, Inf */
  } /* End NaN or Inf ? */

  /* If we are here, we might be overflowed, denormalized or underflowed in the result
     but there is no special case (NaN, Inf) left */

  /* Test if actually overflowed */
  if (x > OVRFLWBOUND) {
    /* We are actually overflowed in the result */
    return LARGEST * LARGEST;
  }

  /* Test if surely underflowed */
  if (x <= UNDERFLWBOUND) {
    /* We are actually sure to be underflowed and not denormalized any more
         So we return 0 and raise the inexact flag */
    return SMALLEST * SMALLEST;
  }

  /* Test if possibly denormalized */
  if (x <= DENORMBOUND) {
    /* We know now that we are not sure to be normalized in the result
         We just set an internal flag for a further test
    */
    mightBeDenorm = 1;
  }
} /* End might be a special case */
```

6.3 Argument reduction

Mathematically, the argument reduction used for quick and accurate phase is the same. The reduced argument is nevertheless different for the both phases because the reduction is inexact. It is therefore implemented in two or three phases:

First, k corresponding to $\left\lfloor x \cdot \frac{2^{12}}{\ln(2)} \right\rceil$ is computed in both integer and floating point (double) representation. Herein $\frac{2^{12}}{\ln(2)}$ is represented in double precision and the multiplication by x is performed in double precision, too. Then the nearest integer k to the number obtained is computed. This computational problem is principally subject to the Table Maker's Dilemma: $x \cdot \frac{2^{12}}{\ln(2)}$ is transcendental for all $x \neq \ln(2)$ and the to-nearest-integer operation is equivalent to a rounding to the nearest. It is not

possible to guarantee that the computed k is really the integer nearest to $x \cdot \frac{2^{12}}{\ln(2)}$. Nevertheless, this is not a problem. The following argument reduction steps, as $\hat{r} = x - k \cdot \frac{\ln(2)}{2^{12}}$, and the reconstruction are mathematically correct for any k. So computing the wrong k only leads to a slight enlargement of the interval of the reduced argument $\hat{r}$, as one can see with the following argument:

The function's argument's value x is bounded in magnitude by 2^{10}. So, $x \cdot \frac{2^{12}}{\ln(2)}$ is bounded in magnitude by 2^{22}. Since the accuracy of the value $xMultLog2InvMult2L = \circ\left(x \cdot \circ\left(\frac{2^{12}}{\ln(2)}\right)\right)$ with respect to the exact value $x \cdot \frac{2^{12}}{\ln(2)}$ is at least 51 bits, an error is made not earlier than at the 19th bit following the integer-fractional point. Thus

$$\left|\frac{k - x \cdot \frac{2^{12}}{\ln(2)}}{x \cdot \frac{2^{12}}{\ln(2)}}\right| \leq \frac{1}{2} + 2^{-19}$$

Therefore $\hat{r}'$ is actually bounded by $|\hat{r}'| \leq \frac{\ln(2)}{2^{12}} \cdot \left(\frac{1}{2} + 2^{-19}\right)$ instead of $|\hat{r}| \leq \frac{\ln(2)}{2^{12}} \cdot \frac{1}{2}$. This difference can be taken into account in the error computation, and we may safely assume that its effects will be negligible.

Note further that k is computed exactly ($k = 0$) for $|x| \leq 2^{-14}$ because no Table Maker's Dilemma can no longer occur.

This first step is performed by the following code sequence:

Listing 6.2: Argument reduction - first step

```
1 xMultLog2InvMult2L = x * log2InvMult2L;
2 shiftedXMult = xMultLog2InvMult2L + shiftConst;
3 kd = shiftedXMult - shiftConst;
4 shiftedXMultdb.d = shiftedXMult;
5 k = shiftedXMultdb.i[LO];
```

Here, `k` and `kd` represent k in integer and floating point (double) format. The technique for computing $\lfloor z \rceil$ out of z is explained in section 2.2.3, page 20.

In the second step of the argument reduction, an arithmetical approximation to $\hat{r} = x - k \cdot \frac{\ln(2)}{2^{12}}$, $r_h + r_m = \hat{r} + \delta_{\text{argred}}$, is computed using a double-double approximation to $\frac{\ln(2)}{2^{12}}$ and double-double precision for computations. We will consider its absolute error $\delta_{\text{argredquick}} = r_h + r_m - \hat{r}$ and the resulting relative error on the exponential function $\varepsilon_{\text{argredquick}} = \frac{e^{\hat{r}+\delta} - e^{\hat{r}}}{e^{\hat{r}}}$ below.

This argument reduction step is implemented as follows:

Listing 6.3: Argument reduction - second step

```
1 Mul12(&s1,&s2,msLog2Div2Lh,kd);
2 s3 = kd * msLog2Div2Lm;
3 s4 = s2 + s3;
4 s5 = x + s1;
5 Add12Cond(rh,rm,s5,s4);
```

Here, `msLog2Div2Lh` and `msLog2Div2Lm` are a double-double representing $-\frac{\ln(2)}{2^{12}}$ with an relative error $|\varepsilon_{\text{logconstquick}}| = \left|\frac{msLog2Div2L_h + msLog2Div2L_m + 2^{-12} \cdot \ln(2)}{-2^{-12} \cdot \ln(2)}\right| \leq 2^{-109}$. Further, $|msLog2Div2L_h| \leq 2^{-54} \cdot |msLog2Div2L_m|$.

Let us now show first a bound on the absolute round-off error of the given sequence. We know that the multiplication $s_1 + s_2 = kd \cdot msLog2Div2h$ and the final addition $r_h + r_m = s_5 + s_4$ are exact. Further, the addition $s_5 = x \oplus s_1$ is exact as per Sterbenz' lemma. In fact, since

$$\left|x - \left\lfloor x \cdot \frac{2^{12}}{\ln(2)} \right\rceil \cdot \frac{\ln(2)}{2^{12}}\right| \leq 2^{-12}$$

and

$$s_1 = -\left\lfloor x \cdot \frac{2^{12}}{\ln(2)} \right\rceil \cdot \frac{\ln(2)}{2^{12}} \cdot (1 + \varepsilon')$$

with $|\varepsilon'| \leq 2^{-50}$, it is trivial to see that

$$\frac{1}{2} \cdot |s_1| \leq |x| \leq 2 \cdot |s_1|$$

In addition, s_1 and x are clearly of opposed sign. So, noting

$$s_3 = k \cdot msLog2Div2L_m \cdot (1 + \varepsilon_1)$$

and

$$s_4 = (s_2 + s_3) \cdot (1 + \varepsilon_2)$$

with $|\varepsilon_1| \leq 2^{-53}$ and $|\varepsilon_2| \leq 2^{-53}$, we get

$$r_h + r_m = x + k \cdot (msLog2Div2L_h + msLog2Div2L_m) + \delta'$$

with

$$\left|\delta'\right| \leq |\varepsilon_1| \cdot |k \cdot msLog2Div2L_m| + |\varepsilon_2| \cdot |s_2 + s_3|$$

We have

$$|s_2| \leq 2^{-53} \cdot |s_1| \leq 2^{-53} \cdot |\circ (k \cdot msLog2Div2L_h)| \leq 2^{-52} \cdot |k \cdot msLog2Div2L_h|$$

and further

$$|s_3| \leq |k \cdot msLog2Div2L_m\,(1 + \varepsilon_1)| \leq |k \cdot msLog2Div2L_m| + |\varepsilon_1| \cdot |k \cdot msLog2Div2L_m|$$

So one can easily check that

$$\left|\delta'\right| \leq 2^{-104} \cdot |k \cdot msLog2Div2Lh|$$

In consequence using the fact that $|msLog2Div2L_h| = \left|\circ\left(\frac{\ln(2)}{2^{12}}\right)\right| \leq 2 \cdot \left|\frac{\ln(2)}{2^{12}}\right|$ and the bound for k, one obtains

$$\left|\delta'\right| \leq \left|\left(x \cdot \frac{2^{12}}{\ln(2)} + \frac{1}{2} + 2^{-19}\right) \cdot \frac{\ln(2)}{2^{12}}\right|$$

Since $|x| \leq 746$ after filtering out the special cases, we obtain $|\delta'| \leq 2^{-103} \cdot 2^{10} = 2^{-93}$.

To this round-off error adds the approximation error comitted by rounding $\frac{\ln(2)}{2^{12}}$ to a double-double. We can note

$$r_h + r_m = x - k \cdot \frac{\ln(2)}{2^{12}} \cdot (1 + \varepsilon_{\text{logconstquick}}) + \delta'$$

This gives us

$$r_h + r_m = \hat{r} + \delta$$

with $|\delta| \leq \left|k \cdot \frac{\ln(2)}{2^{12}} \cdot \varepsilon_{\text{logconstquick}}\right| + |\delta'|$. One can easily check that one obtains thus finally $|\delta| \leq 2^{-92}$.

This absolute error δ in the reduced argument $\hat{r}$ translates to a relative error $\varepsilon_{\text{argredquick}}$ in the function $e^{\hat{r}}$ as follows:

$$\begin{aligned} e^r &= e^{\hat{r}+\delta} \\ &= e^{\hat{r}} \cdot e^{\delta} \\ &= e^{\hat{r}} \cdot \sum_{i=0}^{\infty} \frac{1}{i!} \cdot \delta^i \\ &= e^{\hat{r}} \cdot \left(1 + \sum_{i=1}^{\infty} \frac{1}{i!} \cdot \delta^i\right) \\ &= e^{\hat{r}} \cdot (1 + \varepsilon_{\text{argredquick}}) \end{aligned}$$

with $\varepsilon_{\text{argredquick}} = \sum_{i=1}^{\infty} \frac{1}{i!} \cdot \delta^i = \delta \cdot \sum_{i=0}^{\infty} \frac{1}{(i+1)!} \cdot \delta^i$. Since $|\delta| < \frac{1}{2}$, we get

$$\forall i \geq 0\,.\ \left|\frac{1}{(i+1)!} \cdot \delta^i\right| \leq \left(\frac{1}{2}\right)^i$$

In consequence, $\sum_{i=0}^{\infty} \frac{1}{(i+1)!} \cdot \delta^i \leq \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i = 2$.

Thus we get $|\varepsilon_{\text{argredquick}}| \leq 2 \cdot |\delta| \leq 2^{-91}$.

Still in the second step of the argument reduction, the values M, i_1 and i_2 are computed exactly in integer computation as follows:

Listing 6.4: Argument reduction - second step (cont'd)

```
M = k >> L;
index1 = k & INDEXMASK1;
index2 = (k & INDEXMASK2) >> LHALF;
```

Here `L` is equal to 12 and `LHALF` is equal to 6. The values `INDEXMASK1` and `INDEXMASK2` are masks to the lowest 6 bits and respectively to bits 6 through 15 of an 32 bit word.

If ever the accurate phase must be launched, a third argument reduction phase is performed. It computes a triple-double $r_h + r_m + r_l = \hat{r} + \delta$ such that the resulting relative error on the exponential function $\varepsilon_{\text{argredaccurate}} = \frac{e^{\hat{r}+\delta} - e^{\hat{r}}}{e^{\hat{r}}}$ is bounded by $|\varepsilon_{\text{argredaccurate}}| \leq 2^{-140}$ as will be shown below. This step uses a triple-double approximation to $\frac{\ln(2)}{2^{12}}$ with a relative error

$$|\varepsilon_{\text{logconstaccurate}}| = \left| \frac{msLog2Div2L_h + msLog2Div2L_m + msLog2Div2L_l + 2^{-12} \cdot \ln(2)}{-2^{-12} \cdot \ln(2)} \right| \leq 2^{-163}$$

It is implemented as follows:

Listing 6.5: Argument reduction - third step

```
Mul133(&msLog2Div2LMultKh,&msLog2Div2LMultKm,&msLog2Div2LMultKl,kd,msLog2Div2Lh,msLog2Div2Lm,
    msLog2Div2Ll);
t1 = x + msLog2Div2LMultKh;
Add12Cond(rh,t2,t1,msLog2Div2LMultKm);
Add12Cond(rm,rl,t2,msLog2Div2LMultKl);
```

The values for k, M, i_1 and i_2 that have been exactly computed already at the second argument reduction step can of course be reused.

All operation but the first `Mul133` operator are exact: the last two by their general properties, the addition $t_1 = x \oplus msLog2Div2LMultK_h$ as per Sterbenz' lemma analogously as in the second argument reduction step. So one can note

$$r_h + r_m + r_l = x - k \cdot (msLog2Div2L_h + msLog2Div2L_m + msLog2Div2L_l) \cdot (1 + \varepsilon_1)$$

where ε_1 is the relative error bound of the `Mul133` operator, for instance $|\varepsilon_1| \leq 2^{-153}$. Integrating also the rounding error in the constant, we get

$$r_h + r_m + r_l = x - k \cdot \frac{\ln(2)}{2^{12}} + \delta$$

with $|\delta| \leq \left| k \cdot \frac{\ln(2)}{2^{12}} \right| \cdot \left(2^{-163} + 2^{-153}\right) \leq 2^{-152} \cdot \left| k \cdot \frac{\ln(2)}{2^{12}} \right| \leq 2^{-141}$. One checks the previous upper bounds by using analogous arguments as the ones given for the second argument reduction step. Once again, this absolute error δ translates to a relative error $\varepsilon_{\text{argredaccurate}} = \frac{e^{\hat{r}+\delta} - e^{\hat{r}}}{e^{\hat{r}}}$ in a similar way as mentioned above. We get

$$|\varepsilon_{\text{argredaccurate}}| \leq 2^{-140}$$

which is the bound to prove.

Let us still remark that argument reduction is exact for arguments x such that $|x| \leq \frac{\ln(2)}{2^{13}} < 2^{-13}$. In fact, in this case, $k = 0$ which implies that all multiplications of k by the constants for $\frac{\ln(2)}{2^{12}}$ are exact.

According to their uses either in quick or accurate phase, the table values $t_1 = e^{\frac{i_1}{2^{12}}}$ and $t_2 = e^{\frac{i_1}{2^{12}}}$ for $i_1, i_2 \in \{0 \ldots 63\}$ are read as a double-double or as a triple-double. By construction, the double-double values have a relative error $\varepsilon_{\text{tablequick}} = \frac{tbl_{ih} + tbl_{im} - t_i}{t_i}$ bounded by $|\varepsilon_{\text{tablequick}}| \leq 2^{-106}$. The triple-double values with $\varepsilon_{\text{tableaccurate}} = \frac{tbl_{ih} + tbl_{im} + tbl_{il} - t_i}{t_i}$ verify $|\varepsilon_{\text{tableaccurate}}| \leq 2^{-159}$. For $i_1 = 0$ and $i_2 = 0$ both errors are equal to 0, so both argument reduction and reconstruction steps are exact for argument x that verify $|x| \leq 2^{-13}$.

6.4 Polynomial approximation and reconstruction

In both quick and accurate phase, the reduced argument $r_h + r_m$ respectively $r_h + r_m + r_l$ corresponding to their mathematical equivalent $\hat{r}$ are bounded by $|\hat{r}| \leq \frac{\ln(2)}{2^{12}} \cdot \left(2^{-1} + 2^{-19}\right) + \delta < 2^{-13}$. After sim-

plification by $e^{r_h+r_m+r_l} = e^{r_h} \cdot e^{r_m} \cdot e^{r_l}$, $e^{r_h} - 1$ is approximated by a polynomial of degree 4 or 7. The functions $e^{r_m} - 1$ and potentially $e^{r_l} - 1$ are both approximated linearly by r_m or, respectively, r_l.

Concerning the approximation errors for e^{r_m} and e^{r_l}, one can note that $|r_m| \leq 2^{-52} \cdot 2^{-13} \leq 2^{-65}$ and $|r_l| \leq 2^{-105} \cdot 2^{-13} \leq 2^{-118}$. In consequence $\varepsilon_{\text{approxargmiddle}} = \frac{r_m - (e^{r_m}-1)}{(e^{r_m}-1)}$ and $\varepsilon_{\text{approxarglower}} = \frac{r_l - (e^{r_l}-1)}{(e^{r_l}-1)}$ are bounded by $|\varepsilon_{\text{approxargmiddle}}| \leq 2^{-66}$ and $|\varepsilon_{\text{approxarglower}}| \leq 2^{-119}$.

6.4.1 Quick phase polynomial approximation and reconstruction

In quick phase $e^{r_h} - 1$ is approximated by a polynomial $p(r_h)$ of the following form

$$p(r_h) = r_h + \frac{1}{2} \cdot r_h^2 + c_3 \cdot r_h^3 + c_4 \cdot r_h^4$$

The coefficients c_3 and c_4 are stored in double precision, $\frac{1}{2}$ is exactly representable. For r_h bounded as indicated above, the relative approximation error $\varepsilon_{\text{approxarghighquick}} = \frac{p(r_h) - e^{r_h} + 1}{e^{r_h} - 1}$ is bounded by $|\varepsilon_{\text{approxarghighquick}}| \leq 2^{-62}$.

The polynomial $p(r_h) - r_h = \frac{1}{2} \cdot r_h^2 + c_3 \cdot r_h^3 + c_4 \cdot r_h^4$ is evaluated in double precision using the following scheme:

1. In the beginning, an approximation to r_h^2, *rhSquare*, is computed. Concurrently, c_3 is multiplied by r_h yielding to $rhC3 = c_3 \cdot r_h \cdot (1 + \varepsilon)$.
2. At the first step, the squared argument *rhSquare* is multiplied by $\frac{1}{2}$ yielding to the approximation $rhSquareHalf$. At the same time, it is multiplied by *rhC3* which results in *monomialCube* approximating now $c_3 \cdot r_h^3$. Still in the same moment, it is squared once again yielding to *rhFour* which approximates thus r_h^4.
3. At the next dependency top, *rhFour* is multiplied by c_4. This results in *monomialFour* – a value corresponding to $c_4 \cdot r_h^4$.
4. In the next moment, *monomialCube* and *monomialFour* are added together. This gives a value approximating $c_3 \cdot r_h^3 + c_4 \cdot r_h^4$.
5. Finally, $rhSquareHalf$ is added to the value obtained at the previous step yielding to an arithmetical approximation of $\frac{1}{2} \cdot r_h + c_3 \cdot r_h^3 + c_4 \cdot r_h^4$, stored in *highPolyWithSquare*.

This scheme does not completely exploit the possible parallelism for purpose of not deteriorating too much the final accuracy. It is implemented as follows:

Listing 6.6: Quick phase polynomial evaluation - high order terms

```
1  rhSquare = rh * rh;
2  rhC3 = c3 * rh;
3
4  rhSquareHalf = 0.5 * rhSquare;
5  monomialCube = rhC3 * rhSquare;
6  rhFour = rhSquare * rhSquare;
7
8  monomialFour = c4 * rhFour;
9
10 highPoly = monomialCube + monomialFour;
11
12 highPolyWithSquare = rhSquareHalf + highPoly;
```

The following steps must still add the linear term r_h of the polynomial ($p(r_h) = r_h + highPolyWithSquare$) and reconstruct the exponential as

$$e^x = 2^M \cdot (tbl_{1h} + tbl_{1m}) \cdot (tbl_{2h} + tbl_{2m}) \cdot (1 + r_h + highPolyWithSquare) \cdot (1 + r_m) + \delta$$

In order to allow for increasing speed by approximating different terms, they are implemented as this:

1. The two table values $tbl_{1h} + tbl_{1m}$ and $tbl_{2h} + tbl_{2m}$ are multiplied using a double-double multiplication operator yielding to $tables_h + tables_l = (tbl_{1h} + tbl_{1m}) \cdot (tbl_{2h} + tbl_{2m}) \cdot (1 + \varepsilon)$.

2. The term
   $(1 + r_h + highPolyWithSquare) \cdot (1 + r_m) =$
   $1 + r_h + highPolyWithSquare + r_m + r_h \cdot r_m + highPolyWithSquare \cdot r_m$
   is approximated by neglecting the quadratic terms. First r_h and $highPolyWithSquare$ are added together in double precision. Their sum is then added to r_h in double precision, too. This yields to the intermediate value t_9. The addition with 1 is not explicited.

3. The multiplication $(tables_h + tables_l) \cdot (1 + t_9)$ is approximated by $tables_h + tables_l + tables_h \cdot t_9$. The multiplication $tables_h \cdot t_9$ is performed in double precision. It produces t_{10}.

4. The addition $tables_h + tables_l + t_{10}$ is carried out in double-double precision. Its result $polyTbl_h + polyTbl_m$ approximates finally

   $$polyTbl_h + polyTbl_m = (tbl_{1h} + tbl_{1m}) \cdot (tbl_{2h} + tbl_{2m}) \cdot (1 + p(r_h)) \cdot (1 + r_m) \cdot (1 + \varepsilon)$$

   for some relative error ε.

The steps explained above are implemented in the code as follows:

Listing 6.7: Quick phase reconstruction

```
1  Mul22(&tablesh,&tablesl,tbl1h,tbl1m,tbl2h,tbl2m);
2
3  t8 = rm + highPolyWithSquare;
4  t9 = rh + t8;
5
6  t10 = tablesh * t9;
7
8  Add12(t11,t12,tablesh,t10);
9  t13 = t12 + tablesl;
10 Add12(polyTblh,polyTblm,t11,t13);
```

The final result for the exponential is $e^x \approx 2^M \cdot (polyTbl_h + polyTbl_m)$. On this value, a rounding test would have to be performed. Since gradual underflow is excluded by special case handling, the multiplication by 2^M is exact. In consequence, it is possible to do the rounding test on $polyTbl_h + polyTbl_m$ and to multiply then by 2^M. This is the way chosen in the given implementation. See section 6.5, page 84, for further details.

6.4.2 Accurate phase polynomial approximation and reconstruction

In accurate phase, $e^{r_h} - 1$ is approximated by a polynomial $p(r_h)$ of degree 7. It has the following form:

$$p(r_h) = r_h + \frac{1}{2} \cdot r_h^2 + r_h^3 \cdot ((c_{3h} + c_{3l}) + r_h \cdot ((c_{4h} + c_{4l}) + r_h \cdot (c_5 + r_h \cdot (c_6 + r_h \cdot c_7))))$$

Coefficients $c_{3h} + c_{3l}$ and $c_{4h} + c_{4l}$ are stored as double-double numbers, coefficients c_5 through c_7 are stored in double precision.

For r_h bounded by $|r_h| \leq 2^{-13}$, the relative approximation error $\varepsilon_{approxarghighaccu} = \frac{p(r_h) - e^{r_h} + 1}{e^{r_h} - 1}$ is bounded by $|\varepsilon_{approxarghighaccu}| \leq 2^{-113}$. For r_h such that $|r_h| \leq 2^{-30}$, $\varepsilon_{approxarghighaccu}$ is bounded by $|\varepsilon_{approxarghighaccu}| \leq 2^{-160}$.

The polynomial $p(r_h)$ is evaluated as follows:

1. The high order terms of $p(r_h)$, $c_5 + r_h \cdot (c_6 + r_h \cdot c_7)$ are evaluated in double precision using Horner's scheme. The result of this evaluation is stored in $highPoly \approx c_5 + r_h \cdot (c_6 + r_h \cdot c_7)$.

2. The multiplication $r_h \cdot highPoly$ is implemented using an exact operator and yields to a double-double $t_{1h} + t_{1l}$. The following steps leading to $t_{4h} + t_{4l} \approx (c_{3h} + c_{3l}) + r_h \cdot ((c_{4h} + c_{4l}) + (t_{1h} + t_{1l}))$ are implemented using double-double computations and Horner's scheme.

3. The sum of the low order terms $r_h + \frac{1}{2} \cdot r_h^2$, stored in a non-overlapped triple-double $lowPoly_h + lowPoly_m + lowPoly_l$, is computed exactly as follows: first r_h is squared exactly using an exact multiplication operator producing $rhSquare_h + rhSquare_l = r_h \cdot r_h$. Since for arguments x to the exponential function such that $|x| \leq 2^{-55}$ rounding is trivial in all rounding modes, we can suppose that $|x\rangle\, 2^{-55}$. In consequence, since x is a double precision number, the reduced argument r_h

is either equal to 0 or greater in magnitude than 2^{-58}. So both $rhSquare_h$ and $rhSquare_l$ are either exactly 0 or greater in magnitude than $\left(2^{-58}\right)^2 \cdot 2^{-53} = 2^{-169}$. They are thus never subnormal. Therefore the pairwise multiplication of $rhSquare_h + rhSquare_l$ by $\frac{1}{2}$ yielding to $rhSquareHalf_h + rhSquareHalf_l = \frac{1}{2} \cdot r_h^2$ is exact. Since r_h is such that $|r_h| \leq 2^{-13}$, $r_h + rhSquareHalf_h + rhSquareHalf_l$ can be considered as a partially overlapped triple-double number. The overlap bound is such that $|rhSquareHalf_h| \leq 2^{-12} \cdot |r_h|$ and $|rhSquareHalf_l| \leq 2^{-53} \cdot |rhSquareHalf_h|$. Thus it is possible to use the **Renormalize3** sequence [25] to obtain a non-overlapped triple-double $lowPoly_h + lowPoly_m + lowPoly_l$ which is exactly equal to $r_h + \frac{1}{2} \cdot r_h^2$ because the renormalization operator is exact and all preceeding operations have been, too.

4. An triple-double approximation to r_h^3, stored in $rhCube_h + rhCube_m + rhCube_l$ is computed by multiplying r_h by the exact $rhSquare_h + rhSquare_l$ computed in the previous step. For this operation the **Mul23** sequence is used.

5. The approximation of the high order terms of the polynomial, $t_{4h} + t_{4l}$ is then multiplied by $rhCube_h + rhCube_m + rhCube_l$ using a triple-double multiplication operator. The result of this operation is added to $lowPoly_h + lowPoly_m + lowPoly_l$ in order to obtain a potentially overlapped triple-double $p_h + p_m + p_l$ approximating $p(r_h)$.

All this steps are implemented by the following code:

Listing 6.8: Accurate phase polynomial approximation

```
highPoly = accPolyC5 + rh * (accPolyC6 + rh * accPolyC7);

Mul12(&t1h,&t1l,rh,highPoly);
Add22(&t2h,&t2l,accPolyC4h,accPolyC4l,t1h,t1l);
Mul22(&t3h,&t3l,rh,0,t2h,t2l);
Add22(&t4h,&t4l,accPolyC3h,accPolyC3l,t3h,t3l);

Mul12(&rhSquareh,&rhSquarel,rh,rh);
Mul23(&rhCubeh,&rhCubem,&rhCubel,rh,0,rhSquareh,rhSquarel);

rhSquareHalfh = 0.5 * rhSquareh;
rhSquareHalfl = 0.5 * rhSquarel;

Renormalize3(&lowPolyh,&lowPolym,&lowPolyl,rh,rhSquareHalfh,rhSquareHalfl);

Mul233(&highPolyMulth,&highPolyMultm,&highPolyMultl,t4h,t4l,rhCubeh,rhCubem,rhCubel);

Add33(&ph,&pm,&pl,lowPolyh,lowPolym,lowPolyl,highPolyMulth,highPolyMultm,highPolyMultl);
```

For reconstructing e^x out of the polynomial approximation of $e^{r_h} - 1$, $e^{r_m} - 1$, $e^{r_l} - 1$ and the table values $tbl_{1h} + tbl_{1m} + tbl_{1l}$ and $tbl_{2h} + tbl_{2m} + tbl_{2l}$ the following term must be approximated:

$$e^x \approx 2^M \cdot (tbl_{1h} + tbl_{1m} + tbl_{1l}) \cdot (tbl_{2h} + tbl_{2m} + tbl_{2l}) \cdot (1 + (p_h + p_m + p_l)) \cdot (1 + r_m) \cdot (1 + r_l)$$

First, the following approximation is possible

$$(1 + (p_h + p_m + p_l)) \cdot (1 + r_m) \cdot (1 + r_l) \approx 1 + (p_h + p_m + p_l + r_m + r_l + (r_m + r_l) \cdot (p_h + p_m))$$

First an arithmetical approximation $fullPoly_h + fullPoly_m + fullPoly_l$ to $p_h + p_m + p_l + r_m + r_l + (r_m + r_l) \cdot (p_h + p_m)$ is computed as follows:

1. By means of an exact, unconditional addition, p_h and p_m, which may be potentially overlapped because of being the higher significant parts of an overlapped triple-double, are renormalized.

2. They are then multiplied by the non-overlapping $r_m + r_l$ using a double-double multiplication operator. The result of this product is added to $r_m + r_l$ in double-double precision.

3. The result of this addition, $q_h + q_l$, approximates thus $r_m + r_l + (r_m + r_l) \cdot (p_h + p_m)$.

4. The triple-double addition operator **Add233** allows then for adding $p_h + p_m + p_l$ to $q_h + q_l$ resulting in $fullPoly_h + fullPoly_m + fullPoly_l$.

Then 1 is added to $fullPoly_h + fullPoly_m + fullPoly_l$ by the following code sequence:

Listing 6.9: Addition with 1

```
Add12(polyAddOneh, t5, 1, fullPolyh);
Add12Cond(polyAddOnem, t6, t5, fullPolym);
polyAddOnel = t6 + fullPolyl;
```

Since the **Add12** operator is exact, the round-off error of this adding of 1 is equal to the round-off error in the last addition $polyAddOne_l = t_6 \oplus fullPoly_l$. In absolute value, it is always less than $2^{-53} \cdot |t_6 + fullPoly_l|$. Since $|r_h| \leq 2^{-12}$, one checks that $\frac{1}{2} < polyAddOne_h < 2$. So the relative error of this addition with 1, $\varepsilon_{\text{addOne}}$ is in magnitude less than $2^{-52} \cdot |t_6 + fullPoly_l|$. This is particularly important when considering high critical precision worst cases, see section 6.6, page 88.

The result of the polynomial approximation,

$$polyWithOne_h + polyWithOne_m + polyWithOne_l \approx e^{r_h} \cdot e^{r_m} \cdot e^{r_l}$$

is then multiplied in two steps first by $tbl_{1h} + tbl_{1m} + tbl_{2l}$ and then by $tbl_{2h} + tbl_{2m} + tbl_{2l}$ using each time the triple-double multiplication operator **Mul33**. Remark that these multiplications are exact for arguments $|x| \leq 2^{-14}$ because in this case, $k = 0$, $i_1 = 0$ and $i_2 = 0$ which implies that $tbl_{1h} = 1$, $tbl_{2h} = 1$ and $tbl_{im} = 0$ and $tbl_{il} = 0$. In fact machine multiplications by 1 are always exact.

The final product of this multiplications may be overlapped and is therefore renormalized using the **Renormalize3** sequence. This yields to the non-overlapped triple-double $polyTbl_h + polyTbl_m + polyTbl_l$ approximating $\frac{e^x}{2^M}$.

All these steps are implemented by the following code sequence:

Listing 6.10: Accurate phase reconstruction

```
Add12(phnorm,pmnorm,ph,pm);
Mul22(&rmlMultPh,&rmlMultPl,rm,rl,phnorm,pmnorm);
Add22(&qh,&ql,rm,rl,rmlMultPh,rmlMultPl);

Add233Cond(&fullPolyh,&fullPolym,&fullPolyl,qh,ql,ph,pm,pl);
Add12(polyAddOneh, t5, 1, fullPolyh);
Add12Cond(polyAddOnem, t6, t5, fullPolym);
polyAddOnel = t6 + fullPolyl;
Mul33(&polyWithTbl1h,&polyWithTbl1m,&polyWithTbl1l,tbl1h,tbl1m,tbl1l,polyAddOneh,polyAddOnem,
    polyAddOnel);
Mul33(&polyWithTableshesh,&polyWithTableshesm,&polyWithTableshesl,
        tbl2h,tbl2m,tbl2l,
        polyWithTbl1h,polyWithTbl1m,polyWithTbl1l);

Renormalize3(polyTblh,polyTblm,polyTbll,polyWithTableshesh,polyWithTableshesm,polyWithTableshesl);
```

The multiplication of $polyTbl_h + polyTbl_m + polyTbl_l$ by 2^M and the final rounding will be discussed in the next section.

6.5 Final rounding

For both quick and accurate phase, final rounding is simple when the result cannot be underflowed. In this case, the final multiplication of $t_1 \cdot t_2 \cdot e^r$ by 2^M only affects the exponent of $t_1 \cdot t_2 \cdot e^r$. So the rounding test and the final rounding (in quick and accurate phase) can be done on $t_1 \cdot t_2 \cdot e^r$ before this value is multiplied by 2^M. This rounding is a standard `crlibm` rounding of respectively a double-double or a triple-double. Since M might be as great as $M = 1024$ (whilst $t_1 \cdot t_2 \cdot e^r < 1$ in this case, because overflow in the final result has been filtered out), 2^M may not be representable. Nevertheless it is possible not to representate 2^M explicitely in a variable and to replace the operation by the following sequence. We suppose that $polyTbl_h = \circ(t_1 \cdot t_2 \cdot e^r + \delta)$.

Listing 6.11: Final multiplication by 2^M

```
polyTblhdb.d = polyTblh;
polyTblhdb.i[HI] += M << 20;
return polyTblhdb.d;
```

If the result might be gradually underflowed but is not completely underflowed (i.e. not equal to 0 or 0^+ depending on the rounding mode), the quick phase is not used and the accurate phase is launched in any case. This does not sensibly affect performance. The triple-double result of the accurate phase,

`polyTblh`, `polyTblm` and `polyTbll`, where $polyTbl_h + polyTbl_m + polyTbl_l \approx t_1 \cdot t_2 \cdot e^r$, is multiplied by 2^M and rounded to double precision as follows:

- $polyTbl_h$ is multiplied by 2^M in two steps: 2^M is not representable in double precision but 2^{-1000} and 2^{M+1000} are. This multiplication generates a subnormal t_4, which prevents us from replacing it by an integer sequence manipulating the exponent of the numbers involved. We have $t_4 = \circ\left(2^M \cdot polyTbl_h\right) = 2^M polyTbl_h + \delta$ with $|\delta| \leq \frac{1}{2} \cdot \text{ulp}\,(denorm) = 2^{-1075}$.
- The obtained value t_4 is remultiplied by 2^{-M}, once again in 2 steps. This multiplication produces a normal number and is therefore exact: $t_6 = 2^{-M} \cdot \circ\left(2^M \cdot polyTbl_h\right) = polyTbl_h + 2^{-M} \cdot \delta$.
- Since $|M| \leq 1075$ and $|polyTbl_h| \leq 2$, one checks that

  $$\frac{1}{2} \cdot polyTbl_h \leq t_6 \leq 2 \cdot polyTbl_h$$

  is verified. So the arithmetical substraction $t_7 = polyTbl_h \ominus t_6$ is exact by Sterbenz' lemma and one obtains:

  $$t_7 = 2^{-M} \cdot \left(2^M \cdot polyTbl_h - \circ\left(2^M \cdot polyTbl_h\right)\right)$$

  We obtain therefore

  $$t_4 + 2^M \cdot (t_7 + polyTbl_m + polyTbl_l) = 2^M \cdot (polyTbl_h + polyTbl_m + polyTbl_l)$$

  Further, since the triple-double number $polyTbl_h + polyTbl_m + polyTbl_l$ is non-overlapping, we know that

  $$t_4 \in \left\{r^-, r, r^+\right\}$$

  with $r = \circ\,(polyTbl_h + polyTbl_m + polyTbl_l)$. In addition, since the precision of a denormal is maximally 52 bits and that of a normal (like $polyTbl_h$) is 53 bits, t_7 is either 0 or greater in magnitude than $polyTbl_m + polyTbl_l$. If this were not the case, the triple-double would overlap. Thus $2^M \cdot (t_7 + polyTbl_m + polyTbl_l)$ corresponds to the part of the high accuracy mantissa that is rounded off when rounding to the subnormal and if t_7 is not zero, it contains at least the first bit that is rounded off.

The arithmetical steps mentioned are performed by the following code sequence:

Listing 6.12: Underflowed final multiplication and rounding

```
/* Final rounding and multiplication with 2^M

   We first multiply the highest significant byte by 2^M in two steps
        and adjust it then depending on the lower significant parts.

        We cannot multiply directly by 2^M since M is less than -1022.
        We first multiply by 2^(-1000) and then by 2^(M+1000).

*/

t3 = polyTblh * twoPowerM1000;

/* Form now twoPowerM with adjusted M */
twoPowerMdb.i[LO] = 0;
twoPowerMdb.i[HI] = (M + 2023) << 20;


/* Multiply with the rest of M, the result will be denormalized */
t4 = t3 * twoPowerMdb.d;

/* For x86, force the compiler to pass through memory for having the right rounding */

t4db.d = t4;   /* Do not #if-ify this line, we need the copy */
#if defined(CRLIBM_TYPECPU_AMD64) || defined(CRLIBM_TYPECPU_X86)
t4db2.i[HI] = t4db.i[HI];
t4db2.i[LO] = t4db.i[LO];
t4 = t4db2.d;
#endif

/* Remultiply by 2^(-M) for manipulating the rounding error and the lower significant parts */
```

```
M *= -1;
twoPowerMdb.i[LO] = 0;
twoPowerMdb.i[HI] = (M + 23) << 20;
t5 = t4 * twoPowerMdb.d;
t6 = t5 * twoPower1000;
t7 = polyTblh - t6;
```

One remarks that on $x86$ platforms, it is necessary to write t_4 to memory and to reread it from there in order to to overcome the inexistence of pseudosubnormals in the underlying 80 bit register format.

Depending on the rounding mode, t_4 is then adjusted to the correct rounding of $2^M \cdot (polyTbl_h + polyTbl_m + polyTbl_l)$ as follows:

- In round-to-nearest mode, the rounding decision when rounding to a subnormal is made at a constant value $\frac{1}{2} \cdot \text{ulp}\,(denorm) = 2^{-1075}$.

  Rounding is simple for round-to-nearest when the mantissa does not contain a one following the last mantissa bit of the rounded result. In this case, the rounding downwards and thus equivalent to a truncation. Truncation to different lengths is associative. Omitting $polyTbl_m + polyTbl_l$ is a truncation. The rounding of $2^M \cdot polyTbl_h$ to a subnormal is correct as per the use of a IEEE 754 operation. So t_4 already is equal to $\circ\,(polyTbl_h + polyTbl_m + polyTbl_l)$.

  If the first bit following the last mantissa bit of the rounded result is a one, two cases must be considered: if the TMD's case is such that the next following mantissa one is contained in the bits of $polyTbl_h$ that are rounded of, the rounding $\circ\left(2^M \cdot polyTbl_h\right)$ is equivalent to the rounding $\circ\left(2^M \cdot (polyTbl_h + polyTbl_m + polyTbl_l)\right)$ because both roundings are upwards. So in this case, too, t_4 already contains the correct result.

  In contrast, if the the TMD's case is such that the next following mantissa one (after the first one following the rounded mantissa) is only contained in $polyTbl_m + polyTbl_l$, the rounding direction of $\circ\left(2^M \cdot polyTbl_h\right)$ and $\circ\left(2^M \cdot (polyTbl_h + polyTbl_m + polyTbl_l)\right)$ may be different. As t_7 is the (scaled) correction of the roundoff of $\circ\left(2^M \cdot polyTbl_h\right)$ and greater in magnitude than $polyTbl_m + polyTbl_l$, its sign determines the rounding direction of the rounding $t_4 = \circ\left(2^M \cdot polyTbl_h\right)$. The sign of $polyTbl_m = \circ\,(polyTbl_h + polyTbl_l)$ determines then that of the direction of $\circ\left(2^M \cdot (polyTbl_h + polyTbl_m + polyTbl_l)\right)$ and such whether t_4 must be adjusted by $\pm$1ulp.

  The algorithm is thus the following: $2^M \cdot t_7$ is compared to $\frac{1}{2} \cdot \text{ulp}\,(denorm)$ by comparing t_7 to $2^{-1075-M}$. This determines whether the rounding is easy and t_4 can be returned or whether an adjustment must be made. If the latter is the case, the signs of t_7 and $polyTbl_h$ are examined and t_4 is adjusted. Since $e^x > 0\ \forall x \in \mathbb{R}$, the adjustment of t_4 by $\pm$ulp can be simplified. The code sequence below implements this:

Listing 6.13: Rounding adjustment in round-to-nearest

```
/* The rounding decision is made at 1/2 ulp of a denormal, i.e. at 2^(-1075)
        We construct this number and by comparing with it we get to know
        whether we are in a difficult rounding case or not. If not we just return
        the known result. Otherwise we continue with further tests.
*/

twoPowerMdb.i[LO] = 0;
twoPowerMdb.i[HI] = (M - 52) << 20;

if (ABS(t7) != twoPowerMdb.d) return t4;

/* If we are here, we are in a difficult rounding case */

/* We have to adjust the result iff the sign of the error on
        rounding 2^M * polyTblh (which must be an ulp of a denormal)
        and polyTblm +arith polyTbll is the same which means that
        the error made was greater than an ulp of an denormal.
*/

polyTblm = polyTblm + polyTbll;

if (t7 > 0.0) {
  if (polyTblm > 0.0) {
        t4db.l++;
        return t4db.d;
  } else return t4;
```

```
} else {
  if (polyTblm < 0.0) {
        t4db.l--;
        return t4db.d;
  } else return t4;
}
```

- In round-upwards mode, the rounding $\triangle(polyTbl_h + polyTbl_m + polyTbl_l)$ is determined by $\circ(polyTbl_h)$ and the rounding rest $2^M \cdot (t_7 + polyTbl_m + polyTbl_l)$.

  If t_7 not equal to 0, it is greater in magnitude than $polyTbl_m + polyTbl_l$. Thus the rounding direction of the to-nearest rounding $\circ(polyTbl_h)$ is downwards if $t_7 + polyTbl_m + polyTbl_l$ is positive. In this case, an adjustment of +1ulp must be made on t_4.

  If t_7 is equal to 0, the rounding $\circ(polyTbl_h)$ was errorless and $t_4 = 2^M \cdot polyTbl_h$. So we get

$$\triangle\left(2^M \cdot (polyTbl_h + polyTbl_m + polyTbl_l)\right) = \triangle\left(t_4 + 2^M \cdot (polyTbl_m + polyTbl_l)\right)$$

  Clearly, $2^M \cdot (polyTbl_m + polyTbl_l)$ is less than ulp (t_4) because $polyTbl_h + polyTbl_m + polyTbl_l$ is non-overlapping. One can show that

$$\triangle(x + \mu) = \triangle(x + \nu)$$

  if $x \in \mathbb{F}$ and $\operatorname{sgn}(\mu) = \operatorname{sgn}(\nu)$, $|\mu|, |\nu| < \text{ulp}(x)$ and $\mu, \nu \in \mathbb{R}$ [25]. Since the algebraic images of double arguments have been filtered out, it is thus possible to find $\mu \in \mathbb{R}$ such that with $\nu = 2^M \cdot (polyTbl_m + polyTbl_l)$ respectively $\nu = \text{ulp}(t_4) + 2^M \cdot (polyTbl_m + polyTbl_l)$ the following can be verified:

$$\triangle\left(t_4 + 2^M \cdot (polyTbl_m + polyTbl_l)\right) = \begin{cases} t_4 & \text{if } polyTbl_m + polyTbl_l < 0 \\ t_4^+ & \text{otherwise} \end{cases}$$

  So it suffices to add t_7 and $polyTbl_m = \circ(polyTbl_m + polyTbll)$ together and to check for the sign of this sum. The rounding error of this operation may not affect the sign of its result. If the sign is positive, +1ulp is added to t_4. The following code sequence realizes this:

Listing 6.14: Rounding adjustment in round-upwards

```
/* The rounding can be decided using the sign of the arithmetical sum of the
       round-to-nearest-error (i.e. t7) and the lower part(s) of the final result.
       We add first the lower parts and add the result to the error in t7. We have to
       keep in mind that everything is scaled by 2^(-M).
       t8 can never be exactly 0 since we filter out the cases where the image of the
       function is algebraic and the implementation is exacter than the TMD worst case.
*/

polyTblm = polyTblm + polyTbll;
t8 = t7 + polyTblm;

/* Since we are rounding upwards, the round-to-nearest-rounding result in t4 is
       equal to the final result if the rounding error (i.e. the error plus the lower
           parts)
       is negative, i.e. if the rounding-to-nearest was upwards.
*/

if (t8 < 0.0) return t4;

/* If we are here, we must adjust the final result by +1ulp
       Relying on the fact that the exponential is always positive, we can simplify this
       adjustment
*/

t4db.l++;
return t4db.d;
```

- Round-downwards mode is analogous to round-upwards mode with signs inverted.
- Round-towards-zero is equivalent to rounding downwards.

6.6 Accuracy bounds

In this section we give an overview on the error bound computation. The actual error bound proof is implemented using the Gappa tool.

An implementation of the exponential function in double precision can be considered to be correctly rounding if the accuracy of the accurate phase is at least 113 bits for arguments x such that $|x| \geq 2^{-30}$ and at least 158 bits for $2^{-54} \leq |x| < 2^{-30}$ – provided that the rounding test after the quick phase is correct [11].

In order to give a first error estimate, one can consider

$$\begin{aligned}
\texttt{exp(x)} &= 2^M \cdot tbl_1 \cdot tbl_2 \cdot (1 + p(r_h)) \cdot (1 + r_m) \cdot (1 + r_l) \cdot (1 + \varepsilon_{\text{arith}}) \\
&= 2^M \cdot t_1 \cdot (1 + \varepsilon_{\text{tbl1}}) \cdot t_2 (1 + \varepsilon_{\text{tbl1}}) \cdot \\
&\quad \cdot (1 + (e^{r_h} - 1) \cdot (1 + \varepsilon_{\text{approxhigh}})) \cdot \\
&\quad \cdot (1 + (e^{r_m} - 1) \cdot (1 + \varepsilon_{\text{approxmiddle}})) \cdot \\
&\quad \cdot (1 + (e^{r_l} - 1) \cdot (1 + \varepsilon_{\text{approxlower}})) \cdot (1 + \varepsilon_{\text{arith}}) \\
&= 2^M \cdot t_1 \cdot t_2 \cdot e^{r_h + r_m + r_l} \cdot \\
&\quad \cdot (1 + \varepsilon_{\text{tbl1}}) \cdot (1 + \varepsilon_{\text{tbl2}}) \cdot \left(1 + \varepsilon'_{\text{approxhigh}}\right) \cdot \left(1 + \varepsilon'_{\text{approxmiddle}}\right) \cdot \left(1 + \varepsilon'_{\text{approxlower}}\right) \cdot (1 + \varepsilon_{\text{arith}}) \\
&= 2^M \cdot t_1 \cdot t_2 \cdot e^{\hat{r}} \cdot \\
&\quad \cdot (1 + \varepsilon_{\text{tbl1}}) \cdot (1 + \varepsilon_{\text{tbl2}}) \cdot \left(1 + \varepsilon'_{\text{approxhigh}}\right) \cdot \left(1 + \varepsilon'_{\text{approxmiddle}}\right) \cdot \left(1 + \varepsilon'_{\text{approxlower}}\right) \cdot (1 + \varepsilon_{\text{arith}}) \cdot (1 + \varepsilon_{\text{argred}}) \\
&= e^x \cdot (1 + \varepsilon)
\end{aligned}$$

where $\varepsilon'_{\text{approxhigh}} = \varepsilon_{\text{approxhigh}} - \frac{\varepsilon_{\text{approxhigh}}}{e^{r_h}}$, $\varepsilon'_{\text{approxmiddle}} = \varepsilon_{\text{approxmiddle}} - \frac{\varepsilon_{\text{approxmiddle}}}{e^{r_m}}$ and $\varepsilon'_{\text{approxlower}} = \varepsilon_{\text{approxlower}} - \frac{\varepsilon_{\text{approxlower}}}{e^{r_l}}$. The Gappa proof files integrate all these errors and allow for evaluating the relative arithmetical round-off error $\varepsilon_{\text{arith}}$.

6.7 Timings

For evaluating the timings of the triple-double based implementation of the exponential function e^x, in comparison with other correctly rounded functions. "`crlibm` portable using `scslib`" still stands for a logarithm implementation in `crlibm` before the work on triple-double. The values are given in arbitrary units and obtained on a IBM Power 5 processor with gcc 3.3.3 on a Linux Kernel 2.6.5. The timings on other systems are comparable.

Library	avg time	max time
`MPFR`	2128	4908
`crlibm` portable using `scslib`	44	1976
`crlibm` portable using triple-double	39	258
default `libm` (IBM's `libultim`)	34	221062

On average, our triple-double based implementation wins about 12% speed-up in comparison with the SCS based implementation. It is however slightly less performant than Ziv's library.

Concerning worst case timing, the results are more striking and confirm the general results concering triple-double: the use of triple-double arithmetic instead of the SCS format allows for a speed-up of a factor of about 7.65. In comparison with IBM's `libultim` a performance gain of a factor 857 is achieved. The timing difference between average and worst-case is decreased to a factor of about 6.6.

Chapter 7

The `expm1` function

Todo. In between, see files expm1-td.{h,c,mpl,gappa}.

Chapter 8

The `log1p` function

Todo. In between, see files log1p-td.{h,c,mpl,gappa}.

Chapter 9

The trigonometric functions

This chapter is contributed by F. de Dinechin with assistance of C. Daramy-Loirat and D. Defour.

Introduction

This chapter describes the implementations of sine, cosine and tangent, as they share much of their code. The proof sketch below is supported by the Maple script `maple/trigo.mpl` and the Gappa scripts `gappa/trigoSinCosCase3.gappa` and `gappa/trigoTanCase2.gappa` of the `crlibm` distribution. These scripts implement the computations of error bounds and validity bounds for the various algorithmic paths described here.

9.1 Overview of the algorithms

9.1.1 Exceptional cases

The three trigonometric functions return NaN for infinite and NaN arguments, and are defined otherwise.

An argument based on continued fractions to find the worst cases for range reduction may also be used to show that the sine and cosine of a floating-point number outside of $[-1, 1]$ is always larger than 2^{-150}, and therefore never flushes to zero nor to subnormal (see [34] p. 151 and following). Therefore $\tan(x) = \sin(x)/\cos(x)$ also remains larger than 2^{-150}.

This has two important consequences:

- as the output a trigonometric function is never a subnormal except for inputs around zero (for which the value to return is trivial anyway), we can safely use the rounding tests from Section 2.7 p. 39.
- as the cosine never flushes to zero, the tangent of a floating-point number is never an infinity, and does not even come close, so again we may safely use the rounding tests from Section 2.7.

For very small arguments,

- $\sin(x) = x - x^3/6 + O(x^5) = x(1 - x^2/6) + O(x^5)$ where $O(x^5)$ has the sign of x. Therefore $\sin(x)$ is rounded to x or one of its floating-point neighbours as soon as $|x| < 2^{-26}$.
- $\cos(x) = 1 - x^2/2 + O(x^4)$ where $O(x^4)$ is positive. Therefore $\cos(x)$ is rounded to 1 in RN and RU mode if $x < \sqrt{2^{-53}}$. In RD and RZ modes, we have $\cos(0) = 1$ and $\cos(x) = 1 - 2^{-53}$ for $|x| < 2^{-26}$.
- $\tan(x) = x + x^3/3 + O(x^5) = x(1 + x^2/3) + O(x^5)$ where $O(x^5)$ has the sign of x. Therefore $\tan(x)$ is rounded to x or one of its neighbours for all the rounding modes if $|x| < 2^{-27}$.

9.1.2 Range reduction

Most implementations of the trigonometric functions have two steps of range reduction:

- first the input number x is reduced to $y \in [-\frac{\pi}{4}, \frac{\pi}{4}]$, with reconstruction using periodicity and symmetry properties,
- then the reduced argument is further broken down as $y = a + z$, with reconstruction using the formula for $\sin(a+z)$ and $\cos(a+z)$, using tabulated values of $\sin(a)$ and $\cos(a)$

We chose to implement range reduction in one step only, which computes an integer k and a reduced argument y such that

$$x = k\frac{\pi}{256} + y \tag{9.1}$$

where k is an integer and $|y| \leq \pi/512$. This step computes y as a double-double: $y \approx y_h + y_l$.

In the following we note

$$a = k\pi/256.$$

Then we read off a table

$$sa_h + sa_l \approx sin(a)$$
$$ca_h + ca_l \approx cos(a)$$

Only 64 quadruples (sa_h, sa_l, ca_h, ca_l) are tabulated (amounting to $64 \times 8 \times 4 = 2048$ bytes), the rest is obtained by periodicity and symmetry, implemented as masks and integer operations on the integer k. For instance, $a \mod 2\pi$ is implemented by $k \mod 512$, $\pi/2 - a$ is implemented as $128 - k$, etc.

Then we use the reconstruction steps:

$$\sin(x) = \sin(a+y) = \cos(a)\sin(y) + \sin(a)\cos(y) \tag{9.2}$$

$$\cos(x) = \cos(a+y) = \cos(a)\cos(y) - \sin(a)\sin(y) \tag{9.3}$$

$$tan(x) = \frac{\sin(x)}{\cos(x)} \tag{9.4}$$

9.1.3 Polynomial evaluation

To implement the previous equations, $\cos(y)$ and $\sin(y)$ are computed as unevaluated $1 + t_c$ and $(y_h + y_l)(1 + t_s)$ respectively, where t_c and t_s are doubles computed using a polynomial approximation of small degree:

- $t_s = y^2(s_3 + y^2(s_5 + y^2 s_7)))$ with $s3$, $s5$ and $s7$ the Taylor coefficients.
- $t_c = y^2(c_2 + y^2(c_4 + y^2 c_6))$ with $c2$, $c4$ and $c6$ the Taylor coefficients (or a more accurate minimax approximation).

9.1.4 Reconstruction

Sine

According to equation (9.2), we have to compute:

$$\begin{aligned} \sin(a+y) &= \sin(a)\cos(y) + \cos(a)\sin(y) \\ &\approx (sa_h + sa_l)(1+t_c) + (ca_h + ca_l)(y_h + y_l)(1+t_s) \end{aligned}$$

Figure 9.1 shows the worst-case respective orders of magnitude of the terms of this sum. The terms completely to the right of the vertical bar will be neglected, and a bound on the error thus entailed is computed in the following. Note that the term $ca_h y_h$ has to be computed exactly by a Mul12.

Finally the reconstruction consists of adding together the lower-order terms in increasing order of magnitude, and computing the double-double result by an Add12.

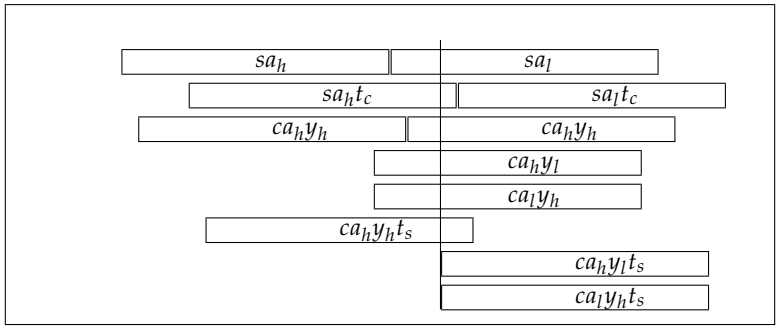


Figure 9.1: The sine reconstruction

Cosine

According to equation (9.3), we have to compute in double-double precision:

$$\begin{aligned}\cos(a+y) &= \cos(a)\cos(y) - \sin(a)\sin(y)\\ &\approx (ca_h + ca_l)(1+t_c) - (sa_h + sa_l)(y_h + y_l)(1+t_s)\end{aligned}$$

This is similar to the case of the sine, and the respective orders of magnitude are given by Figure 9.1.

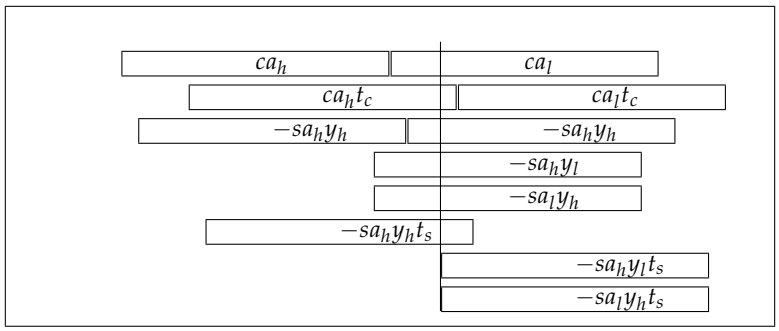


Figure 9.2: The cosine reconstruction

Tangent

The tangent is obtained by the division of the sine by the cosine, using the `Div22` procedure which is accurate to 2^{-104}.

9.1.5 Precision of this scheme

As we have $|y| < 2^{-7}$, this scheme computes these functions accurately to roughly 2^{53+13} bits, so these first steps are very accurate.

9.1.6 Organisation of the code

The code for range reduction is shared by the three trigonometric functions. It is detailed and proven in Section 9.2.

Then there are four procedures (currently implemented as macros for efficiency), respectively called `DoSinZero`, `DoCosZero`, `DoSinNotZero` and `DoCosNotZero` which do the actual computation after range reduction as per Figures 9.1 and 9.2. The tangent function is computed by dividing the sine by the cosine. These procedures are studied in section 9.3.

Finally, each of the three trigonometric functions comes in four variants for the four rounding modes. These four variants differ in the beginning (special cases) and in the end (rounding), but share the bulk of the computation. The shared computation is called `compute_trig_with_argred`.

9.2 Details of range reduction

9.2.1 Which accuracy do we need for range reduction?

The additive reduction consists in adding or removing N times a certain constant C from the input argument. For trigonometric function this constant is usually equal to $\pi/2$ or $\pi/4$, in our case it is $\pi/128$. A naive range reduction based on machine precision for trigonometric function would be implemented as :

$$\begin{array}{rcl} k & = & \lfloor \frac{x}{C} \rfloor \\ x^* & = & x - kC. \end{array} \tag{9.5}$$

Obviously, this subtraction cancels all the bits common to x and kC:

- The absolute accuracy of x^* with respect to the exact value of $x - kC$ depends only on the precision used to compute the subtraction and the product in $x - kC$ (x is considered exact as it is the input to the function).
- However, the relative accuracy of x^* with respect to the exact value of $x - kC$ also depends on this exact value. For a given precision of the operations used in computing $x - kC$, the closer x to kC, the smaller the exact value of $x - kC$, and the worse the relative accuracy (or, in less formal terms, the more bits of the results are lost to cancellation).

As the theorems for correct rounding of Section 2.7 p. 39 depend on the relative accuracy of the evaluation (and therefore on the relative accuracy of the reduced argument), we have to prove bounds on the relative accuracy of range reduction.

Formulated in simpler terms, how many bits can be lost in the subtraction $x - kC$? This number is related to the knowledge of the closest input number x to a multiple of C. There exists an algorithm due to Kahan/Douglas and based on continued fraction to compute this number and therefore the number of bit lost (see [34] p. 151 and following). We used a Maple version from [34] to determine than up to 62 bits may be cancelled during range reduction for a ieee double precision number. This Maple procedure is implemented as function `WorstCaseForAdditiveRangeReduction` in `maple/common-procedures.mpl`.

One advantage of having $C = \pi/128$, however, is that this is only a concern when x is close to a multiple of $\pi/2$ (that is, $k \mod 128 = 0$): in the other cases (i.e. in the general, most frequent case) the reconstruction will add some tabulated non-zero value, so the error to consider in the range reduction is the absolute error. Only in the cases when $k \mod 128 = 0$ do we need to have 62 extra bits to compute with. This is ensured by using a slower, more accurate range reduction. As a compensation, in this case when $k \mod 128 = 0$, there is no table to read and no reconstruction to perform: a simple polynomial approximation to the function suffices.

9.2.2 Details of the used scheme

We have 4 possible range reductions, depending on the magnitude of the input number (and the previous considerations):

- Cody and Waite with 2 constants (the fastest),
- Cody and Waite with 3 constants (almost as fast),
- Cody and Waite with 3 constants in double-double and k a 64-bit int, and
- Payne and Hanek, implemented in SCS (the slowest).

Each of these range reductions except Payne and Hanek is valid for x smaller than some bound. The computation of these bounds is detailed below.

Section 9.2.3 details the organization of this multi-level range reduction, and is followed by a detailed proof of each level.

9.2.3 Structure of the range reduction

The complete code is detailed below. To provide a complete picture to the reader it actually also includes the reconstruction.

Listing 9.1: Multilevel range reduction

```
struct rrinfo_s {double rh; double rl; double x; int absxhi; int function;} ;
typedef struct rrinfo_s rrinfo;
#define changesign function /* saves one int in the rrinfo structure */

static void ComputeTrigWithArgred(rrinfo *rri){
  double sah,sal,cah,cal, yh, yl, yh2, ts,tc, kd;
  double kch_h,kch_l, kcm_h,kcm_l, th, tl,sh,sl,ch,cl;
  int k, quadrant, index;
  long long int kl;

  if  (rri->absxhi < XMAX_CODY_WAITE_3) {
    /* Compute k, deduce the table index and the quadrant */
    DOUBLE2INT(k, rri->x * INV_PIO256);
    kd = (double) k;
    quadrant = (k>>7)&3;
    index=(k&127)<<2;
    if((index == 0)) {
      /* Here a large cancellation on yh+yl would be a problem, so use double-double RR */
      /* all this is exact */
      Mul12(&kch_h, &kch_l,   kd, RR_DD_MCH);
      Mul12(&kcm_h, &kcm_l,   kd, RR_DD_MCM);
      Add12 (th,tl,  kch_l, kcm_h) ;
      /* only rounding error in the last multiplication and addition */
      Add22 (&yh, &yl,    (rri->x + kch_h) , (kcm_l - kd*RR_DD_CL),   th, tl) ;
      goto computeZero;
    }
    else {
      /* index <> 0, don't worry about cancellations on yh+yl */
      if (rri->absxhi < XMAX_CODY_WAITE_2) {
        /* CW 2: all this is exact but the rightmost multiplication */
        Add12 (yh,yl,  (rri->x - kd*RR_CW2_CH), (kd*RR_CW2_MCL) ) ;
      }
      else {
        /* CW 3: all this is exact but the rightmost multiplication */
        Add12Cond(yh,yl,  (rri->x - kd*RR_CW3_CH) -  kd*RR_CW3_CM,   kd*RR_CW3_MCL);
      }
    }
    goto computeNotZero;
  }

  else if ( rri->absxhi < XMAX_DDRR ) {
    /* x sufficiently small for a Cody and Waite in double-double */
    DOUBLE2LONGINT(kl, rri->x*INV_PIO256);
    kd=(double)kl;
    quadrant = (kl>>7)&3;
    index=(kl&127)<<2;
    if(index == 0) {
      /* Here again a large cancellation on yh+yl would be a problem,
         so we do the accurate range reduction */
      RangeReductionSCS();   /*recomputes k, index, quadrant, and yh and yl*/
      /* Now it may happen that the new k differs by 1 of kl, so check that */
      if(index==0)   /* no surprise */
        goto computeZero;
      else
        goto computeNotZero;
    }
    else {   /*  index<>0 : double-double range reduction*/
      /* all this is exact */
      Mul12(&kch_h, &kch_l,   kd, RR_DD_MCH);
      Mul12(&kcm_h, &kcm_l,   kd, RR_DD_MCM);
      Add12 (th,tl,  kch_l, kcm_h) ;
      /* only rounding error in the last multiplication and addition */
      Add22 (&yh, &yl,    (rri->x + kch_h) , (kcm_l - kd*RR_DD_CL),   th, tl) ;
      goto computeNotZero;
    }
  } /* closes if ( absxhi < XMAX_DDRR ) */

  else {
    /* Worst case : x very large, sin(x) probably meaningless, we return
       correct rounding but do't mind taking time for it */
    RangeReductionSCS();
```

```
72      quadrant = (k>>7)&3;
73      if(index == 0)
74        goto computeZero;
75      else
76        goto computeNotZero;
77    }
78
79
80  computeZero:
81    switch(rri->function) {
82
83    case SIN:
84      if (quadrant&1)
85        DoCosZero(&rri->rh, &rri->rl);
86      else
87        DoSinZero(&rri->rh, &rri->rl);
88      rri->changesign=(quadrant==2)||(quadrant==3);
89      return;
90
91    case COS:
92      if (quadrant&1)
93        DoSinZero(&rri->rh, &rri->rl);
94      else
95        DoCosZero(&rri->rh, &rri->rl);
96      rri->changesign= (quadrant==1)||(quadrant==2);
97      return;
98
99    case TAN:
100     rri->changesign = quadrant&1;
101     if (quadrant&1) {
102       DoSinZero(&ch, &cl);
103       DoCosZero(&sh, &sl);
104     } else {
105       DoSinZero(&sh, &sl);
106       DoCosZero(&ch, &cl);
107     }
108     Div22(&rri->rh, &rri->rl, sh, sl, ch, cl);
109     return;
110   }
111
112 computeNotZero:
113   if(index<=(64<<2)) {
114     sah=sincosTable[index+0].d; /* sin(a), high part */
115     sal=sincosTable[index+1].d; /* sin(a), low part  */
116     cah=sincosTable[index+2].d; /* cos(a), high part */
117     cal=sincosTable[index+3].d; /* cos(a), low part  */
118   }else { /* cah <= sah */
119     index=(128<<2) - index;
120     cah=sincosTable[index+0].d; /* cos(a), high part */
121     cal=sincosTable[index+1].d; /* cos(a), low part  */
122     sah=sincosTable[index+2].d; /* sin(a), high part */
123     sal=sincosTable[index+3].d; /* sin(a), low part  */
124   }
125   yh2 = yh*yh ;
126   ts = yh2 * (s3.d + yh2*(s5.d + yh2*s7.d));
127   tc = yh2 * (c2.d + yh2*(c4.d + yh2*c6.d ));
128   switch(rri->function) {
129
130   case SIN:
131     if (quadrant&1)
132       DoCosNotZero(&rri->rh, &rri->rl);
133     else
134       DoSinNotZero(&rri->rh, &rri->rl);
135     rri->changesign=(quadrant==2)||(quadrant==3);
136     return;
137
138   case COS:
139     if (quadrant&1)
140       DoSinNotZero(&rri->rh, &rri->rl);
141     else
142       DoCosNotZero(&rri->rh, &rri->rl);
143     rri->changesign=(quadrant==1)||(quadrant==2);
144     return;
145
146   case TAN:
147     rri->changesign = quadrant&1;
148     if (quadrant&1) {
149       DoSinNotZero(&ch, &cl);
150       DoCosNotZero(&sh, &sl);
```

```
151     } else {
152       DoSinNotZero(&sh, &sl);
153       DoCosNotZero(&ch, &cl);
154     }
155     Div22(&rri->rh, &rri->rl, sh, sl, ch, cl);
156     return;
157   }
158 }
```

Here are some comments on the structure of this code (the details on actual range reduction come in the following sections).

- The DOUBLETOINT macro at line 13 is called only if $x <$ `XMAX_CODY_WAITE_3` (line 11). This constant is defined (see Listing 9.6 below) such that the conditions for this macro to work (see Section 2.2.3) are fullfilled.
- Similarly for the DOUBLETOLONGINT macro (see Section 2.2.4) at line 43, which is called for inputs smaller than `XMAX_DDR` defined in Listing 9.8 below.
- There is one subtlety at lines 51 and following. There we take the decision of computing a more accurate range reduction depending on the value of $index = k \mod 256$. However, in the case when $x \times \frac{256}{\pi}$ is very close to the middle between two integers, it may happen (very rarely) that the value of $k \mod 256$ computed by this second range reduction differs from the first by ± 1. In such cases, both values will provide different but equally valid reduced arguments, but we have to ensure that k and the reduced value match, hence the test line 52.

9.2.4 Cody and Waite range reduction with two constants

Here we split C into two floating-point constants C_h and C_l such that C_h holds 21 bits of the mantissa of C (the rest being zeroes), and $C_l = \circ(C - C_h)$. The following gives the Maple code that computes these constants, and then computes the bound on which this range reduction is valid.

Listing 9.2: Maple script for computing constants for Cody and Waite 2

```
bitsCh_0:=32:  # ensures at least 53+11 bits

# 1/2 <= C/2^(expC+1) <1
Ch:= round(evalf( C * 2^(bitsCh_0-expC-1))) / (2^(bitsCh_0-expC-1)):
# recompute bitsCh in case we are lucky (and we are for bitsCh_0=32)
bitsCh:=1+log2(op(2,ieeedouble(Ch)[3])) :  # this means the log of the denominator

Cl:=nearest(C - Ch):
# Cody and Waite range reduction will work for |k|<kmax_cw2
kmax_cw2:=2^(53-bitsCh):

# The constants to move to the .h file
RR_CW2_CH := Ch:
RR_CW2_MCL := -Cl:
XMAX_CODY_WAITE_2 := nearest(kmax_cw2*C):
```

The C code that performs the reduction in this case is the following:

Listing 9.3: Cody and Waite range reduction with two constants

```
31        Add12 (yh,yl,  (x - kd*RR_CW2_CH),  (kd*RR_CW2_MCL) ) ;
```

Here only the rightmost multiplication involving a rounding:

- The multiplication kd $\otimes$ RR_CW2_CH is exact because kd is a small integer and RR_CW2_CH has enough zeroes in the mantissa.
- The subtraction is exact thanks to Sterbenz Lemma.
- The `Add12` procedure is exact.

The following Maple code thus computes the maximum absolute error on the reduced argument (with respect to the ideal reduced argument) in this case.

Listing 9.4: Maple script for computing absolute error for Cody and Waite 2

```
delta_repr_C_cw2    := abs(C-Ch-Cl);
delta_round_cw2     := kmax_cw2 * 1/2 * ulp(Cl) ;
delta_cody_waite_2 := kmax_cw2 * delta_repr_C_cw2 + delta_round_cw2;
# This is the delta on y, the reduced argument
```

9.2.5 Cody and Waite range reduction with three constants

The C code that performs the reduction in this case is the following:

Listing 9.5: Cody and Waite range reduction with three constants

```
35        Add12Cond(yh,yl,  (x - kd*RR_CW3_CH) -  kd*RR_CW3_CM,    kd*RR_CW3_MCL);
```

Here again all the operations are exact except the rightmost multiplication.

The following Maple code computes the constants, the bound on x for this reduction to work, and the resulting absolute error of the reduced argument with respect to the ideal reduced argument.

Listing 9.6: Maple script for computing constants for Cody and Waite 3

```
bitsCh_0:=21:
Ch:= round(evalf(  C * 2^(bitsCh_0-expC-1))) / (2^(bitsCh_0-expC-1)):
# recompute bitsCh in case we are lucky
bitsCh:=1+log2(op(2,ieeedouble(Ch)[3])) :  # this means the log of the denominator

r := C-Ch:
Cmed := round(evalf(  r * 2^(2*bitsCh-expC-1))) / (2^(2*bitsCh-expC-1)):
bitsCmed:=1+log2(op(2,ieeedouble(Cmed)[3])) :

Cl:=nearest(C - Ch - Cmed):

kmax_cw3 := 2^31:# Otherwise we have integer overflow

# The constants to move to the .h file
RR_CW3_CH  := Ch;
RR_CW3_CM  := Cmed:
RR_CW3_MCL := -Cl:
XMAX_CODY_WAITE_3 := nearest(kmax_cw3*C):

# The error in this case (we need absolute error)
delta_repr_C_cw3    := abs(C - Ch - Cmed - Cl):
delta_round_cw3     := kmax_cw3 * 1/2 * ulp(Cl) :
delta_cody_waite_3 := kmax_cw3 * delta_repr_C_cw3 + delta_round_cw3:
# This is the delta on y, the reduced argument
```

9.2.6 Cody and Waite range reduction in double-double

The C code that performs the reduction in this case is the following:

Listing 9.7: Cody and Waite range reduction in double-double

```
20      /* all this is exact */
21      Mul12(&kch_h, &kch_l,   kd, RR_DD_MCH);
22      Mul12(&kcm_h, &kcm_l,   kd, RR_DD_MCM);
23      Add12 (th,tl,  kch_l, kcm_h) ;
24      /* only rounding error in the last multiplication and addition */
25      Add22 (&yh, &yl,    (x + kch_h) , (kcm_l - kd*RR_DD_CL),   th, tl) ;
```

The error and the bound are computed by the following Maple code.

Listing 9.8: Maple script for computing constants for Cody and Waite double-double

```
# This max int value can be produced by DOUBLE2LONGINT
kmax:=2^46-1:
XMAX_DDRR:=nearest(kmax*C);

#in this case we have C stored as 3 doubles
Ch   := nearest(C):
Cmed := nearest(C-Ch):
Cl   := nearest(C-Ch-Cmed):

RR_DD_MCH := -Ch:
```

```
RR_DD_MCM := -Cmed:
RR_DD_CL  := Cl:

delta_repr_C := abs(C - Ch - Cmed - Cl):

# and we have only exact Add12 and Mul12  operations. The only place
# with possible rounding errors is:
#       Add22 (pyh, pyl,    (x + kch_h) , (kcm_l - kd*RR_DD_CL),   th, tl) ;
# where (x + kch_h) is exact (Sterbenz) with up to kmax bits of cancellation
# and the error is simply the error in  (kcm_l - kd*RR_DD_CL)
# At the very worst :
delta_round :=
              kmax * 1/2 * ulp(Cl) # for   kd*RR_DD_CL
              + kmax*ulp(Cl)         # for the subtraction
              + 2^(-100) * Pi/512 :    # for the Add22
delta_RR_DD :=  kmax * delta_repr_C + delta_round:
```

The value of kmax defined here will be explained below in Section 9.2.9.

9.2.7 Payne and Hanek range reduction

This range reduction is very classical (see K.C. Ng's paper[35] or Muller's book [34]) and the code both too long and too simple to appear here. The Payne and Hanek reduction uses SCS computations which ensure relative accuracy of 2^{-200}. The result is then converted to a double-double. Even counting a worst-case cancellation of less than 70 bits, the final absolute error is much smaller than for the other reductions.

Listing 9.9: Payne and Hanek error

```
delta_PayneHanek := 2^(-100):
```

9.2.8 Maximum error of range reduction

We have two cases here.

Case when $k \mod 256 \neq 0$ In this case we will need the absolute error of range reduction to compute the total relative error of DoSinNotZero and DoCosNotZero: These procedures add tabulated values to the reduced argument. This error bound is computed by the following Maple code.

Listing 9.10: Maple script computing the absolute error bound of range reduction

```
delta_ArgRed := max(delta_cody_waite_2, delta_cody_waite_3,
                    delta_RR_DD, delta_PayneHanek):
```

We find that $\overline{\delta}_{\text{argred}} \approx 2^{-71}$

Case when $k \mod 256 = 0$ Here we directly need the relative error $\varepsilon_{\text{argred}}$ on range reduction, which will be used below in Section 9.3.1. Looking back at Listing 9.2.3, we see that in this case we compute range reduction either in double-double, or in SCS. The following Maple code computes $\overline{\varepsilon}_{\text{argred}}$.

Listing 9.11: Maple script computing the relative error bound of range reduction

```
# First, what is the worst case for cancellation ?

emax := ieeedouble(XMAX_DDRR)[2] +1 :
# above emax, we will use Payne and Hanek so we do not worry

(wcn, wce, wceps) := WorstCaseForAdditiveRangeReduction(2,53,-8, emax, C):
wcx := wcn * 2^wce:
wck := round(wcx/C):
wcy := wcx - wck*C:

log2(wcy);    # y > 2^(-67);

# In these cases we use the double-double range reduction, for |k|<kmax_cw3
# and the relative precision in the worst case is for wcy

delta_round := kmax_cw3 * 1/2 * ulp(Cl)       # for   kd*RR_DD_CL
```

```
            + kmax_cw3 * ulp(Cl) :          # for the subtraction
delta_RR_DD :=  kmax_cw3 * delta_repr_C + delta_round:
eps_ArgRed := (1+delta_RR_DD/wcy)*(1+2^(-100)) -1:
```

This script first computes the smallest possible reduced value thanks to the Kahan/Douglas algorithm. It then computes the absolute worst-case error of double-double reduction, divides it by the smallest possible value to get a relative error, and adds the relative error of the final Add22.

We find that $\bar{\varepsilon}_{\text{argred}} \approx 2^{-69.6}$.

9.2.9 Maximum value of the reduced argument

For simplicity, we want to define a common upper bound $y_{\max}$ on $|\hat{y}|$, $|\texttt{yh} + \texttt{yl}|$ and $|\texttt{yh}|$. This bound takes into account ε_{argred}, an ε_{53} for the case when it is a bound on y_h, and also an error due to the fact that we had a rounding error when computing $x \otimes \texttt{INV\_PIO256}$, so the reduced value may slightly exceed $\pi/512$. This rounding error is at most one half-ulp of $x \times 256/\pi$, but cancellation may then scale it up.

More precisely, the `DOUBLE2INT` macro always returns the nearest integer of its argument, so we have $\lceil x \rfloor = x + \varepsilon_{-1}$. However its argument is

$$x \otimes \texttt{INV\_PIO256} = x \times \texttt{INV\_PIO256}(1+\varepsilon_{-53}) = x \times 256/\pi(1+\varepsilon_{-53})(1+\varepsilon') = x \times 256/\pi(1+\varepsilon_k)$$

where $\varepsilon' = \texttt{INV\_PIO256} \times 256/\pi - 1$ can be computed exactly.

Therefore we have

$$k \;=\; \left\lceil x \times \frac{256}{\pi}(1+\varepsilon_k) \right\rfloor \;=\; x \times \frac{256}{\pi}(1+\varepsilon_k) - f \quad \text{where} \quad |f| \leq 1/2$$

And, assuming this computation was done exactly,

$$\hat{y} = x - k\frac{\pi}{256} = f\frac{\pi}{256} - x\varepsilon_k$$

In absolute value,

$$|\hat{y}| \leq \frac{\pi}{512} + |x|\bar{\varepsilon}_k$$

where $\bar{\varepsilon}_k < \bar{\varepsilon}_{-52}$.

Again to compute this value precisely we have to consider the various paths of the algorithm. The SCS range reduction is not concerned, since the Payne and Hanek algorithm used there does not compute k in this manner. For the other paths, the worst case is of course for the larger x, in the double-double argument reduction. There we define $x_{\max} = k_{\max} * C$, hence $|\hat{y}| \leq \frac{\pi}{512} + 2^{k_{\max}}\frac{\pi}{256}\bar{\varepsilon}_k$ where $\bar{\varepsilon}_k < 2^{-52}$.

This is the reason for the bound on x defined in 9.2.6: although this range reduction could work for values up to $k_{\max} = 2^{52} - 1$, larger values would increase the maximum value of the reduced argument, decreasing the accuracy of the polynomial evaluation. With a smaller value of $k_{\max}$, we still have $|\hat{y}|$ close to $\pi/512$.

Finally we compute the common maximum value of y_h, $y_h + y_l$ and $\widehat{y}$ by taking into account the rounding errors and the less-than-ulp difference between y_h, $y_h + y_l$ and $\widehat{y}$.

9.3 Actual computation of sine and cosine

A sine or a cosine will actually be computed by one of DoSinZero, DoSinNotZero, DoCosZero and DoCosNotZero (with a possible change of sine). Section 9.3.2 will show how we compute the maximum total error of DoSinNotZero and DoCosNotZero. As DoCosZero is a simpler, more accurate computation than DoCosNotZero, its worst-case error will be smaller than that of DoCosNotZero, and we do not need to compute it. We currently have only one rounding constant for the Zero and NotZero cases, because having separate constants would degrade performance.

However, DoSinZero is slightly different in that it doesn't add a constant value at the end of the computation, as do the three others. Its error computation has to consider more relative errors than absolute errors. We therefore need to compute its error separately, which is done in the following section.

This section 9.3.1 should also help understanding the method used to write the Gappa input file given in Section 9.3.2.

9.3.1 DoSinZero

Upon entering DoSinZero, we have in $y_h + y_l$ an approximation to the ideal reduced value $\hat{y} = x - k\frac{\pi}{256}$ with a relative accuracy $\varepsilon_{\text{argred}}$:

$$y_h + y_l = (x - k\frac{\pi}{256})(1 + \varepsilon_{\text{argred}}) = \hat{y}(1 + \varepsilon_{\text{argred}}) \tag{9.6}$$

with, depending on the quadrant, $\sin(\hat{y}) = \pm\sin(x)$ or $\sin(\hat{y}) = \pm\cos(x)$ and similarly for $\cos(\hat{y})$. This just means that $\hat{y}$ is the ideal, errorless reduced value.

In the following we will assume we are in the case $\sin(\hat{y}) = \sin(x)$, (the proof is identical in the other cases), therefore the relative error that we need to compute is

$$\varepsilon_{\text{sinkzero}} = \frac{(*\texttt{psh} + *\texttt{psl})}{\sin(\texttt{x})} - 1 = \frac{(*\texttt{psh} + *\texttt{psl})}{\sin(\hat{y})} - 1 \tag{9.7}$$

Listing 9.12: DoSinZero

```
1  yh2 = yh*yh;                                    \
2  ts = yh2 * (s3.d + yh2*(s5.d + yh2*s7.d));      \
3  Add12(*psh,*psl,   yh, yl+ts*yh);               \
```

One may remark that we almost have the same code as we have for computing the sine of a small argument (without range reduction) in Section 9.4.5. The difference is that we have as input a double-double `yh` + `yl`, which is itself an inexact term.

At Line 4, the error of neglecting y_l and the rounding error in the multiplication each amount to half an ulp: $\texttt{yh2} = \texttt{yh}^2(1 + \varepsilon_{-53})$, with $\texttt{yh} = (\texttt{yh} + \texttt{yl})(1 + \varepsilon_{-53}) = \hat{y}(1 + \varepsilon_{\text{argred}})(1 + \varepsilon_{-53})$

Therefore

$$\texttt{yh2} = \hat{y}^2(1 + \varepsilon_{\texttt{yh2}}) \tag{9.8}$$

with

$$\bar{\varepsilon}_{\texttt{yh2}} = (1 + \bar{\varepsilon}_{\text{argred}})^2(1 + \bar{\varepsilon}_{-53})^3 - 1 \tag{9.9}$$

Line 5 is a standard Horner evaluation. Its approximation error is defined by:

$$P_{\texttt{ts}}(\hat{y}) = \frac{\sin(\hat{y}) - \hat{y}}{\hat{y}}(1 + \varepsilon_{\text{approxts}})$$

This error is computed in Maple as in 9.4.5, only the interval changes:

$$\bar{\varepsilon}_{\text{approxts}} = \left\| \frac{xP_{\texttt{ts}}(x)}{\sin(x) - x} - 1 \right\|_\infty$$

We also compute $\bar{\varepsilon}_{\text{hornerts}}$, the bound on the relative error due to rounding in the Horner evaluation thanks to the `compute_horner_rounding_error` procedure. This time, this procedure takes into account the relative error carried by `yh2`, which is $\bar{\varepsilon}_{\texttt{yh2}}$ computed above. We thus get the total relative error on `ts`:

$$\texttt{ts} = P_{\texttt{ts}}(\hat{y})(1 + \varepsilon_{\text{hornerts}}) = \frac{\sin(\hat{y}) - \hat{y}}{\hat{y}}(1 + \varepsilon_{\text{approxts}})(1 + \varepsilon_{\text{hornerts}}) \tag{9.10}$$

The final `Add12` is exact. Therefore the overall relative error is:

$$\begin{aligned}\varepsilon_{\text{sinkzero}} &= \frac{((\mathtt{yh} \otimes \mathtt{ts}) \oplus \mathtt{yl}) + \mathtt{yh}}{\sin(\hat{y})} - 1 \\ &= \frac{(\mathtt{yh} \otimes \mathtt{ts} + \mathtt{yl})(1+\varepsilon_{-53}) + \mathtt{yh}}{\sin(\hat{y})} - 1 \\ &= \frac{\mathtt{yh} \otimes \mathtt{ts} + \mathtt{yl} + \mathtt{yh} \ + \ (\mathtt{yh} \otimes \mathtt{ts} + \mathtt{yl}).\varepsilon_{-53}}{\sin(\hat{y})} - 1\end{aligned}$$

Let us define for now

$$\delta_{\text{addsin}} = (\mathtt{yh} \otimes \mathtt{ts} + \mathtt{yl}).\varepsilon_{-53} \tag{9.11}$$

Then we have

$$\varepsilon_{\text{sinkzero}} = \frac{(\mathtt{yh}+\mathtt{yl})\mathtt{ts}(1+\varepsilon_{-53})^2 + \mathtt{yl} + \mathtt{yh} \ + \ \delta_{\text{addsin}}}{\sin(\hat{y})} - 1$$

Using (9.6) and (9.10) we get:

$$\varepsilon_{\text{sinkzero}} = \frac{\hat{y}(1+\varepsilon_{\text{argred}}) \times \frac{\sin(\hat{y})-\hat{y}}{\hat{y}}(1+\varepsilon_{\text{approxts}})(1+\varepsilon_{\text{hornerts}})(1+\varepsilon_{-53})^2 + \mathtt{yl} + \mathtt{yh} \ + \ \delta_{\text{addsin}}}{\sin(\hat{y})} - 1$$

To lighten notations, let us define

$$\varepsilon_{\text{sin1}} = (1+\varepsilon_{\text{approxts}})(1+\varepsilon_{\text{hornerts}})(1+\varepsilon_{-53})^2 \ - \ 1 \tag{9.12}$$

We get

$$\begin{aligned}\varepsilon_{\text{sinkzero}} &= \frac{(\sin(\hat{y})-\hat{y})(1+\varepsilon_{\text{sin1}}) + \hat{y}(1+\varepsilon_{\text{argred}}) \ + \ \delta_{\text{addsin}} - \sin(\hat{y})}{\sin(\hat{y})} \\ &= \frac{(\sin(\hat{y})-\hat{y}).\varepsilon_{\text{sin1}} + \hat{y}.\varepsilon_{\text{argred}} \ + \ \delta_{\text{addsin}}}{\sin(\hat{y})}\end{aligned}$$

Using the following bound:

$$|\delta_{\text{addsin}}| = |(\mathtt{yh} \otimes \mathtt{ts} + \mathtt{yl}).\varepsilon_{-53}| \quad < \quad 2^{-53} \times |y|^3/3 \tag{9.13}$$

we may compute the value of $\bar{\varepsilon}_{\text{sinkzero}}$ as an infinite norm under Maple. We get an error smaller than 2^{-67}.

9.3.2 DoSinNotZero and DoCosNotZero

The proof would here be much longer than the previous one, in the same spirit. It would therefore be much more error prone. We probably would be even less confident in such a proof than if it was generated automatically using experimental software. Therefore, let us do just that: We will use `Gappa` (see `http://lipforge.ens-lyon.fr/projects/gappa/`), another development of the Arenaire project to automate the proof of numerical properties, including an interface to the automatic theorem prover Coq. Gappa will assist us in computing bounds (in the form of intervals) for all the errors entailed by our code.

Here we need to compute the error bound for the following straight line of code.

Listing 9.13: DoSinNotZero

```
yh2 = yh*yh ;
ts = yh2 * (s3.d + yh2*(s5.d + yh2*s7.d));
tc = yh2 * (c2.d + yh2*(c4.d + yh2*c6.d ));
Mul12(&cahyh_h,&cahyh_l, cah, yh);
Add12(thi, tlo, sah,cahyh_h);
tlo = tc*sah+(ts*cahyh_h+(sal+(tlo+(cahyh_l+(cal*yh + cah*yl)))));
Add12(*psh,*psl, thi, tlo);
```

The additional information we have is

- The range reduction total absolute error, such that

$$y_h + y_l = (x - k\frac{\pi}{256})(1 + \varepsilon_{\text{argred}}) = \hat{y}(1 + \varepsilon_{\text{argred}}) \tag{9.14}$$

- The rounding error of the double-double tabulated values $sa_h + sa_l$ and $ca_h + ca_l$ ($\bar{\varepsilon}_{-104}$) and their ranges
- The approximation error of the polynomials used, computed in Maple

We use Gappa to compute the evaluation error, then the total error. Currently, we actually have to lauch Gappa 63 times, one time for each of the possibles values of $a = k\pi/256$ that appear in our tables. The reason is that, in the current version of Gappa, we are unable to express the identity $sin^2(a) + cos^2(a) = 1$ in a useful manner. Without this identity, the only information that we could give to Gappa is that both $sin(a)$ and $cos(a)$ belong to $[-1, 1]$, which leads to an overestimation of the error.

The Maple script that computes the table of (ca_h, ca_l, sa_h, sa_l), the polynomial coefficient, and the approximation errors, also outputs it in a form suitable for input to Gappa (for technical reasons they have to be substituted in the code using the Unix utility `sed`).

The input to Gappa is as follows:

Listing 9.14: Gappa input to compute the error of DoSineNotZero

```
# Usage: You need to set the constants cah, cal, sah, sal. Running the trigo.mpl Maple script
# should create 64 files in maple/TEMPTRIG, which can be tested independently as
#    sed -f    ../maple/TEMPTRIG/SinACosA_1.sed  trigoSinCosCase3.gappa | ~/gappa/src/gappa >
    /dev/null

# NOTATION CONVENTION
# Variables that correspond to double-precision variables in the code begin with a small
    letter
# Other variables begin with a capital letter.
# Variables that will be replaced with Maple-computed constants begin with an underscore
# Otherwise avoid underscores as they are painful to carry on to LaTeX :)


# Rounding operators and sequences definition

@IEEEdouble = float<ieee_64,ne>;


# polynomial coefficients, computed by Maple
s3 = IEEEdouble(_s3);
s5 = IEEEdouble(_s5);
s7 = IEEEdouble(_s7);
c2 = IEEEdouble(_c2);
c4 = IEEEdouble(_c4);
c6 = IEEEdouble(_c6);

# Table values, computed by Maple
cah = IEEEdouble(_cah);
cal = IEEEdouble(_cal);
sah = IEEEdouble(_sah);
sal = IEEEdouble(_sal);

# The variables used here:
# x input
# My perfect reduced argument
# Yhl = yh+yl, his distance to My is specified as an hypothesis
# Mts perfect ts
# Mtc perfect tc
# Msinx exact result for sin(x)
# Msina, Mcosa perfect sin(kPi/256) and cos(kPi/256)


yh = IEEEdouble(Yhl);
yl = Yhl - yh;

######################################################################

# First, a transcription of the actual computation, which could (and
# should eventually) be generated automatically from the actual code
```

```
# ------------------------Code shared by sin and cos, cut from ComputeTrigWithArgRed :
yh2 IEEEdouble=  yh * yh;
ts  IEEEdouble=  yh2 * (s3 + yh2*(s5 + yh2*s7));
tc  IEEEdouble=  yh2 * (c2 + yh2*(c4 + yh2*c6));

# ------------------------Code for the sine, cut from DosinNotZero:
# Mul12(&cahyh_h,&cahyh_l, cah, yh);
cahyh = cah * yh;
cahyh_h = IEEEdouble(cahyh);
cahyh_l = cahyh - cahyh_h;    # Exact equation because Mul12 is exact

# Add12(thi, tlo, sah, cahyh_h);
TSin = sah + cahyh_h;
thiSin = IEEEdouble(TSin);
tloSin1 = TSin - thiSin;          # Exact equation because Add12 is exact

# Rem: need to Rename tlo to tloSin1, and its second use to tloSin2.
# It would be safer to translate code to single-assignment before
# using Gappa, modern compilers won't make any difference.

#  tlo = tc*sah+(ts*cahyh_h+(sal+(tlo+(cahyh_l+(cal*yh + cah*yl))))) ;
tloSin2 IEEEdouble= tc*sah + (ts*cahyh_h + (sal + (tloSin1 + (cahyh_l + (cal*yh + cah*yl))))));

# Add12(*reshi, *reslo, thi, tlo);
ResSinhilo = thiSin + tloSin2; # we don't need to split it for the proof.


# ------------------------Code for the cos, cut from DoCosNotZero:
#  Mul12(&sahyh_h,&sahyh_l, sah, yh);
sahyh = sah * yh;
sahyh_h = IEEEdouble(sahyh);
sahyh_l = sahyh - sahyh_h;    # Exact equation because Mul12 is exact

#  Add12(thi, tlo,  cah, -sahyh_h);
TCos = cah - sahyh_h;
thiCos = IEEEdouble(TCos);
tloCos1 = TCos - thiCos; # Exact equation because Add12 is exact

#  tlo = tc*sah+(ts*cahyh_h+(sal+(tlo+(cahyh_l+(cal*yh + cah*yl))))) ;
tloCos2 IEEEdouble= tc*cah-(ts*sahyh_h-(cal+(tloCos1-(sahyh_l+(sal*yh+sah*yl))))) ;

#  Add12(*pch, *pcl,    thi, tlo);
ResCoshilo = thiCos + tloCos2;   # No need to split it for the proof.


#########################################################################

#Definitions of the mathematical objects

#-----------------------------------------
#With these notations, the exact sine and cosine are given by these
#exact mathematical formulae

Msinx =  Msiny * Mcosa  +  Mcosy * Msina;
Mcosx =  Mcosy * Mcosa  -  Msiny * Msina;

# Now let us pile up layers of approximations
#-----------------------------------------
# yh2 is an approximation to :
My2 = My*My;
# through three layers:
# 1/    Yhl=yh+hl = My +/- delta_ArgRed  : in the hypotheses below
# 2/    yh = Yhl - yl                    : already written
# 3/    rounding error in the mult       : already written

#-----------------------------------------
# ts is an approximation to :
Mts = My2 * (s3 + My2*(s5 + My2*s7));
# through two layers:
# 1/ the approximation y2 of My2       : done just above
# 2/ the rounding errors in Horner     : already written

#-----------------------------------------
PolySinY = My * (1 + Mts);
# PolySinY is an approximation to sin(My) as expressed in the hypotheses below :
#  PolySinY - Msiny in [-0.24126e-23, 0.24126e-23]     # delta_approx_Sin_Case3

#-----------------------------------------
```

```
# Same for PolyCosY
Mtc = My2 * (c2 +(My2 * (c4 + (My2 * c6))));
PolyCosY = 1 + Mtc;

#-----------------------------------------
#tc is an approximation to Mtc through the rounding errors, defined
#in the definition of tc. Same for ts

#-----------------------------------------
# SinReconstrExact is an approximation to ResSinhilo
SinReconstrExact = PolySinY * Mcosa  +  PolyCosY * Msina ;

# The delta between ResSinhilo and SinReconstrExact is due to the two
# mathematical poly approx, and has been defined just above
CosReconstrExact = PolyCosY * Mcosa  -  PolySinY * Msina ;

#-----------------------------------------
# The reconstruction approximates the following
SinReconstrNoRound = Yhl*(1 + ts)*(cah+cal) + (1 + tc)*(sah+sal);
# where Yhl is an approximation to Y
#       ts is an approximation to Mts
#       tc is an approximation to Mtc : all already described
# All what we still need to express is that the actual computation will neglect some terms
CosReconstrNoRound = ( (1 + tc) * (cah+cal)   -  Yhl * (1 + ts)  * (sah+sal) );

#-----------------------------------------
# tloSin2 is an approximation to TloSin2NoRound (because of rounding error in the operations,
    already described)
TloSin2NoRound = tc*sah+(ts*cahyh_h+(sal+(tloSin1+(cahyh_l+(cal*yh+cah*yl)))));
TloCos2NoRound = tc*cah-(ts*sahyh_h-(cal+(tloCos1-(sahyh_l+(sal*yh+sah*yl)))));

#  tloSinNoRound is an approximation to SinReconstrNoRound - tSinhi, the
#  difference being the neglected terms. This error will be given as an hint

NeglectedSinTerms = SinReconstrNoRound - (thiSin + TloSin2NoRound);
NeglectedCosTerms = CosReconstrNoRound - (thiCos + TloCos2NoRound);

# And finally, ResSinhilo is an approximation to Msinx through many layers which are given in
    the hints.

##############################################################################################
#  The theorem to prove
{
#    (Yhl in [-_ymaxCase3, -1b-200]    \/   Yhl in [-1b-200, _ymaxCase3])
                              # computed by Maple
     Yhl in [-_ymaxCase3, _ymaxCase3]                                  # computed by Maple
 /\ Yhl - My in          [-_delta_ArgRed,  _delta_ArgRed]                                  # computed by
      Maple
 /\ PolySinY - Msiny in [-_delta_approx_Sin_Case3,  _delta_approx_Sin_Case3]     # computed by
      Maple
 /\ PolyCosY - Mcosy in [-_delta_approx_Cos_Case3,  _delta_approx_Cos_Case3]     # computed by
      Maple
 /\ Msina-sah-sal in [-1b-104, 1b-104]
 /\ Mcosa-cah-cal in [-1b-104, 1b-104] # double-double absolute rounding error, with margin

 ->
(ResSinhilo - Msinx)/Msinx in     [-3b-66,3b-66]
/\
(ResCoshilo - Mcosx)/Mcosx in    [-3b-66,3b-66]
}


##############################################################################################
# Hints to the reduction engine: Gappa is not supposed to be clever, it's an assistant

# To get bounds on Msinx, try ResSinhilo
Msinx -> ResSinhilo - (ResSinhilo - Msinx);
Mcosx -> ResCoshilo - (ResCoshilo - Mcosx);

# To get bounds on Msina, try sah+sal
Msina -> sah + sal + (Msina - sah - sal);
Mcosa -> cah + cal + (Mcosa - cah - cal);

# To get bounds on My, try Yhl
 My ->  Yhl - (Yhl - My);

# One layer of approx error, and one layer of rounding error for Msiny and Mcosy
1 + tc - Mcosy   ->   (1 + tc - PolyCosY) + (PolyCosY - Mcosy);
(My + My * ts) - Msiny ->   ((My + My * ts) - PolySinY) + (PolySinY - Msiny);
```

```
# Layers of approximations
ResSinhilo - Msinx -> (ResSinhilo - SinReconstrNoRound) + (SinReconstrNoRound -
    SinReconstrExact) + (SinReconstrExact - Msinx);
ResCoshilo - Mcosx -> (ResCoshilo - CosReconstrNoRound) + (CosReconstrNoRound -
    CosReconstrExact) + (CosReconstrExact - Mcosx);

NeglectedSinTerms -> sal*tc + cal*yl + ts*(cahyh_l + cah*yl + cal*yh + cal*yl);
ResSinhilo - SinReconstrNoRound -> (ResSinhilo - (thiSin + TloSin2NoRound)) - (
    SinReconstrNoRound - (thiSin + TloSin2NoRound)) ;

NeglectedCosTerms -> cal*tc - sal*yl - ts*(sahyh_l + sah*yl + sal*yh + sal*yl);
ResCoshilo - CosReconstrNoRound -> (ResCoshilo - (thiCos + TloCos2NoRound)) - (
    CosReconstrNoRound - (thiCos + TloCos2NoRound)) ;



Yhl*(1 + ts)*(cah+cal) - PolySinY * Mcosa -> (Yhl*(1 + ts)- PolySinY)*(cah+cal) - PolySinY*(
    Mcosa - cah - cal) ;
(1 + tc) * (cah+cal) - PolyCosY * Mcosa -> ((1 + tc)- PolyCosY) * (cah+cal) - PolyCosY *
    (Mcosa - cah -cal);
(1 + tc)*(sah+sal) - PolyCosY * Msina -> ((1 + tc)- PolyCosY) * (sah+sal) - PolyCosY *
    (Msina - sah -sal);
```

The max of the bounds computed by Gappa for all the values of this table is a little bit smaller than 2^{-66}. Therefore, we take $\overline{\varepsilon}_{SinCosCase3} = 2^{-66}$ for the trigonometric functions (because the reconstruction simply manipulates the signs and is therefore exact).

In the very near future, this bound should be machine-checked by the Coq proof assistant.

9.4 Detailed examination of the sine

The sine begins with casting the high part of the absolute value of the input number into a 32-bit integer, to enable faster comparisons. However we have to be aware that we lost the lower part of `x`, which had a value up to $(2^{31}-1)\text{ulp}(x)$. This is taken care of in the procedure that converts a Maple high-precision number into an integer to which `x` will be compared (procedure `outputHighPart` in `trigo.mpl`).

Listing 9.15: Casting to an int for faster comparisons

```
1  db_number x_split;
2  x_split.d=x;
3  absxhi = x_split.i[HI] & 0x7fffffff;
```

9.4.1 Exceptional cases in RN mode

Listing 9.16: Exceptional cases for sine RN

```
1  /* SPECIAL CASES: x=(Nan, Inf) sin(x)=Nan */
2  if (absxhi>=0x7ff00000) return x-x;
3
4  else if (absxhi < XMAX_SIN_CASE2){
5    /* CASE 1 : x small enough sin(x)=x */
6    if (absxhi <XMAX_RETURN_X_FOR_SIN)
7      return x;
```

9.4.2 Exceptional cases in RU mode

Listing 9.17: Exceptional cases for sine RU

```
1  /* SPECIAL CASES: x=(Nan, Inf) sin(x)=Nan */
2  if (absxhi>=0x7ff00000) return x-x;
3
4  if (absxhi < XMAX_SIN_CASE2){
5
6    /* CASE 1 : x small enough, return x suitably rounded */
7    if (absxhi <XMAX_RETURN_X_FOR_SIN) {
8      if(x>=0.)
9        return x;
```

```
10        else {
11          x_split.l --;
12          return x_split.d;
13        }
14      }
```

9.4.3 Exceptional cases in RD mode

Listing 9.18: Exceptional cases for sine RD

```
1   /* SPECIAL CASES: x=(Nan, Inf) sin(x)=Nan */
2   if (absxhi>=0x7ff00000) return x-x;
3
4   if (absxhi < XMAX_SIN_CASE2){
5
6     /* CASE 1 : x small enough, return x suitably rounded */
7     if (absxhi <XMAX_RETURN_X_FOR_SIN) {
8       if(x<=0.)
9         return x;
10      else {
11        x_split.l --;
12        return x_split.d;
13      }
14    }
```

9.4.4 Exceptional cases in RZ mode

Listing 9.19: Exceptional cases for sine RZ

```
1   /* SPECIAL CASES: x=(Nan, Inf) sin(x)=Nan */
2   if (absxhi>=0x7ff00000) return x-x;
3
4   if (absxhi < XMAX_SIN_CASE2){
5
6     /* CASE 1 : x small enough, return x suitably rounded */
7     if (absxhi <XMAX_RETURN_X_FOR_SIN) {
8       x_split.l --;
9       return x_split.d;
10    }
```

9.4.5 Fast approximation of sine for small arguments

Listing 9.20: Sine, case 2

```
1     xx = x*x;
2     ts = xx * (s3.d + xx*(s5.d + xx*s7.d ));
3     Add12(sh,sl, x, x*ts);
```

Here we have had no range reduction, therefore `x` is exact. We need to compute the relative error of `sh` + `sl` with respect to sin(`x`). As `sh` + `sl` is the result of an (exact) `Add12`, the error is:

$$\varepsilon_{\text{sinCase2}} = \frac{\mathtt{x} \otimes \mathtt{ts} + \mathtt{x}}{\sin(\mathtt{x})} - 1 = \frac{\mathtt{x} \times \mathtt{ts}(1+\varepsilon_{-53}) + \mathtt{x}}{\sin(\mathtt{x})} - 1 \tag{9.15}$$

The polynomial used to compute `ts` approximates $\frac{\sin(x)-x}{x}$:

$$P_{\mathtt{ts}}(x) = \mathtt{s3}.x^2 + \mathtt{s5}.x^4 + \mathtt{s7}.x^6 = \frac{\sin(x)-x}{x}(1+\varepsilon_{\text{approxts}})$$

We compute a bound on this error in Maple as

$$\overline{\varepsilon}_{\text{approxts}} = \left\| \frac{xP_{\mathtt{ts}}(x)}{\sin(x)-x} - 1 \right\|_{\infty} [-x_{\max}...x_{\max}]$$

We also compute $\varepsilon_{\text{hornerts}}$, the relative error due to rounding in the Horner evaluation thanks to the `compute_horner_rounding_error` procedure. For this we need the relative error carried by `xx`, which is only due to the rounding error in the multiplication since `x` is exact:

$$\mathtt{xx} = \mathtt{x}^2(1+\varepsilon_{-53})$$

We therefore have:

$$\mathtt{ts} = P_{\mathtt{ts}}(\mathtt{x})(1+\varepsilon_{\text{hornerts}}) = \frac{\sin(\mathtt{x})-\mathtt{x}}{\mathtt{x}}(1+\varepsilon_{\text{approxts}})(1+\varepsilon_{\text{hornerts}})$$

Reporting this in (9.15), we get

$$\varepsilon_{\text{sinCase2}} = \frac{(\sin(\mathtt{x})-\mathtt{x})(1+\varepsilon_{\text{approxts}})(1+\varepsilon_{\text{hornerts}})(1+\varepsilon_{-53})+\mathtt{x}}{\sin(\mathtt{x})} - 1$$

or,

$$\varepsilon_{\text{sinCase2}} = \frac{\sin(\mathtt{x})-\mathtt{x}}{\sin(\mathtt{x})}\left((1+\varepsilon_{\text{approxts}})(1+\varepsilon_{\text{hornerts}})(1+\varepsilon_{-53}) \;-\; 1\right)$$

Finally

$$\overline{\varepsilon}_{\text{sinCase2}} = \left\|\frac{\sin(\mathtt{x})-\mathtt{x}}{\sin(\mathtt{x})}\right\|_{\infty}((1+\overline{\varepsilon}_{\text{approxts}})(1+\overline{\varepsilon}_{\text{hornerts}})(1+\overline{\varepsilon}_{-53}) \;-\; 1) \tag{9.16}$$

9.5 Detailed examination of the cosine

The bulk of the computation is shared with the sine. The main differences are therefore in handling special values. We juste show the code here. The error computation for small arguments is similar to that of the sine, and is implemented in `maple/trigo.mpl`.

9.5.1 Round to nearest mode

Listing 9.21: Exceptional cases for cosine RN

```
1  double cos_rn(double x){
2    double tc, x2;
3    rrinfo rri;
4    db_number x_split;
5
6    x_split.d=x;
7    rri.absxhi = x_split.i[HI] & 0x7fffffff;
8
9    /* SPECIAL CASES: x=(Nan, Inf) cos(x)=Nan */
10   if (rri.absxhi>=0x7ff00000) {
11     /* was : return x-x;
12        but it's optimized out by Intel compiler (bug reported).
13        Who cares to be slow in this case anyway... */
14     x_split.l=0xfff8000000000000LL;
15     return x_split.d-x_split.d;
16   }
17
18   if (rri.absxhi < XMAX_COS_CASE2){
19     /* CASE 1 : x small enough cos(x)=1. */
20     if (rri.absxhi <XMAX_RETURN_1_FOR_COS_RN)
21       return 1.;
22     else {
23       /* CASE 2 : Fast polynomial evaluation */
24       x2 = x*x;
25       tc = x2 * (c2.d + x2*(c4.d + x2*c6.d ));
26       Add12(rri.rh,rri.rl, 1.0, tc);
27       if(rri.rh == (rri.rh + (rri.rl * RN_CST_COS_CASE2)))
28         return rri.rh;
29       else
30         return scs_cos_rn(x);
31     }
32   }
33   else {
34   /* CASE 3 : Need range reduction */
```

```
35      rri.x=x;
36      rri.function=COS;
37      ComputeTrigWithArgred(&rri);
38      if(rri.rh == (rri.rh + (rri.rl * RN_CST_COS_CASE3)))
39        if(rri.changesign) return -rri.rh; else return rri.rh;
40      else
41        return scs_cos_rn(x);
42    }
43  }
```

9.5.2 RU mode

Listing 9.22: Exceptional cases for cosine RU

```
1  double cos_ru(double x){
2    double x2, tc, epsilon;
3    rrinfo rri;
4    db_number x_split;
5
6    x_split.d=x;
7    rri.absxhi = x_split.i[HI] & 0x7fffffff;
8
9    /* SPECIAL CASES: x=(Nan, Inf) cos(x)=Nan */
10   if (rri.absxhi>=0x7ff00000) {
11     x_split.l=0xfff8000000000000LL;
12     return x_split.d - x_split.d;
13   }
14
15   if (rri.absxhi < XMAX_COS_CASE2){
16     /* CASE 1 : x small enough cos(x)=1. */
17     if (rri.absxhi <XMAX_RETURN_1_FOR_COS_RDIR)
18       return 1.;
19     else{
20       /* CASE 2 : Fast polynomial evaluation */
21       x2 = x*x;
22       tc = x2 * (c2.d + x2*(c4.d + x2*c6.d ));
23       Add12(rri.rh,rri.rl, 1, tc);
24       epsilon=EPS_COS_CASE2;
25     }
26   }
27
28   else {
29     /* CASE 3 : Need range reduction */
30     rri.x=x;
31     rri.function=COS;
32     ComputeTrigWithArgred(&rri);
33     epsilon=EPS_COS_CASE3;
34     if(rri.changesign) {
35       rri.rh = -rri.rh;
36       rri.rl = -rri.rl;
37     }
38   }
39
40
41   TEST_AND_RETURN_RU(rri.rh, rri.rl, epsilon);
42
43   /* if the previous block didn't return a value, launch accurate phase */
44   return scs_cos_ru(x);
45 }
```

9.5.3 RD mode

Listing 9.23: Exceptional cases for cosine RD

```
1  double cos_rd(double x){
2    double x2, tc, epsilon;
3    rrinfo rri;
4    db_number x_split;
5
6    x_split.d=x;
7    rri.absxhi = x_split.i[HI] & 0x7fffffff;
8
```

```
  /* SPECIAL CASES: x=(Nan, Inf) cos(x)=Nan */
  if (rri.absxhi>=0x7ff00000) {
    x_split.l=0xfff8000000000000LL;
    return x_split.d - x_split.d;
  }

  if (rri.absxhi < XMAX_COS_CASE2){
    if (x==0) return 1;
    /* CASE 1 : x small enough cos(x)=1. */
    if (rri.absxhi <XMAX_RETURN_1_FOR_COS_RDIR)
      return ONE_ROUNDED_DOWN;
    else {
      /* CASE 2 :  Fast polynomial evaluation */
      x2 = x*x;
      tc = x2 * (c2.d + x2*(c4.d + x2*c6.d ));
      Add12(rri.rh,rri.rl, 1, tc);
      epsilon=EPS_COS_CASE2;
    }
  }
  else {
  /* CASE 3 : Need range reduction */
    rri.x=x;
    rri.function=COS;
    ComputeTrigWithArgred(&rri);
    epsilon=EPS_COS_CASE3;
    if(rri.changesign) {
      rri.rh = -rri.rh;
      rri.rl = -rri.rl;
    }
  }

  TEST_AND_RETURN_RD(rri.rh, rri.rl, epsilon);

  /* if the previous block didn't return a value, launch accurate phase */
  return scs_cos_rd(x);
}
```

9.5.4 RZ mode

Listing 9.24: Exceptional cases for cosine RZ

```
double cos_rz(double x){
  double x2, tc, epsilon;
  rrinfo rri;
  db_number x_split;

  x_split.d=x;
  rri.absxhi = x_split.i[HI] & 0x7fffffff;

  /* SPECIAL CASES: x=(Nan, Inf) cos(x)=Nan */
  if (rri.absxhi>=0x7ff00000) {
    x_split.l=0xfff8000000000000LL;
    return x_split.d - x_split.d;
  }

  if (rri.absxhi < XMAX_COS_CASE2){
    if (x==0) return 1;
    /* CASE 1 : x small enough cos(x)=1. */
    if (rri.absxhi <XMAX_RETURN_1_FOR_COS_RDIR)
      return ONE_ROUNDED_DOWN;
    else {
      /* CASE 2 : Fast polynomial evaluation */
      x2 = x*x;
      tc = x2 * (c2.d + x2*(c4.d + x2*c6.d ));
      Add12(rri.rh,rri.rl, 1, tc);
      epsilon=EPS_COS_CASE2;
    }
  }
  else {
    /* CASE 3 : Need range reduction */
    rri.x=x;
    rri.function=COS;
    ComputeTrigWithArgred(&rri);
    epsilon=EPS_COS_CASE3;
    if(rri.changesign) {
      rri.rh = -rri.rh;
```

```
36        rri.rl = -rri.rl;
37      }
38    }
39
40    TEST_AND_RETURN_RZ(rri.rh, rri.rl, epsilon);
41
42    /* if the previous block didn't return a value, launch accurate phase */
43    return scs_cos_rz(x);
44  }
```

9.6 Detailed examination of the tangent

9.6.1 Total relative error

In the general case, we compute the tangent by using successively the macros for computing the sine and the cosine (each accurate to $\overline{\varepsilon}_{SinCosCase3}$), then dividing sine by cosine using the Div22 macro, accurate to $2^{-100} << \overline{\varepsilon}_{SinCosCase3}$ (see lines 146-156 of listing 9.1). The overall error is thus bounded by

$$\overline{\varepsilon}_{tan} = 2.1\overline{\varepsilon}_{SinCosCase3}.$$

9.6.2 RN mode

Listing 9.25: Exceptional cases for tangent RN

```
1  double tan_rn(double x){
2    double x2, p5, tt;
3    rrinfo rri;
4    db_number x_split, rndcst;
5
6    x_split.d=x;
7    rri.absxhi = x_split.i[HI] & 0x7fffffff;
8
9    /* SPECIAL CASES: x=(Nan, Inf) cos(x)=Nan */
10   if (rri.absxhi>=0x7ff00000) {
11     x_split.l=0xfff8000000000000LL;
12     return x_split.d - x_split.d;
13   }
14
15   if (rri.absxhi < XMAX_TAN_CASE2){
16     if (rri.absxhi < XMAX_RETURN_X_FOR_TAN)
17       return x;
18     /* Dynamic computation of the rounding constant */
19     rndcst.i[HI] = 0x3ff00000 + (((rri.absxhi & 0x000fffff)+0x00100000) >> (0x3ff+2 - (rri.
           absxhi>>20))) ;
20     rndcst.i[LO] =0xffffffff;
21     /* Fast Taylor series */
22     x2 = x*x;
23     p5 = t5.d + x2*(t7.d + x2*(t9.d + x2*t11.d));
24     tt = x2*(t3h.d + (t3l.d +x2*p5));
25     Add12(rri.rh, rri.rl, x, x*tt);
26     /* Test if round to nearest achieved */
27     if(rri.rh == (rri.rh + (rri.rl * rndcst.d)))
28       return rri.rh;
29     else
30       return scs_tan_rn(x);
31   }
32   else {
33     /* Otherwise : Range reduction then standard evaluation */
34     rri.x=x;
35     rri.function=TAN;
36     ComputeTrigWithArgred(&rri);
37
38     /* Test if round to nearest achieved */
39     if(rri.rh == (rri.rh + (rri.rl * RN_CST_TAN_CASE3)))
40       if(rri.changesign) return -rri.rh; else return rri.rh;
41     else
42       return scs_tan_rn(x);
43   }
44 }
```

There is a peculiarity in lines 19 and 20: We compute the rounding constant dynamically, out of the value of x. The idea here is that in the neighborhood of zero, both *tan* and its approximation are equivalent to x with no order-2 term, therefore the relative error $\bar{\varepsilon}$ is equivalent to x^2 and therefore $\bar{\varepsilon}/x$ will vanish as $x \to 0$ (of course, in the presence of rounding error this has to be proven more rigorously - we will use Gappa below). As the error constant is computed out of $\bar{\varepsilon}$ (see Theorem 21 page 39), for x sufficiently small we can compute e out of x and get a finer rounding constant, hence a lower probability of going through the accurate phase.

The lines 19-20 implement

$$\texttt{rndcst} \approx 1 + 2^{-2}|x|$$

with

$$\texttt{rndcst} \geq 1 + 2^{-2}|x|$$

ensured by line 20.

To prove that that this rounding constant is correct, we just have to check that it fulills the requirements of Theorem 21.

Let us note $\bar{\varepsilon}_{TanCase2}$ simply $\bar{\varepsilon}$ in this section. First, we compute in Gappa (file `maple/trigoTanCase2.gappa` below), using the same constants (produced by the same Maple) as in the C code, a bound M on $\frac{\bar{\varepsilon}}{|x|}$. We find that, for $x < 2^{-4}$, we have

$$\bar{\varepsilon}_{TanCase2} < 2^{-60.9}$$

(hence the k of Theorem 21 may be chosen as $k = 7$) and

$$\frac{\bar{\varepsilon}}{|x|} < M = 2^{-56.5}.$$

Or,

$$|x| > \bar{\varepsilon}/M$$

Therefore,

$$\texttt{rndcst} > 1 + 2^{-2}/M\bar{\varepsilon}$$

.

It is now trivial to check that $2^{-2}/M > \frac{2^{54}}{(1-2^{-7})(1-2^{-53})}$, therefore for any x the value of `rndcst` thus computed allows to determine correct rounding according to Theorem 21.

Note that the computation of the rounding constant, although complex, is performed in integer arithmetic and independently of the evaluation of the polynomial. Therefore both computations may be carried out in parallel in a superscalar processor.

The input to Gappa is as follows. Note that it needs a bound on $\frac{p(x) - \tan(x)}{x \tan(x)}$ which has been computed as an infinite norm in Maple.

Listing 9.26: Gappa input file to prove the previous bounds on $\bar{\varepsilon}_{TanCase2}$ and $\bar{\varepsilon}_{TanCase2}/x$

```
# Usage: You need to replace a few constants (beginning by _) by numerical
# values. Running the trigo.mpl Maple script will generate a
# TEMPTRIG/tanCase2.sed sed script that does it.
# Then sed -f TEMPTRIG/TanCase2.sed trigoTanCase2.gappa | gappa  > /dev/null


# NOTATION CONVENTION
# Variables that correspond to double-precision variables in the code begin with a small
    letter
# Other variables begin with a capital letter.
# Otherwise avoid underscores as they are painful to carry on to LaTeX :)



# Definition of the polynomial constants:
t11 = <float64ne>(_t11);
t9  = <float64ne>(_t9);
t7  = <float64ne>(_t7);
t5  = <float64ne>(_t5);
```

```
t3h = <float64ne>(_t3h);
t3l = <float64ne>(_t3l);

###########################################################################

# First, a transcription of the actual computation, which could (and
# should eventually) be generated automatically from the actual code

# ------------------------Code cut from tan_rn :
#    x2 = x*x;
x2 <float64ne>= x * x;
X2 = x*x;

#    p5 = t5 + x2*(t7 + x2*(t9 + x2*t11));
p5 <float64ne>= t5 + x2*(t7 + x2*(t9 + x2*t11));
P5             = t5 + X2*(t7 + X2*(t9 + X2*t11));

#    tt = x2*(t3h + (t3l + x2*p5));
tt <float64ne>= x2*(t3h + (t3l + x2*p5));
Tt             = X2*(t3h + (t3l + X2*p5));

#    Add12(rri.rh, rri.rl, x, x*tt);
rdd = x + <float64ne>(x*tt);  # The Add12 is exact

Poly = x+x*Tt;

epsilon=(rdd - TanX)/TanX;

{
    x in [1b-30, _xmax]
 /\ (Poly - TanX)/TanX in [-_maxEpsApprox, _maxEpsApprox]
 /\ ((Poly - TanX)/TanX)/x in [-_maxEpsApproxOverX, _maxEpsApproxOverX]

->

epsilon in ?
/\
epsilon/x in  [-1b-56,1b-56]

}

# Use a dichotomy on x to get an interval of epsilon/x
epsilon/x $ x;

# The usual hint for relative errors, with an additional /x*x so that Gappa uses the Maple-
    computed mathematical bound
(rdd - TanX)/TanX -> (rdd - Poly)/Poly  + (((Poly - TanX)/TanX)/x)*x + ((rdd - Poly)/Poly) *
    ((Poly - TanX)/TanX) ;

(rdd - Poly)/Poly  ->  ((<float64ne>(x*tt) - x*Tt) / x) * (x/Poly);

# I'm not sure I understand why this one improves the result
(<float64ne>(x*tt) - x*Tt)/x -> ((<float64ne>(x*tt) - x*tt)/(x*tt)) * tt   +  (tt - Tt);

# Easy hints
x/Poly -> 1/(Poly/x);
Poly/x -> 1+Tt;
```

9.6.3 RU mode

Listing 9.27: Exceptional cases for tangent RU

```
1  double tan_ru(double x){
2    double epsilon, p5, tt, x2;
3    db_number x_split;
4    rrinfo rri;
5
6    x_split.d=x;
7    rri.absxhi = x_split.i[HI] & 0x7fffffff;
8
9    /* SPECIAL CASES: x=(Nan, Inf) cos(x)=Nan */
10   if (rri.absxhi>=0x7ff00000) {
11     x_split.l=0xfff8000000000000LL;
12     return x_split.d - x_split.d;
13   }
14
```

```
  if (rri.absxhi < XMAX_TAN_CASE2){
    if (rri.absxhi < XMAX_RETURN_X_FOR_TAN) {
      if(x<=0.)
        return x;
      else {
        x_split.l ++;
        return x_split.d;
      }
    }
    else {
      /* Fast Taylor series */
      x2 = x*x;
      p5 = t5.d + x2*(t7.d + x2*(t9.d + x2*t11.d));
      tt = x2*(t3h.d + (t3l.d +x2*p5));
      Add12(rri.rh, rri.rl, x, x*tt);

      /* TODO dynamic computation of error constant */
      TEST_AND_RETURN_RU(rri.rh, rri.rl, EPS_TAN_CASE2);

      /* if the previous block didn't return a value, launch accurate phase */
      return  scs_tan_ru(x);
    }
  }
  else {
    /* Normal case: Range reduction then standard evaluation */
    rri.x=x;
    rri.function=TAN;
    ComputeTrigWithArgred(&rri);
    epsilon=EPS_TAN_CASE3;
    if(rri.changesign) {
      rri.rh= -rri.rh;
      rri.rl=-rri.rl;
    }
  }

  TEST_AND_RETURN_RU(rri.rh, rri.rl, epsilon);

  /* if the previous block didn't return a value, launch accurate phase */
  return  scs_tan_ru(x);
}
```

9.6.4 RD mode

Listing 9.28: Exceptional cases for tangent RD

```
double tan_rd(double x){
  double epsilon, p5, tt, x2;
  rrinfo rri;
  db_number x_split;


  x_split.d=x;
  rri.absxhi = x_split.i[HI] & 0x7fffffff;

  /* SPECIAL CASES: x=(Nan, Inf) cos(x)=Nan */
  if (rri.absxhi>=0x7ff00000){
    x_split.l=0xfff8000000000000LL;
    return x_split.d - x_split.d;

  }

  if (rri.absxhi < XMAX_TAN_CASE2){
    if (rri.absxhi < XMAX_RETURN_X_FOR_TAN) {
      if(x>=0.)
        return x;
      else {
        x_split.l ++;
        return x_split.d;
      }
    }

    /* Fast Taylor series */
    x2 = x*x;
    p5 = t5.d + x2*(t7.d + x2*(t9.d + x2*t11.d));
    tt = x2*(t3h.d + (t3l.d +x2*p5));
    Add12(rri.rh, rri.rl, x, x*tt);
```

```
32
33      TEST_AND_RETURN_RD(rri.rh, rri.rl, EPS_TAN_CASE2);
34
35      /* if the previous block didn't return a value, launch accurate phase */
36      return  scs_tan_rd(x);
37    }
38
39    else {
40      /* normal case: Range reduction then standard evaluation */
41      rri.x=x;
42      rri.function=TAN;
43      ComputeTrigWithArgred(&rri);
44      epsilon=EPS_TAN_CASE3;
45      if(rri.changesign) {
46        rri.rh= -rri.rh;
47        rri.rl=-rri.rl;
48      }
49    }
50
51    TEST_AND_RETURN_RD(rri.rh, rri.rl, epsilon);
52
53    /* if the previous block didn't return a value, launch accurate phase */
54    return  scs_tan_rd(x);
55  }
```

9.6.5 RZ mode

Listing 9.29: Exceptional cases for tangent RZ

```
1   double tan_rz(double x){
2     double epsilon, p5, tt, x2;
3     rrinfo rri;
4     db_number x_split;
5
6     x_split.d=x;
7     rri.absxhi = x_split.i[HI] & 0x7fffffff;
8
9     /* SPECIAL CASES: x=(Nan, Inf) cos(x)=Nan */
10    if (rri.absxhi>=0x7ff00000) {
11      x_split.l=0xfff8000000000000LL;
12      return x_split.d - x_split.d;
13    }
14
15    if (rri.absxhi < XMAX_TAN_CASE2){
16      if (rri.absxhi < XMAX_RETURN_X_FOR_TAN) {
17        return x;
18      }
19      else{
20        /* Fast Taylor series */
21        x2 = x*x;
22        p5 = t5.d + x2*(t7.d + x2*(t9.d + x2*t11.d));
23        tt = x2*(t3h.d + (t3l.d +x2*p5));
24        Add12(rri.rh, rri.rl, x, x*tt);
25
26        TEST_AND_RETURN_RZ(rri.rh, rri.rl, EPS_TAN_CASE2);
27
28        /* if the TEST_AND_RETURN block didn't return a value, launch accurate phase */
29        return  scs_tan_rz(x);
30      }
31    }
32    else {
33      /* Normal case: Range reduction then standard evaluation */
34      rri.x=x;
35      rri.function=TAN;
36      ComputeTrigWithArgred(&rri);
37      epsilon=EPS_TAN_CASE3;
38      if(rri.changesign) {
39        rri.rh = -rri.rh;
40        rri.rl = -rri.rl;
41      }
42    }
43
44    TEST_AND_RETURN_RZ(rri.rh, rri.rl, epsilon);
45
46    /* if the previous block didn't return a value, launch accurate phase */
47    return  scs_tan_rz(x);
```

```
48 }
```

9.7 Accurate phase

For simplicity, the accurate phase (in file `trigo_accurate.c`) always computes a Payne and Hanek range reduction to $[-\pi/4, \pi/4]$, then a polynomial evaluation using a Taylor formula.

The results of the search for worst cases are the following so far:

Function	interval	worst-case accuracy
$\sin(x)$	$\lvert x\rvert < 2^{-17}$	2^{-126}
	$2^{-17} \leq \lvert x\rvert \leq 2 + \frac{4675}{8192}$	2^{-119}
$\cos(x)$	$2^{-25} \leq \lvert x\rvert \leq 2^{-22}$	2^{-142}
	$2^{-22} \leq \lvert x\rvert \leq 2^{-18}$	2^{-136}
	$2^{-18} \leq \lvert x\rvert \leq 2^{-17}$	2^{-114}
	$2^{-17} \leq \lvert x\rvert \leq \frac{12867}{8192}$	2^{-112}
$\tan(x)$	$2^{-25} \leq \lvert x\rvert \leq 2^{-18}$	2^{-132}
	$2^{-18} \leq \lvert x\rvert \leq \arctan(2)$	2^{-111}

The polynomials used are Pade approximation computed in `maple/trigo.mpl`. This maple script produces the `trigo.h` file, and also prints out the approximation error, as follows.

- of degree 25 for sine, with an approximation error lower than 2^{-125} on $[-\pi/4, \pi/4]$, and lower than 2^{-158} for $|x| < 2^{-17}$,
- of degree 26 for cosine, with an approximation error lower than 2^{-132} on $[-\pi/4, \pi/4]$, and lower than 2^{-168} for $|x| < 2^{-18}$,
- of degree 69 for the tangent, with an approximation error lower than 2^{-30} on $[-\pi/4, \pi/4]$, and lower than 2^{-163} for $|x| < 2^{-18}$.

The polynomial evaluation is an Horner scheme in SCS, ensuring better than 2^{-200} accumulated roundoff errors. Therefore, the overall evaluation error for the accurate phase is lower than the worst case accuracy for each function on each interval of the previous table.

This Maple script, and `crlibm` in general, are designed to allow easy increase of the accuracy, should cases worst than those of this table be found in the future.

9.8 Performance results

Table 9.1 gives performance results for input numbers with random mantissa and exponents uniformely distributed between -20 and 40, by the command:
`tests/crlibm_testperf sin RN 10000`.

In this case the second step was taken 3 times out of 10000.

Which input interval should be used to measure the performance of trigonometric functions is an open question for us. For larger exponents, `libultim` is faster than `crlibm`.

Results for the cosine are very similar. The tangent leaves more room for improvement, as Table 9.2 shows. The culprit is the Div22 procedure, which is very expensive.

Directed rounding mode have a penalty of about 50 cycles on a Pentium III, due to the heavy use of integer 64-bit arithmetic.

Pentium III / Linux Debian sarge / gcc 3.3			
	min time	avg time	max time
`libm`	108	118	142
`mpfr`	16715	67153	186925
`libultim`	91	300	1294619
`crlibm`	81	229	13616

Table 9.1: Absolute timings for the sine (arbitrary units)

Pentium III / Linux Debian sarge / gcc 3.3			
	min time	avg time	max time
`libm`	158	167	183
`mpfr`	22759	80108	222550
`libultim`	113	428	1357592
`crlibm`	105	367	33830

Table 9.2: Absolute timings for the tangent (arbitrary units)

Chapter 10

The arcsine

This chapter is contributed by Ch. Q. Lauter.

WARNING: This chapter is out-of-sync with the code. The function was completely rewritten using machine-generated polynomials. Gappa proofs were automatically generated, too, and are available in the `gappa/asin` directory. This chapter will be updated soon.

10.1 Overview of the algorithm

The arcsine $\arcsin(x) = \sum_{i=0}^{\infty} \frac{(2i-1)!!}{(2i+1)\cdot(2i)!!} \cdot x^{2i+1}$ is defined on the domain $x \in [-1;1]$. It is a odd function: $\arcsin(-x) = -\arcsin(x)$. Its value in 0 is $\arcsin(0) = 0$. Its derivative tends to infinity when x tends to 1: $\lim_{x\to 1} \left(\frac{d}{dx}\arcsin\right)(x) = \infty$; the function's value in 1 is nevertheless finite: $\arcsin(1) = \frac{\pi}{2}$. There is no simple additive or multiplicative decomposition of this function.

A correctly rounded implementation of arcsin must provide an accuracy of at least 126 bits for $|x| \leq 2^{-18}$ and of at least 118 bits for the rest of the definition domain in the accurate phase [11].

The algorithm chosen principally consists of a piecewise polynomial approximation either of the function itself or of a asymptotic development of the function. More precisely, the following is done:

- Special cases, such as $|x| > 1$, $x = \pm\infty$, $x = \text{NaN}$, are handled.
- The sign of the argument x is stripped off because $\arcsin(x) = \text{sgn}(x) \cdot \arcsin(|x|)$. We will suppose in the following, that x stands for a positive argument, $x \geq 0$.
- The argument is classified in one of 10 subdomains of $[0;1]$. This means an integer $i \in [0 \dots 9]$ is computed such that $x \in I_i$ where
$$\bigcup_{i=0}^{9} I_i = [0;1]$$
$$i \neq j \Rightarrow I_i \cap I_j = \emptyset$$
$$i < j \Rightarrow \forall x_i \in I_i, x_j \in I_j \,.\, x_i < x_j$$
- If $i = 0$, arcsin is directly approximated as
$$\arcsin(x) \approx x + x^3 \cdot p_0\left(x^2\right)$$
- If $1 \leq i \leq 8$, an interval midpoint value $m_i \approx \frac{\inf I_i + \sup I_i}{2}$ is read in a table. The function is then approximated as
$$\arcsin(x) \approx \arcsin(m_i) + (x - m_i) \cdot p_i(x - m_i)$$
- If $i = 9$, arcsin is approximated as
$$\arcsin(x) \approx p_9(1-x) \cdot \sqrt{2 - 2\cdot x} + \frac{\pi}{2}$$

The polynomials p_i, $1 \leq i \leq 8$, for the middle intervals are all of the same degree and, loaded from a table, can be evaluated in the same computation path. The polynomials p_0 and p_9 are of different degree. So there are three distinct paths in the code. In the following, they are referred to as the low, middle and high path.

Concerning the quick and accurate phase of the implementation, it must mentioned that code and cache size considerations do not allow for using different coefficient tables for the quick phase polynomials $p_{i\,\text{quick}}$ as for the accurate phase polynomials $p_{i\,\text{accurate}}$ that are obviously longer and must contain more accurately stored coefficients. So the quick phase polynomials are simply the accurate phase polynomials truncated to some degree and simplified by omitting low significance components of the coefficients.

The double precision midpoint values $m_i \in \mathbb{F}$ for the middle path intervals I_i, $1 \leq i \leq 8$, are chosen such that a double-double approximation $asinm_{ih} + asinm_{il}$ of $\arcsin(m_i)$ is accurate to at least 121 bits. This allows for saving up memory in the tables used.

The intervals I_i are not uniformly distributed. This has the disadvantage that the computation of i cannot be done by simple bitmasks on the arcsine's argument x but that a dichotomy must be performed. On the other hand, this is the only way of using polynomials for the same degree for all middle intervals without wasting accuracy for the lower ones. In fact, the derivative of arcsin grows over-polynomially, which means that the polynomial degrees must increase for equally sized reduced arguments in order to achieve the same approximation error.

The decomposition of the domain $[0;1]$ into the I_is is relatively ad-hoc. The given implementation uses:

$$I_i = [bound_i; bound_{i+1}]$$

with

$$\begin{aligned}
bound_0 &= 0 \\
bound_1 &= 0.184999942779541015625 \\
bound_2 &= 0.2997171878814697265625 \\
bound_3 &= 0.40296268463134765625 \\
bound_4 &= 0.4932067394256591796875 \\
bound_5 &= 0.5696849822998046875 \\
bound_6 &= 0.639662265777587890625 \\
bound_7 &= 0.696256160736083984375 \\
bound_8 &= 0.741760730743408203125 \\
bound_9 &= 0.77999973297119140625 \\
bound_{10} &= 1
\end{aligned}$$

One remarks that in order to simplify the dichotomy for computing i, the bounds of the intervals are chosen all such that the low order word of the double precision numbers they are stored in are 0. See 10.2, page 123 for more precise considerations on that subject.

Let be

$$z_i = \begin{cases} x & \text{if } i = 0 \\ x - m_i & \text{if } 1 \leq i \leq 8 \\ 1 - x & \text{if } i = 9 \end{cases}$$

This value z_i is the argument to the polynomial p_i. With the given interval bounds, it is bounded by

$$\begin{aligned}
|z_1| &\leq 2^{-2.434403} \\
|z_2| &\leq 2^{-4.123846} \\
|z_3| &\leq 2^{-4.275849} \\
|z_4| &\leq 2^{-4.470024} \\
|z_5| &\leq 2^{-4.708807} \\
|z_6| &\leq 2^{-4.836970}
\end{aligned}$$

$$|z_7| \leq 2^{-5.143210}$$
$$|z_8| \leq 2^{-5.457845}$$
$$|z_9| \leq 2^{-5.708811}$$
$$|z_{10}| \leq 2^{-2.184423}$$

The degrees of the polynomials p_i and the number of double (D), double-double (DD) and triple-double (TD) coefficients stored in the table are listed below. Here the degree of the polynomial is the highest exponent of the monomial whose coefficient is not equal to 0. We repeat that the p_i are such that

$$\arcsin(x) \approx x + x^3 \cdot p_0\left(x^2\right)$$
$$\arcsin(x) \approx \arcsin(m_i) + (x - m_i) \cdot p_i(x - m_i), \ 1 \leq i \leq 8$$
$$\arcsin(x) \approx p_9(1 - x) \cdot \sqrt{2 - 2 \cdot x} + \frac{\pi}{2}$$

i	quick phase	accurate phase	D quick	DD quick	D accur.	DD accur.	TD accur.
0	8	17	5	4	7	6	5
1...8	13	34	8	6	20	9	6
9	18	28	10	8	12	9	8

Remark that the listing does not account neither for the 8 double-double values representing approximations to $\arcsin(m_i)$ nor for the 8 double precision numbers m_i.

Taking into account also these values, the overall table size for both quick and accurate phase is 4640 bytes. Some additional values, for example the triple-double $PiHalf_h + PiHalf_m + PiHalf_l \approx \frac{\pi}{2}$, interval bounds or table indices, are directly compiled into the code. Their overall size is 100 bytes.

10.2 Special case handling, interval discrimination and argument reduction

As already mentioned, arcsin is only defined on the domain $[-1; 1]$. For other arguments, including $\pm\infty$ and NaN, NaN must be returned. This is implemented in the code as follows: the sign of x is stripped off by integer computations and stored in variable `sign`. Than, $|x|$ is compared to 1 by integer comparisons.

Listing 10.1: Handling special cases - definition domain

```
/* Transform the argument into integer */
xdb.d = x;

/* Special case handling */

/* Strip off the sign of argument x */
if (xdb.i[HI] & 0x80000000) sign = -1; else sign = 1;
xdb.i[HI] &= 0x7fffffff;

/* asin is defined on -1 <= x <= 1, elsewhere it is NaN */
if ((xdb.i[HI] > 0x3ff00000) || ((xdb.i[HI] == 0x3ff00000) && (xdb.i[LO] != 0x00000000))) {
  return (x-x)/0.0;     /* return NaN */
}
```

Concerning subnormals in argument and in result of the function, the following is to be mentioned. The Taylor series of arcsin developed in 0 is

$$\arcsin(x) = x \cdot \left(1 + \frac{1}{6} \cdot x^2 + \sum_{n=2}^{\infty} \frac{(2n-1)!!}{(2n)!! \cdot (2n+1)} \cdot x^{2n}\right)$$

It is easy to check that $\sum_{n=2}^{\infty} \frac{(2n-1)!!}{(2n)!! \cdot (2n+1)} \cdot x^{2n} < \frac{1}{3} \cdot x^2$ for $|x| < \frac{1}{2}$. So for $|x| \leq 2^{-28}$, one gets $\arcsin(x) \leq x \cdot \left(1 + \frac{1}{2} \cdot x^2\right) \leq x \cdot \left(1 + 2^{-57}\right) < x + \frac{1}{2}\text{ulp}(x)$. So the rounding can be decided without even computing

the cubic term of the Taylor development. Thus subnormals in argument and in result can be avoided by performing a simple test on the absolute value of x. In particular, since $\sum_{n=1}^{\infty} \frac{(2n-1)!!}{(2n)!!\cdot(2n+1)} \cdot x^{2n}$ is an even function, the sign of the truncation rest is known, which allows for simplifications in the directed rounding modes. Here, only some special care is needed for the case where x is exactly equal to 0.

The test and the rounding are implemented as follows. Let us first consider the round-to-nearest case:

Listing 10.2: Handling special cases - rounding (to nearest)

```
if (xdb.i[HI] < 0x3e300000) {
  return x;
}
```

In the round-upward case, a correction of x is potentially necessary. We implement:

Listing 10.3: Handling special cases - rounding (upwards)

```
/* If x == 0 then we got the algebraic result arcsin(0) = 0
   If x < 0 then the truncation rest is negative but less than
   1 ulp; we round upwards by returning x
*/
if (x <= 0) return x;
/* Otherwise the rest is positive, less than 1 ulp and the
   image is not algebraic
   We return x + 1ulp
*/
xdb.l++;
return xdb.d;
```

The other directed rounding cases are analogous to the round-upwards case.

For the discrimination of the argument $|x|$ in the 10 possible approximation intervals I_i, the following technique is used. The intervals at the definition domain borders I_0 and I_9 are first filtered out by tests checking the high order word of x against the corresponding bounds. If x is found to be in one of these two intervals, the function is approximated in quick and if needed in accurate phase and the correctly rounded value is returned. In any case, the two intervals have particular properties in comparison to the other 8 middle intervals, so this technique should not be considered as a performance disadvantage. The polynomial coefficients' indices in the main coefficient table are fixed in this case and directly compiled into the code via macros.

If x does not fall in one of the both border intervals I_0 and I_9, the corresponding interval I_i, $1 \leq i \leq 8$ is computed by a 3-level dichotomy on the bounds $bound_2 \ldots bound_8$. Its result is not a number i in $1 \leq i \leq 8$ but an index `i` to the main coefficient table. Beginning at the point indexed, the table reads the midpoint value m_i and the polynomial coefficients for the corresponding interval I_i.

The correponding code is the following:

Listing 10.4: Interval discrimination

```
/* Recast x */
x = xdb.d;

/* Find correspondant interval and compute index to the table
   We start by filtering the two special cases around 0 and 1
*/

if (xdb.i[HI] < BOUND1) {

– Compute quick and potentially accurate phase polynomial approximation p0 and return –

}

if (xdb.i[HI] > BOUND9) {

– Reduce the argument, compute quick and potentially accuratephase approximation using p9, reconstruct and return –
}

/* General 8 main intervals
   We can already suppose that BOUND1 <= x <= BOUND9
*/

if (xdb.i[HI] < BOUND5) {
```

```
  if (xdb.i[HI] < BOUND3) {
    if (xdb.i[HI] < BOUND2) i = TBLIDX2; else i = TBLIDX3;
  } else {
    if (xdb.i[HI] < BOUND4) i = TBLIDX4; else i = TBLIDX5;
  }
} else {
  if (xdb.i[HI] < BOUND7) {
    if (xdb.i[HI] < BOUND6) i = TBLIDX6; else i = TBLIDX7;
  } else {
    if (xdb.i[HI] < BOUND8) i = TBLIDX8; else i = TBLIDX9;
  }
}

– Reduce the argument, compute quick and potentially accurate phase polynomial approximation p_i and return –
```

In the case of x being classified in either the middle or the higher intervals I_i, $1 \leq i \leq 9$, an argument reduction is to be performed. Let us consider it first for the high path interval I_9 and then for the middle path intervals I_i, $1 \leq i \leq 8$. In both cases, we will show that the argument reduction is mathematically exact and that it may not produce a subnormal different from 0.

In the high path, we know that $1 \geq x > bound_9 > 0.77$. The argument reduction to be performed is $z = 1 - x$. Since $\frac{1}{2} \leq x \leq 2$ is verified, we can implement it exactly thanks to Sterbenz' lemma. If x is exactly equal to 1 is produces exactly 0. Otherwise, it may not produce a subnormal, because $x \leq 1 - \frac{1}{2} \cdot \text{ulp}(1) \leq 1 - 2^{-53}$. Thus $z = 1 - x > 2^{-53} > 2^{-1021}$.

In the middle path intervals, the argument reduction to be performed is $z_i = x - m_i$. Since $|z| \leq 0.058$ and $x \geq 0.18$, Sterbenz' lemma is verified in each interval I_i and we can still implement the argument reduction exactly in double precision arithmetic. Since, $x > 0.18 > \frac{1}{8}$, a similar argument as the one given above shows that the result of the reduction is either exactly 0 or a non-subnormal double precision number.

The value m_i is read in the main table at the index `i` computed by the interval discrimination phase. We implement thus:

Listing 10.5: Argument reduction

```
/* Argument reduction
   i points to the interval midpoint value in the table
*/
z = x - tbl[i];
```

Concerning the higher path interval I_9 where arcsin is approximated as

$$\arcsin(x) \approx p_9(1-x) \cdot \sqrt{2 - 2 \cdot x} + \frac{\pi}{2}$$

let us remark that $2 - 2 \cdot x = 2 \cdot z$ can also be computed exactly. Trivially, since z less than 2^{1023}, the multiplication by a positive integer power of 2 is errorfree. If z is not exactly 0, its result may not be subnormal because z cannot.

Since the argument reduction has been shown to be exact, its result can clearly be reused in the accurate phase.

10.3 Polynomial approximation and reconstruction

10.3.1 Quick phase polynomial approximation and reconstruction

As already mentioned, the quick phase polynomials $p_{i\,\text{quick}}$ are truncated versions of the accurate phase polynomials $p_{i\,\text{accurate}}$ with coefficients rounded from triple-double to double-double or from double-double to double. This means simply that not all coefficients are read and used.

Low path - interval I_0

The polynomial $p_{0\,\text{quick}}$ approximates arcsin in the interval I_0 as follows:

$$\arcsin(x) \approx x + x^3 \cdot p_{0\,\text{quick}}\left(x^2\right)$$

It is of degree 8 with 5 double-double and 4 double precision coefficients.

For arguments $|x| \leq 2^{-10}$, the polynomial needs not be evaluated fully to provide enough accuracy. Here, only its constant and linear term are evaluated. So we get in this case

$$\arcsin(x) \approx x + x^3 \cdot \Big((c_{0h} + c_{0l}) + x^2 \cdot (c_{1h} + c_{1l}) \Big)$$

This special path yields to a significant performance gain on average. In fact, since floating point numbers are not equispaced but distributed logarithmatically around 0, speeding up a function for low arguments is worth it.

The square of x, x^2 can be computed exactly by use of a **Mul12** sequence, which will produce a double-double $xSq_h + xSq_l = x^2$. The polynomial $p_{0\,\mathrm{quick}}(xSq_h + xSq_l)$ is evaluated using Horner's scheme and neglecting xSq_l for the 8 higher degree coefficients if these need to be evaluated.

The double-double precision Horner steps are implemented the double-double multiply-and-add macros **MulAdd22** and **MulAdd212** (see section 2.3.6, page 24). It is easy to check that the preconditions on the arguments of these macros are verified: x^2 is bounded by $x^2 \leq \left(2^{-2.434403}\right)^2 \leq 2^{-4}$ in this path. Further, in the order of the Horner evaluation, the coefficients c_i of the polynomial are stricly increasing in magnitude and all less than 1. Concerning the accuracy of this operations, see section 10.4.1, page 132.

Including the test $|x| \overset{?}{\leq} 2^{-10}$, the code computing an approximation $t_{5h} + t_{5l} \approx p_{0\,\mathrm{quick}}\left(x^2\right)$ reads:

Listing 10.6: Low path quick phase polynomial approximation (higher degrees)

```
Mul12(&xSqh,&xSql,x,x);

tmp4 = tbl[3];
tmp5 = tbl[4];
t4h = tmp4;
t4l = tmp5;
if (xdb.i[HI] > EXTRABOUND) {
  /* Double precision evaluation */
  highPoly = tbl[15] + xSqh * (tbl[17] + xSqh * (tbl[19] + xSqh * (tbl[21] + xSqh * tbl[23])))
      ;

  /* Double-double precision evaluation */
  Mul12(&tt1h,&tt1l,xSqh,highPoly);
  Add22(&t1h,&t1l,tbl[12],tbl[13],tt1h,tt1l);

  MulAdd212(&t2h,&t2l,tbl[9],tbl[10],xSqh,t1h,t1l);
  MulAdd212(&t3h,&t3l,tbl[6],tbl[7],xSqh,t2h,t2l);
  MulAdd22(&t4h,&t4l,tmp4,tmp5,xSqh,xSql,t3h,t3l);
}

MulAdd22(&t5h,&t5l,tbl[0],tbl[1],xSqh,xSql,t4h,t4l);
```

Once $t_{5h} + t_{5l}$ are computed, they must be multiplied by x^3 and the result must be added to x. The value x^3 is computed approximatively as a double-double $xCube_h + xCube_l$ by multiplying x by $xSq_h + xSq_l = x^2$. This value $xCube_h + xCube_l$ is than multiplied by $t_{5h} + t_{5l}$, yielding to $tt_{6h} + tt_{6l}$. The last addition implying x and $tt_{6h} + tt_{6l}$ is implemented in an ad-hoc way by means of two exact additions and a double precision addition on the lower part of the addition of the higher significant parts. The code reads:

Listing 10.7: Low path quick phase polynomial approximation (lower degrees)

```
Mul122(&xCubeh,&xCubel,x,xSqh,xSql);
Mul22(&tt6h,&tt6l,xCubeh,xCubel,t5h,t5l);

Add12(tmp1,tmp2,x,tt6h);
tmp3 = tmp2 + tt6l;
Add12(polyh,polyl,tmp1,tmp3);
```

The obtained polynomial approximation value is then multiplied by the sign of x the argument reduction had stripped off and the rounding test is performed. If it fails, the accurate phase is launched; see section 10.3.2, page 129, for its implementation. The code for the last steps in round-to-nearest mode is given below. The code for the direct rounding modes is analogous. One remarks that the rounding test constants, computed by the corresponding Maple scripts in function of the relative error bounds to be shown in section 10.4.1, page 132, are stored also in double precision and also read from the main table.

Listing 10.8: Low path quick phase rounding test

```
/* Multiply by sign */
asinh = sign * polyh;
asinm = sign * polyl;

/* Rounding test
   The RN rounding constant is at tbl[34]
*/
if(asinh == (asinh + (asinm * tbl[34])))
  return asinh;

/* Launch accurate phase */
```

Middle path - interval I_i, $1 \leq i \leq 8$

In the quick phase middle path, i.e. for arguments $x \in I_i$, $1 \leq i \leq 8$, arcsin is approximated by the polynomial $p_{i\,\text{quick}}$ as follows:

$$\arcsin(x) \approx \arcsin(m_i) + z_i \cdot p_{i\,\text{quick}}(z_i)$$

where $z_i = x - m_i$ is a double precision number. Further an approximation $asm_{ih} + asm_{il}$ to $\arcsin(m_i)$ is used. It is read in the main table at `tbl[i+1]` and `tbl[i+2]` where `i` is the index computed at the interval discrimination phase. The polynomial $p_{i\,\text{quick}}$ is of degree 13 with 6 double-double and 8 double coefficients. It is always evaluated completely by means of Horner's scheme. Once again, the **MulAdd212** operator allows for evaluating the 5 last double-double steps. The first double-double step is evaluated in a more ad-hoc way because of the entering only double precision current intermediate result. The preconditions for the **MulAdd212** operator can again be checked easily: z_i is bounded in magnitude by $2^{-4.123846}$, the coefficients increase in the order of Horner's scheme evaluation and are all less than 1. So in each step, the product of z_i and the current value is less than $\frac{1}{4}$ the next coefficient as asked for by the precondition. The multiplication of the result of the polynomial $p_{i\,\text{quick}}$ by z_i and the addition of $asm_{ih} + asm_{il}$ can also be considered as a Horner step and are therefore implemented using the **MulAdd212** macro, too.

The corresponding evaluation code reads thus:

Listing 10.9: Middle path quick phase polynomial approximation

```
highPoly = tbl[i+21] + z * (tbl[i+23] + z * (tbl[i+25] + z * (
           tbl[i+27] + z * (tbl[i+29] + z * (tbl[i+31] + z * (
           tbl[i+33] + z *  tbl[i+35])))));

Mul12(&tt1h,&tt1l,z,highPoly);
Add22(&t1h,&t1l,tbl[i+18],tbl[i+19],tt1h,tt1l);

MulAdd212(&t2h,&t2l,tbl[i+15],tbl[i+16],z,t1h,t1l);
MulAdd212(&t3h,&t3l,tbl[i+12],tbl[i+13],z,t2h,t2l);
MulAdd212(&t4h,&t4l,tbl[i+9],tbl[i+10],z,t3h,t3l);
MulAdd212(&t5h,&t5l,tbl[i+6],tbl[i+7],z,t4h,t4l);
MulAdd212(&t6h,&t6l,tbl[i+3],tbl[i+4],z,t5h,t5l);
MulAdd212(&polyh,&polyl,tbl[i+1],tbl[i+2],z,t6h,t6l);
```

One remarks that in this case, since the polynomial evaluation code is the same for all intervals I_i, $1 \leq i \leq 8$, the coefficients read in the table are not at fixed indices but indexed by `i`, value computed in the interval discrimination phase.

The result of this approximation whose accuracy will be analysed in section 10.4.1, page 132, is then multiplied by the sign of the the original argument and submitted to the rounding test whose rounding constant is read in the main table dependingly on the interval I_i. The corresponding code for round-to-nearest is given below. The directed rounding cases are analogous.

Listing 10.10: Middle path quick phase rounding test

```
asinh = sign * polyh;
asinm = sign * polyl;

/* Rounding test
   The RN rounding constant is at tbl[i+59]
*/
if(asinh == (asinh + (asinm * tbl[i+59])))
```

```
  return asinh;

/* Launch accurate phase */
```

High path - interval I_9

In the high path arcsin is approximated as

$$\arcsin(x) \approx p_{9\,\text{quick}}(z) \cdot \sqrt{2 \cdot z} + \frac{\pi}{2}$$

Herein, $z = 1 - x$ is the exactly computed double precision reduced argument. The constant $\frac{\pi}{2}$ is approximated by the double-double number $PiHalf_h + PiHalf_m = \frac{\pi}{2} \cdot (1 + \varepsilon)$ with $|\varepsilon| \leq 2^{-109}$. It has already been shown that z and $twoZ = 2 \cdot z$ can be computed exactly. The square root $\sqrt{2 \cdot z}$ is approximated in double-double precision using the **sqrt12** macro operator described in section 2.6.1, page 35.

The polynomial $p_{9\,\text{quick}}$ has degree 18 with 9 double-double and 10 double precision coefficients. Its constant term is exactly -1, so this coefficient is not stored in the table. It is always evaluated completely. Horner's scheme is used and implemented by means of the **MulAdd212** operator whose precodnitions can once again easily be verified by an analogous argument as the one given above for low and middle paths.

The corresponding code reads:

Listing 10.11: High path quick phase polynomial approximation

```
highPoly = tbl[TBLIDX10+24] + z * (tbl[TBLIDX10+26] + z * (tbl[TBLIDX10+28] + z * (
           tbl[TBLIDX10+30] + z * (tbl[TBLIDX10+32] + z * (tbl[TBLIDX10+34] + z * (
           tbl[TBLIDX10+36] + z * (tbl[TBLIDX10+38] + z * (tbl[TBLIDX10+40] + z *
           tbl[TBLIDX10+42])))))));

Mul12(&tt1h,&tt1l,z,highPoly);
Add22(&t1h,&t1l,tbl[TBLIDX10+21],tbl[TBLIDX10+22],tt1h,tt1l);

MulAdd212(&t2h,&t2l,tbl[TBLIDX10+18],tbl[TBLIDX10+19],z,t1h,t1l);
MulAdd212(&t3h,&t3l,tbl[TBLIDX10+15],tbl[TBLIDX10+16],z,t2h,t2l);
MulAdd212(&t4h,&t4l,tbl[TBLIDX10+12],tbl[TBLIDX10+13],z,t3h,t3l);
MulAdd212(&t5h,&t5l,tbl[TBLIDX10+9],tbl[TBLIDX10+10],z,t4h,t4l);
MulAdd212(&t6h,&t6l,tbl[TBLIDX10+6],tbl[TBLIDX10+7],z,t5h,t5l);
MulAdd212(&t7h,&t7l,tbl[TBLIDX10+3],tbl[TBLIDX10+4],z,t6h,t6l);
MulAdd212(&t8h,&t8l,tbl[TBLIDX10+0],tbl[TBLIDX10+1],z,t7h,t7l);
MulAdd212(&polyh,&polyl,-1,0,z,t8h,t8l);
```

The result $poly_h + poly_l$ of the polynomial approximation is multiplied by $sqrtz_h + sqrtz_l$, the double-double approximation of $\sqrt{2 \cdot z}$ by the double-double multiplication operator **Mul22** and then added to $PiHalf_h + PiHalf_m$ using the double-double addition operator **Add22**. In this addition no catastrophic cancellation can occur: since $x > 0.78 > \frac{\sqrt{2}}{2}$ in this interval, the result of the addition, a good approximation to $\arcsin(x)$ will always be greater than $\frac{\pi}{4}$ as per the monotony of arcsine. So $pTimesS_h + pTimesS_l \approx p_{9\,\text{quick}}(z) \cdot \sqrt{2 \cdot z}$ will always be less than $\frac{1}{2} \cdot (PiHalf_h + PiHalf_m)$. In this argumentation, approximation errors can be neglected since the bound to be shown is not tight at all.

The corresponding implementation is the following:

Listing 10.12: High path: square root extraction and reconstruction

```
twoZ = 2 * z;
sqrt12(&sqrtzh,&sqrtzl,twoZ);

/* Multiply p(z) by sqrt(2*z) and add Pi/2 */

Mul22(&pTimesSh,&pTimesSl,polyh,polyl,sqrtzh,sqrtzl);
Add22(&allh,&alll,PIHALFH,PIHALFM,pTimesSh,pTimesSl);
```

In this path, too, the obtained result is multiplied by the sign which had stripped off from the function's argument x and submitted to the rounding test using a rounding constant read in the main table. The code reads for round-to-nearest mode:

Listing 10.13: Multiplication of the function's sign, rounding test (round-to-nearest)

```
asinh = sign * allh;
asinm = sign * alll;

/* Rounding test
   The RN rounding constant is at tbl[TBLIDX10+54]
*/
if(asinh == (asinh + (asinm * tbl[TBLIDX10+54])))
  return asinh;

/* Launch accurate phase */
```

10.3.2 Accurate phase polynomial approximation and reconstruction

As already mentioned, in the accurate phase, the polynomials $p_{i\,\text{accurate}}$ for the different intervals I_i whose coefficient are stored in the main table, are evaluated completely. This evaluation is done in double, double-double and triple-double precision using mainly Horner's scheme.

The accurate phase is implemented in three different `C` functions corresponding to the low, middle and high paths. These functions return a triple-double approximate $asin_h + asin_m + asin_l$ to arcsin which is accurate enough that the correct rounded result for arcsin is obtained when rounding the approximate. The final rounding for itself is implemented in the four functions containing the quick phase code and the call to the accurate phase for the four possible rounding modes. In round-to-nearest mode, for the high path the code reads:

Listing 10.14: Final rounding of the accurate phase result (RN)

```
/* Launch accurate phase */

asin_accurate_higher(&asinh,&asinm,&asinl,z,sign);

ReturnRoundToNearest3(asinh,asinm,asinl);
```

For the other rounding modes and evaluation paths the code is completely analogous.

Low path - interval I_0

For the low path, the polynomial $p_{0\,\text{accurate}}$ approximates arcsin as follows:

$$\arcsin(x) \approx x + x^3 \cdot p_{0\,\text{accurate}}\left(x^2\right)$$

is of degree 17 with 7 double precision coefficients and Horner steps, 6 double-double coefficients and steps and 5 triple-double coefficients and steps. It reuses the value $xSq_h + xSq_l = x^2$ which has already been computed in the quick phase. The lower significant term xSq_l is neglected for the first 8 double precision (and early double-double precision) steps. The double-double Horner steps are implemented using the **MulAdd22** sequence. The triple-double steps used separate **Add33** and **Mul33** macros. The value x^3 is computed out of x and x^2 in triple-double precision using the **Mul123** operator. It is multiplied by the polynomial's result in triple-double by a **Mul33**. The resulting triple-double is added to x using the **Add133** macro.

In addition to the final renormalization which is needed before correct rounding to double precision is performed, one intermediate renormalization is necessary. Otherwise overlap in the triple-double intermediate values would deteriorate the accuracy to much. See section 10.4.2, page 132 for the overlap bounding.

The code of the low path accurate phase polynomial approximation is the following:

Listing 10.15: Low path accurate phase polynomial approximation

```
highPoly = tbl[28] + xSqh * (tbl[29] + xSqh * (tbl[30] + xSqh * (tbl[31] + xSqh * (tbl[32] +
    xSqh * tbl[33]))));

/* Double-double computations */

Mul12(&tt1h,&tt1l,xSqh,highPoly);
Add22(&t1h,&t1l,tbl[27],0,tt1h,tt1l);

```

```
MulAdd22(&t2h,&t2l,tbl[25],tbl[26],xSqh,xSql,t1h,t1l);
MulAdd22(&t3h,&t3l,tbl[23],tbl[24],xSqh,xSql,t2h,t2l);
MulAdd22(&t4h,&t4l,tbl[21],tbl[22],xSqh,xSql,t3h,t3l);
MulAdd22(&t5h,&t5l,tbl[19],tbl[20],xSqh,xSql,t4h,t4l);
MulAdd22(&t6h,&t6l,tbl[17],tbl[18],xSqh,xSql,t5h,t5l);
MulAdd22(&t7h,&t7l,tbl[15],tbl[16],xSqh,xSql,t6h,t6l);

/* Triple-double computations */

Mul23(&tt8h,&tt8m,&tt8l,xSqh,xSql,t7h,t7l);
Add33(&t8h,&t8m,&t8l,tbl[12],tbl[13],tbl[14],tt8h,tt8m,tt8l);
Mul233(&tt9h,&tt9m,&tt9l,xSqh,xSql,t8h,t8m,t8l);
Add33(&t9h,&t9m,&t9l,tbl[9],tbl[10],tbl[11],tt9h,tt9m,tt9l);
Mul233(&tt10h,&tt10m,&tt10l,xSqh,xSql,t9h,t9m,t9l);
Add33(&t10h,&t10m,&t10l,tbl[6],tbl[7],tbl[8],tt10h,tt10m,tt10l);
Mul233(&tt11hover,&tt11mover,&tt11lover,xSqh,xSql,t10h,t10m,t10l);

Renormalize3(&tt11h,&tt11m,&tt11l,tt11hover,tt11mover,tt11lover);

Add33(&t11h,&t11m,&t11l,tbl[3],tbl[4],tbl[5],tt11h,tt11m,tt11l);
Mul233(&tt12h,&tt12m,&tt12l,xSqh,xSql,t11h,t11m,t11l);
Add33(&t12h,&t12m,&t12l,tbl[0],tbl[1],tbl[2],tt12h,tt12m,tt12l);

Mul123(&xCubeh,&xCubem,&xCubel,x,xSqh,xSql);

Mul33(&tt13h,&tt13m,&tt13l,xCubeh,xCubem,xCubel,t12h,t12m,t12l);
Add133(&t13h,&t13m,&t13l,x,tt13h,tt13m,tt13l);

Renormalize3(&polyh,&polym,&polyl,t13h,t13m,t13l);
*asinh = sign * polyh;
*asinm = sign * polym;
*asinl = sign * polyl;
```

Middle path - interval I_i, $1 \leq i \leq 8$

For the middle path for $x \in I_i$, $1 \leq i \leq 8$, we use the following polynomial approximation by $p_{i\,\text{accurate}}$:

$$\arcsin(x) \approx (asm_{ih} + asm_{il}) + z_i \cdot p_{i\,\text{accurate}}(z_i)$$

where $z_i = x - m_i$ and $asm_{ih} + asm_{il} = \arcsin(m_i) \cdot (1 + \varepsilon)$ with $|\varepsilon| \leq 2^{-121}$ by construction of m_i.

The polynomial $p_{i\,\text{accurate}}$ is of degree 34. In order of decreasing monomial degrees, its coefficients are stored on 20 double precision numbers, 9 double-double precision numbers and 6 triple-double precision numbers. It is completely evaluated using Horner's scheme using the same intermediate precision for the computation as the one used for the coefficients. The last multiplication by z and the addition of $asm_{ih} + asm_{il}$ can be considered as an additional Horner step and is of course performed in triple-double precision.

The double-double precision steps are implemented using the **MulAdd212** macro. The triple-double steps use separate addition and multiplication operators. In particular, **Add33** and **Mul133** come at hand. Overall, two triple-double renormalizations are necessary: one before final rounding can be performed and one other for avoiding a too great overlap in the components of the intermediate triple-double values. See section 10.4.2, page 132 for the overlap bounding.

The coefficients of the polynomial $p_{i\,\text{accurate}}$ and the value $asm_{ih} + asm_{il}$ are read in the main table using the index `i` computated at the interval discrimination phase. The reduced argument z_i has already been computed exactly at the quick phase and can be reused.

The implementation of the accurate phase middle path polynomial approximation reads:

Listing 10.16: Middle path accurate phase polynomial approximation

```
highPoly = tbl[i+39] + z * (tbl[i+40] + z * (tbl[i+41] + z * (tbl[i+42] + z * (
           tbl[i+43] + z * (tbl[i+44] + z * (tbl[i+45] + z * (tbl[i+46] + z * (
           tbl[i+47] + z * (tbl[i+48] + z * (tbl[i+49] + z * (tbl[i+50] + z * (
           tbl[i+51] + z * (tbl[i+52] + z * (tbl[i+53] + z * (tbl[i+54] + z * (
           tbl[i+55] + z * (tbl[i+56] + z * (tbl[i+57] + z * tbl[i+58]))))))))))))))))));


/* Double-double computations */

Mul12(&tt1h,&tt1l,z,highPoly);
Add22(&t1h,&t1l,tbl[i+37],tbl[i+38],tt1h,tt1l);
```

```
12
13 MulAdd212(&t2h,&t2l,tbl[i+35],tbl[i+36],z,t1h,t1l);
14 MulAdd212(&t3h,&t3l,tbl[i+33],tbl[i+34],z,t2h,t2l);
15 MulAdd212(&t4h,&t4l,tbl[i+31],tbl[i+32],z,t3h,t3l);
16 MulAdd212(&t5h,&t5l,tbl[i+29],tbl[i+30],z,t4h,t4l);
17 MulAdd212(&t6h,&t6l,tbl[i+27],tbl[i+28],z,t5h,t5l);
18 MulAdd212(&t7h,&t7l,tbl[i+25],tbl[i+26],z,t6h,t6l);
19 MulAdd212(&t8h,&t8l,tbl[i+23],tbl[i+24],z,t7h,t7l);
20 MulAdd212(&t9h,&t9l,tbl[i+21],tbl[i+22],z,t8h,t8l);
21
22 /* Triple-double computations */
23
24 Mul123(&tt10h,&tt10m,&tt10l,z,t9h,t9l);
25 Add33(&t10h,&t10m,&t10l,tbl[i+18],tbl[i+19],tbl[i+20],tt10h,tt10m,tt10l);
26 Mul133(&tt11h,&tt11m,&tt11l,z,t10h,t10m,t10l);
27 Add33(&t11h,&t11m,&t11l,tbl[i+15],tbl[i+16],tbl[i+17],tt11h,tt11m,tt11l);
28 Mul133(&tt12h,&tt12m,&tt12l,z,t11h,t11m,t11l);
29 Add33(&t12h,&t12m,&t12l,tbl[i+12],tbl[i+13],tbl[i+14],tt12h,tt12m,tt12l);
30 Mul133(&tt13hover,&tt13mover,&tt13lover,z,t12h,t12m,t12l);
31
32 Renormalize3(&tt13h,&tt13m,&tt13l,tt13hover,tt13mover,tt13lover);
33
34 Add33(&t13h,&t13m,&t13l,tbl[i+9],tbl[i+10],tbl[i+11],tt13h,tt13m,tt13l);
35 Mul133(&tt14h,&tt14m,&tt14l,z,t13h,t13m,t13l);
36 Add33(&t14h,&t14m,&t14l,tbl[i+6],tbl[i+7],tbl[i+8],tt14h,tt14m,tt14l);
37 Mul133(&tt15h,&tt15m,&tt15l,z,t14h,t14m,t14l);
38 Add33(&t15h,&t15m,&t15l,tbl[i+3],tbl[i+4],tbl[i+5],tt15h,tt15m,tt15l);
39 Mul133(&tt16h,&tt16m,&tt16l,z,t15h,t15m,t15l);
40 Add233(&t16h,&t16m,&t16l,tbl[i+1],tbl[i+2],tt16h,tt16m,tt16l);
41
42 Renormalize3(&polyh,&polym,&polyl,t16h,t16m,t16l);
43 *asinh = sign * polyh;
44 *asinm = sign * polym;
45 *asinl = sign * polyl;
```

High path - interval I_9

On the high path interval I_9, computing the approximation for arcsin means evaluating the polynomial $p_{9\,\text{accurate}}(z)$ and calculating a triple-double approximation to $\sqrt{2 \cdot z}$ where z is the reduced argument. The function can then be reconstructed as follows:

$$\arcsin(x) \approx p_{9\,\text{accurate}}(z) \cdot \sqrt{2 \cdot z} + (PiHalf_h + PiHalf_m + PiHalf_l)$$

where $PiHalf_h + PiHalf_m + PiHalf_l = \frac{\pi}{2} \cdot (1 + \varepsilon)$, $|\varepsilon| \leq 2^{-164}$

The polynomial $p_{9\,\text{accurate}}(z)$ is of degree 28 with 12 double precision, 9 double-double precision and 8 triple-double precision coefficients. The intermediate precisions used correspond to the precision the coefficients are stored in. The polynomial is completely evaluated in Horner's scheme. Once again, for doing so, the **MulAdd212** macro is used for the double-double steps whilst the triple-double stage is implemented by means of separate additions and multiplications mainly expressed with **Add33** and **Mul133** operators.

The square root extraction yielding to the triple-double approximate $sqrtz_h + sqrtz_m + sqrtz_l = \sqrt{2 \cdot z} \cdot (1 + \varepsilon)$ is implemented using the **sqrt13** macro. Unfortunately, this means recomputing some steps of the square root extraction already computed in the quick phase in the **sqrt12** macro. The choice made is motivated by implementatory reasons.

The multiplication of the polynomial's value and of the square root is performed in triple-double precision by means of the **Mul33** macro. The following addition with the triple-double approximate of $\frac{\pi}{2}$ is implemented using the **Add33** operator.

Overall, three renormalizations are needed. One of them renormalizes the final result, two allow for sufficient overlap bounding. Remark that the square root extraction by **sqrt13** comprises an additional renormalization; see section 2.6.1, page 35 for further details. The overlap bounds will be given in section 10.4.2, page 132.

The code implementing this accurate phase path is the following:

Listing 10.17: High path accurate phase polynomial approximation

```
1 highPoly = tbl[TBLIDX10+42] + z * (tbl[TBLIDX10+43] + z * (tbl[TBLIDX10+44] + z * (
2            tbl[TBLIDX10+45] + z * (tbl[TBLIDX10+46] + z * (tbl[TBLIDX10+47] + z * (
```

```
            tbl[TBLIDX10+48] + z * (tbl[TBLIDX10+49] + z * (tbl[TBLIDX10+50] + z * (
            tbl[TBLIDX10+51] + z * (tbl[TBLIDX10+52] + z *  tbl[TBLIDX10+53]))))))))))));

/* Double-double computations */

Mul12(&tt1h,&tt1l,z,highPoly);
Add22(&t1h,&t1l,tbl[TBLIDX10+40],tbl[TBLIDX10+41],tt1h,tt1l);

MulAdd212(&t2h,&t2l,tbl[TBLIDX10+38],tbl[TBLIDX10+39],z,t1h,t1l);
MulAdd212(&t3h,&t3l,tbl[TBLIDX10+36],tbl[TBLIDX10+37],z,t2h,t2l);
MulAdd212(&t4h,&t4l,tbl[TBLIDX10+34],tbl[TBLIDX10+35],z,t3h,t3l);
MulAdd212(&t5h,&t5l,tbl[TBLIDX10+32],tbl[TBLIDX10+33],z,t4h,t4l);
MulAdd212(&t6h,&t6l,tbl[TBLIDX10+30],tbl[TBLIDX10+31],z,t5h,t5l);
MulAdd212(&t7h,&t7l,tbl[TBLIDX10+28],tbl[TBLIDX10+29],z,t6h,t6l);
MulAdd212(&t8h,&t8l,tbl[TBLIDX10+26],tbl[TBLIDX10+27],z,t7h,t7l);
MulAdd212(&t9h,&t9l,tbl[TBLIDX10+24],tbl[TBLIDX10+25],z,t8h,t8l);

/* Triple-double computations */

Mul123(&tt10h,&tt10m,&tt10l,z,t9h,t9l);
Add33(&t10h,&t10m,&t10l,tbl[TBLIDX10+21],tbl[TBLIDX10+22],tbl[TBLIDX10+23],tt10h,tt10m,tt10l);
Mul133(&tt11h,&tt11m,&tt11l,z,t10h,t10m,t10l);
Add33(&t11h,&t11m,&t11l,tbl[TBLIDX10+18],tbl[TBLIDX10+19],tbl[TBLIDX10+20],tt11h,tt11m,tt11l);
Mul133(&tt12h,&tt12m,&tt12l,z,t11h,t11m,t11l);
Add33(&t12h,&t12m,&t12l,tbl[TBLIDX10+15],tbl[TBLIDX10+16],tbl[TBLIDX10+17],tt12h,tt12m,tt12l);
Mul133(&tt13h,&tt13m,&tt13l,z,t12h,t12m,t12l);
Add33(&t13hover,&t13mover,&t13lover,
                        tbl[TBLIDX10+12],tbl[TBLIDX10+13],tbl[TBLIDX10+14],tt13h,tt13m,tt13l);

Renormalize3(&t13h,&t13m,&t13l,t13hover,t13mover,t13lover);

Mul133(&tt14h,&tt14m,&tt14l,z,t13h,t13m,t13l);
Add33(&t14h,&t14m,&t14l,tbl[TBLIDX10+9],tbl[TBLIDX10+10],tbl[TBLIDX10+11],tt14h,tt14m,tt14l);
Mul133(&tt15h,&tt15m,&tt15l,z,t14h,t14m,t14l);
Add33(&t15h,&t15m,&t15l,tbl[TBLIDX10+6],tbl[TBLIDX10+7],tbl[TBLIDX10+8],tt15h,tt15m,tt15l);
Mul133(&tt16h,&tt16m,&tt16l,z,t15h,t15m,t15l);
Add33(&t16h,&t16m,&t16l,tbl[TBLIDX10+3],tbl[TBLIDX10+4],tbl[TBLIDX10+5],tt16h,tt16m,tt16l);
Mul133(&tt17hover,&tt17mover,&tt17lover,z,t16h,t16m,t16l);

Renormalize3(&tt17h,&tt17m,&tt17l,tt17hover,tt17mover,tt17lover);

Add33(&t17h,&t17m,&t17l,tbl[TBLIDX10+0],tbl[TBLIDX10+1],tbl[TBLIDX10+2],tt17h,tt17m,tt17l);

Mul133(&tt18h,&tt18m,&tt18l,z,t17h,t17m,t17l);
Add133(&polyh,&polym,&polyl,-1,tt18h,tt18m,tt18l);

/* Compute sqrt(2*z) as a triple-double */

twoZ = 2 * z;
sqrt13(&sqrtzh,&sqrtzm,&sqrtzl,twoZ);

/* Multiply p(z) by sqrt(2*z) and add Pi/2 */

Mul33(&pTimesSh,&pTimesSm,&pTimesSl,polyh,polym,polyl,sqrtzh,sqrtzm,sqrtzl);
Add33(&allhover,&allmover,&alllover,PIHALFH,PIHALFM,PIHALFL,pTimesSh,pTimesSm,pTimesSl);

/* Renormalize and multiply by sign */
Renormalize3(&allh,&allm,&alll,allhover,allmover,alllover);
*asinh = sign * allh;
*asinm = sign * allm;
*asinl = sign * alll;
```

10.4 Accuracy bounds

10.4.1 Quick phase accuracy

TODO: see possibly available Gappa files meanwhile

10.4.2 Accurate phase accuracy

TODO: see possibly available Gappa files meanwhile

10.5 Timings and memory consumption

The given implementation of the arcsine uses tables and constants that consume 4740 bytes of memory. This values are fully shared between quick and accurate phase. The code size, including the tables, is about 22 kbytes when compiled to PowerPC machine code. The quick phase code for one rounding mode needs about 1.5 kbytes without the table. This means that there are about 12 kbytes of code for the accurate phase, which is shared between all rounding modes.

Concerning the timing, we compare our implementation to IBM's `libultim` and to MPFR. The values are given in arbitrary units and obtained on a IBM Power 5 processor with gcc 3.3.3 on a Linux Kernel 2.6.5. The timings on other systems are comparable.

Library	avg time	max time
`MPFR`	6322	83415
`crlibm` portable using triple-double	45	300
default `libm` (IBM's `libultim`)	23	206239

It is worth mentioning that IBM's library uses about 20 kbytes of tables for its mere quick phase.

Chapter 11

The arccosine

TODO. This function was completely rewritten using machine-generated polynomials. Gappa proofs were automatically generated, too, and are available in the `gappa/asin` directory. This chapter will be updated soon.

Chapter 12

The arctangent

This chapter is contributed by Nicolas Gast under the supervision of F. de Dinechin.

12.1 Overview

For the arctangent, the quick phase has a precision of 64 bits, and the accurate phase computes to a precision of 136 bits.

Definition interval and exceptional cases

The inverse tangent is defined over all real numbers.

- If $x = NaN$, then $\arctan(x)$ should return NaN
- If $x = \pm\infty$, then $\arctan(x)$ should return $\pm \circ (\pi/2) = \pm \bigtriangledown (\pi/2)$ in rounding to nearest mode. In directed rounding modes, we may return $\pm \bigtriangleup (\pi/2)$ which invalidates the inequation $|\arctan(x)| < \frac{\pi}{2}$ but respects the rounding.
- For $2^{54} < |x| < +\infty$ we choose to return $\pm \circ (\pi/2)$ in all rounding modes.
- For $|x| < 2^{-27}$ we have $|\arctan(x) - x| < 2^{-53}x$ and $|\arctan(x)| < |x|$, which allows to decide to return either x or the FP number next to x towards zero.

12.2 Quick phase

The code of the quick phase is organized in five functions. The function (`atan_quick`) returns two doubles (*atanhi* and *atanlo*) that represent $\arctan(x)$ with a precision of about 64 bits. Four other functions compute the correct rounding : `atan_rn`, `atan_ru`, `atan_rd` and `atan_rz`.

12.2.1 Overview of the algorithm for the quick phase.

This phase is computed in double or double-double. There are two steps in the algorithm: an argument reduction and a polynomial approximation with a polynomial of degree 9.

We compute $\arctan(x)$ as

$$\arctan(x) = \arctan(b_i) + \arctan(\frac{x - b_i}{1 + x.b_i}) \tag{12.1}$$

The b_i are exact doubles and the $\arctan(b_i)$ are stored in double-double.

We define $X_{\text{red}} = \dfrac{x - b_i}{1 + x.b_i}$ for the rest of this chapter.

We tabulate intervals bounds a_i and values b_i such that

$$x \in [a_i; a_{i+1}] \Rightarrow \frac{x - b_i}{1 + x.b_i} < e \quad . \tag{12.2}$$

The i such that $x \in [a_i; a_{i+1}]$ will be found by dichotomy. Therefore we choose a power of two for the number of intervals: 64 intervals ensure $e = 2^{-6.3}$.

Then we use a polynomial of degree 9 for the approximation of $\arctan(X_{\text{red}})$ which ensures 66 bits of precision:

$$\begin{aligned} \arctan(x) &\approx x - \frac{1}{3}.x^3 + \frac{1}{5}.x^5 - \frac{1}{7}.x^7 + \frac{1}{9}.x^9 \\ &\approx x. + x.Q(x^2) \end{aligned}$$

Q is evaluated thanks to a Horner scheme: $Q(z) = z.(-\frac{1}{3} + z.(\frac{1}{5} + z.(-\frac{1}{7} + z.\frac{1}{9})))$ where each operation is computed in double.

As $|z| \leq e$, $Q(z) \leq e^2$

At the end, the reconstruction implements equation (12.3) and (12.1) in double-double arithmetic.

12.2.2 Error analysis on atan_quick

We choose four rounding constant: two when there is a argument reduction, two in the other case. For each case, we use two constants on order to improve performances.

The error analysis presented here is implemented in `maple/atan.mpl`

Notes on b_i, a_i and $\arctan(b_i)$ The b_i and a_i are computed thanks to the `allbi` maple procedure (see `maple/atan.mpl`). There is no approximation error on the b_i since we chose them to be FP numbers. The $\arctan(b_i)$ are stored in double-double so there is an approximation of 2^{-105} on them. The value of e is fixed, then the a_i are also chosen as FP numbers such that inequation (12.2) is true.

Argument reduction

Listing 12.1: Reduction part 1

```
1  if (x > MIN_REDUCTION_NEEDED) /* test if reduction is necessary : */
2  {
3    double xmBIhi,xmBIlo;
4
5      if (x > arctan_table[61][B].d) {
6        i=61;
7        Add12( xmBihi , xmBilo , x , -arctan_table[61][B].d);
8      }
9      else
10       {
11         /* compute i so that a[i] < x < a[i+1] */
12         i=31;
13         if (x < arctan_table[i][A].d) i-= 16;
14         else i+=16;
15         if (x < arctan_table[i][A].d) i-= 8;
16         else i+= 8;
17         if (x < arctan_table[i][A].d) i-= 4;
18         else i+= 4;
19         if (x < arctan_table[i][A].d) i-= 2;
20         else i+= 2;
21         if (x < arctan_table[i][A].d) i-= 1;
22         else i+= 1;
23         if (x < arctan_table[i][A].d) i-= 1
24         xmBihi = x-arctan_table[i][B].d;
25         xmBilo = 0.0;
26       }
```

Lines 1 test if $x > 2^{-6.3}$ and so need to be reduced
Line 5 test if $x > b[61]$ because when $i \in [0;60] : b_i/2 < x < b_i$ (or $x/2 < b_i < x$) and then $x - b_i$ is computed exactly thanks to Sterbenz lemma.
Line 10...21 compute i so that $\frac{x-b_i}{1+x.b_i} < 2^{-6.3}$
Line 7 and 23 compute $xmBIhi + xmBIlo = x - b_i$
We have no rounding error in the computation of $x - b_i$.

Listing 12.2: Reduction part 2

```
1       Mul12(&tmphi,&tmplo, x, arctan_table[i][B].d);
2
3       if (x > 1)
4         Add22(&x0hi,&x0lo,tmphi,tmplo, 1.0,0.0);
5       else {Add22( &x0hi , &x0lo , 1.0,0.0,tmphi,tmplo);}
6
7       Div22( &Xredhi, &Xredlo, xmBihi , xmBilo , x0hi,x0lo);
```

Line 1 compute $x.b_i$ exactly as a double-double
Line 3-5 We need to have a Add22Comp but as we know that $x.b_i > 0$ (so $tmphi > 0$), we test if $tmphi$ is greater than 1 in order to be faster. The Add22 makes an error of $\varepsilon_{Add22} = \varepsilon_{103}$
Line 7 We compute $X_{\text{red}} = \dfrac{x - b_i}{1 + x.b_i}$. The Div22 makes ε_{104} (according to Ziv [40]) error so we have :

$$\begin{aligned}\circ(X_{\text{red}}) &= \frac{(x - b_i).(1 + \varepsilon_{105})}{(1 + x.b_i).(1 + \varepsilon_{105}).(1 + \varepsilon_{105}))}.(1 + \varepsilon_{105}) \\ &= \frac{x - b_i}{1 + x.b_i}.(1 + \varepsilon_{105})(1 + \varepsilon_{105} + \varepsilon_{105} + \varepsilon_{104}) \\ &= X_{\text{red}}.(1 + \varepsilon_{101.9})\end{aligned}$$

So:

$$\varepsilon_{X_{\text{red}}} = \varepsilon_{102.6} \tag{12.3}$$

Polynomial evaluation

The error due to the polynomial approximation is $\delta_{approx} = ||\arctan(x) - x.(1 + q)||^{\infty} = \delta_{72.38}$

Listing 12.3: Polynomial Evaluation

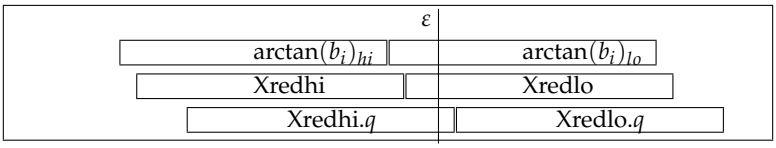
```
1       Xred2 = Xredhi*Xredhi;
2
3       q = Xred2*(coef_poly[3]+Xred2*
4                  (coef_poly[2]+Xred2*
5                   (coef_poly[1]+Xred2*
6                    coef_poly[0]))) ;
```

Line 1 The error between Xred2 and the ideal value of X_{red}^2 comes from
- the error $\varepsilon_{X_{\text{red}}}$ on Xredhi + Xredlo
- the truncation of Xredlo adds ε_{53}
- then the FP multiplication squares this term and adds a rounding error of ε_{53}

which sum up to $\varepsilon_{\text{Xred2}} = ((1 + 2^{-53} + 2^{-105})^2)(1 + 2^{-53}) \approx \varepsilon_{51.4}$
Line 3 Horner approximation with error on Xred2: Maple computes an error around $\varepsilon_{50.7}$ or $\delta_q = \delta_{63.3}$

Reconstruction

The reconstruction adds $\arctan(b_i)$ (read as a double-double from a table) to $\arctan(X_{\text{red}})$ (approximated by $(\text{Xredhi} + \text{Xredlo})(1 + q)$). The terms of this developed product are depicted in the following figure.

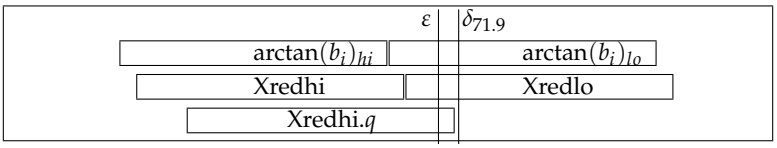


Here the ε bar represents the target accuracy of 2^{-64}. One can see that the term Xredlo.q can be truncated. As Xredlo $<$ Xredhi.2^{-53} and $q < x^2 < 2^{-12.6}$ this entails an error of $\varepsilon_{\text{trunc}} = \varepsilon_{65.6}$ relative to X_{red}, or an absolute error of $\delta_{\text{trunc}} = e.\varepsilon_{\text{trunc}} = e^3.2^{-53} \approx \delta_{71.9}$.

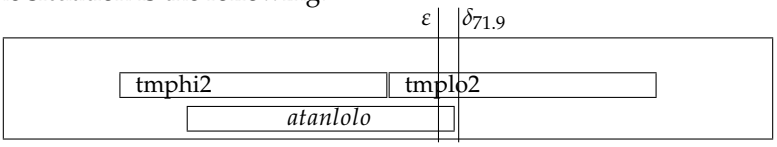
Listing 12.4: Reconstruction

```
1       /* reconstruction : atan(x) = atan(b[i]) + atan(x) */
2       double atanlolo = Xredlo+ arctan_table[i][ATAN_BLO].d + Xredhi*q;
3       double tmphi2, tmplo2;
4       Add12( tmphi2, tmplo2, arctan_table[i][ATAN_BHI].d, Xredhi);
5       Add12( atanhi, atanlo, tmphi2, (tmplo2+atanlolo));
```

Upon entering this code we have:



Before line 5, the situation is the following:



Line 2 The computation of *atanlolo* causes 3 errors :
$\varepsilon_{53}.(\text{Xredlo} + \arctan(b_i)_{lo}) < \delta_{105}$
$\varepsilon_{53}.\text{Xredhi}.q < \delta_{72}$
$\varepsilon_{53}.atanlolo < \delta_{72}$ (again)
Line 4 Add12 adds no error
Line 5 Here we have an FP addition which adds again δ_{72}
Add12 adds no error.

Finally, we get (after accurate computation in Maple)
$\delta_{\text{reconstr}} \approx \delta_{71.8}$

Final error and rounding constant

We have to add all error :

$$\delta_{final} = \delta_{approx} + \delta_q + \delta_{\text{reconstr}} = \delta_{70.2} \tag{12.4}$$

So when $i < 10$, the relative error is $\varepsilon_{63.8}$ that leads to a rounding constant of 1.001.
And when $i > 10$ the relative error is $\varepsilon_{68.24}$ that leads to a rounding constant of 1.000068.

Error when there is no reduction

Listing 12.5: No reduction

```
1
2       x2 = x*x;
3       q = x2*(coef_poly[3]+x2*
4                 (coef_poly[2]+x2*
5                  (coef_poly[1]+x2*
6                   coef_poly[0]))) ;
7       Add12(atanhi,atanlo, x , x*q);
```

The code is very simple so there are few error terms:
Line 1 ε_{53}
Line 2 The Maple procedure to compute Horner approximation gives ε_{51}
Line 3 $\delta_{no\_reduction} = \varepsilon_{105}.x + \varepsilon_q.x^3 + \varepsilon_{x.q}.x^3 + |arctan(x) - x.(1+q)|$
When $x > 2^{-10}$ the relative error is $\varepsilon_{62.9}$. The constant is 1.0024.
When $x < 2^{-10}$ the relative error is $\varepsilon_{70.4}$. The constant is 1.000005.

12.2.3 Exceptional cases and rounding

Rounding to nearest

Listing 12.6: Exceptional cases : rounding to nearest

```
1
2   db_number x_db;
3   x_db.d = x;
4   unsigned int hx = x_db.i[HI] & 0x7FFFFFFF;
5
6   /* Filter cases */
7   if ( hx >= 0x43500000)            /* x >= 2^54 */
8     {
9       if ((hx > 0x7ff00000) || ((hx == 0x7ff00000) && (x_db.i[LO] != 0)))
10        return x+x;                 /* NaN */
11      else
12        return HALFPI.d;            /* \arctan(x) = Pi/2 */
13    }
14  else
15    if ( hx < 0x3E400000 )
16      {return x;                    /* x<2^-27 then \arctan(x) =~ x */}
```

Lines 3 Test if x is greatear than 2^{54}, ∞ or NaN.
Line 5,6 return $\arctan(NaN) = NaN$

Line 8 `HALFPI` is the greatest double smaller than $\frac{\pi}{2}$ in order not to have $\arctan(x) > \frac{pi}{2}$

Line 11 When $x < 2^{-27}$: $x^2 < 2^{-54}$ so $o(\arctan(x)) = x$

Proof

we know that $\arctan(x) = \sum_{i=0}^{\infty} \frac{x^{2i+1}}{2i+1}(-1)^i$.

So:

$$\begin{aligned}
\left|\frac{\arctan(x) - x}{x}\right| &= \left|\frac{\sum_{i=0}^{\infty}\left(\frac{x^{2i+1}}{2i+1}(-1)^i\right) - x}{x}\right| \\
&= \left|\sum_{i=1}^{\infty}\frac{x^{2i}}{2i+1}(-1)^i\right| \\
&< \frac{x^2}{3} \\
&< 2^{-54}
\end{aligned}$$

Then : $\arctan(x) \approx x$

Rounding toward $-\infty$

Listing 12.7: Exceptional cases : rounding down

```
1
2   if ( hx >= 0x43500000)            /* x >= 2^54 */
3     {
4       if ((hx > 0x7ff00000) || ((hx == 0x7ff00000) && (x_db.i[LO] != 0)))
5         return x+x;                 /* NaN */
6       else
7         if (x>0)
8           return HALFPI.d;
9         else
10          return -HALFPI_TO_PLUS_INFINITY.d;          /* atan(x) = Pi/2 */
11    }
12  else
13    if ( hx < 0x3E400000 )
14      {if (sign>0)
15        {if(x==0)
16          {x_db.i[HI]  = 0x80000000;
```

```
17            x_db.i[LO] = 0;}
18          else
19            x_db.l--;
20          return x_db.d;
21          }
22        else
23          return x;
24        }
```

The differences with rounding to nearest mode are for $fracpi2$ for $x < 2^(-27)$.

Listing 12.8: Test for rounding down

```
1   absyh.d=atanhi;
2   absyl.d=atanlo;
3
4   absyh.l = absyh.l & 0x7fffffffffffffffLL;
5   absyl.l = absyl.l & 0x7fffffffffffffffLL;
6   u53.l      = (absyh.l & 0x7ff0000000000000LL) +  0x0010000000000000LL;
7   u.l    = u53.l - 0x0350000000000000LL;
8
9   if(absyl.d > roundcst*u53.d){
10    if(atanlo<0.)
11      {atanhi -= u.d;}
12    return atanhi;
13  }
14  else {
15    return scs_atan_rd(sign*x);
16  }
```

Rounding toward $-\infty$

Listing 12.9: Exceptional cases : rounding up

```
1
2   if ( hx >= 0x43500000)            /* x >= 2^54 */
3     {
4       if ((hx > 0x7ff00000) || ((hx == 0x7ff00000) && (x_db.i[LO] != 0)))
5         return x+x;                 /* NaN */
6       else
7         if (x>0)
8           return HALFPI.d;
9         else
10          return -HALFPI_TO_PLUS_INFINITY.d;          /* atan(x) = Pi/2 */
11    }
12  else
13    if ( hx < 0x3E400000 )
14      {if (sign<0)
15        {x_db.l--;
16        return -x_db.d;
17        }
18      else
19        if(x==0)
20          return 0;
21      return x;
22      }
```

There are the same differences for rounding down.

Rounding to zero

This function is quite simple: it call one of the two function defineded before.

Listing 12.10: Rounding to zero

```
1
2 extern double atan_rz(double x) {
3   if (x>0)
4     return atan_rd(x);
5   else
6     return atan_ru(x);
7 }
```

Test if rounding is possible

This test use the theorem 21. The code is the same than in the theorem except that we have 4 rounding constants :

- 1.0047 when $i < 10$
- 1.000068 when $i \geq 10$
- 1.0024 when $x > 2^{-10}$
- 1.0000132 when $x < 2^{-10}$

12.3 Accurate phase

The accurate phase is the same as the quick phase, except that number are scs and not double.

The intervals are the same as in quick phase. The only difference is that $\arctan(b_i)$ as a third double to improve the precision of $\arctan(b_i)$ to 150 bits. Then we use less memory, that is why we can use the same intervals as in the quick phase.

The polynomial degree is 19 in order to have 136 bits of precision.

$$\arctan(x) \approx x - \frac{1}{3}.x^3 + \frac{1}{5}.x^5 - \frac{1}{7}.x^7 + \frac{1}{9}.x^9 - \frac{1}{11}.x^{11} + \frac{1}{13}.x^{13} - \frac{1}{15}.x^{15} + \frac{1}{17}.x^{17} - \frac{1}{19}.x^{19} \tag{12.5}$$

12.4 Analysis of the performance

12.4.1 Speed

Table 12.1 (produced by the `crlibm_testperf` executable) gives absolute timings for a variety of processors. The test computed 50000 atan in rounding to nearest mode. The second step of `crlibm` was taken 1 time on 50000.

Pentium 4 Xeon / Linux Debian / gcc 2.95			
	min time	max time	avg time
	min time	max time	avg time
`libm`	180	344	231
`mpfr`	417016	3956992	446362
`libultim`	48	257616	189
`crlibm`	36	46428	381
PowerPC G4 / MacOS X / gcc2.95			
	min time	max time	avg time
`libm`	6	11	9
`mpfr`	35291	303019	37022
`libultim`	7	13251	14
`crlibm`	5	1037	19

Table 12.1: Absolute timings for the inverse tangent (arbitrary units)

12.4.2 Memory requirements

Table size is

- for the quick phase, $62 \times (1 + 1 + 2) \times 8 = 1984$ bytes for the 62 a_i, b_i, $\arctan(b_i)$ (hi and lo), plus another 8 bytes for the rounding constant, plus 4×8 for the polynomial, 8 bytes for $\frac{\pi}{2}$ and 64 for the rounding constants or 2096 bytes in total.
- for the accurate phase, we just have 10 SCS constants for the polynomial, and 62 other double for $\arctan(b_i)_{lo_{lo}}$. If we add all : $10 * 11 * 8 + 62 * 8 = 1376$

If we add the fast phase and the acurate one, we have a total of 3472 bytes.

12.5 Conclusion and perspectives

Our arctan is reasonably fast. Libultim is faster but requires ten times more memory (241 polynomials of degree 7 and 241 of degree 16 that represents more than 40KB !). Instead, our argument reduction performs a division, which is an expensive operation.

To improve performances we could inline the code of `atan_quick`, but we prefer to keep it as it is in order to ease the evolution of the algorithm.

The main problem of our arctan is the worst case time which is about 100 times slower than the average time. Thus to improve performances we could try to use double-extended. A double-extended has a mantissa of 64 bits which could transform all double-double operation in double-extended operation. This format number is present in most of the Intel recent processors.

Chapter 13

The trig-of-πx functions

This chapter is contributed by F. de Dinechin.

13.1 Overview

The trigpi functions are defined as follows:

$$\text{sinpi}(x) = sin(\pi x) \tag{13.1}$$
$$\text{cospi}(x) = cos(\pi x) \tag{13.2}$$
$$\text{tanpi}(x) = tan(\pi x) \tag{13.3}$$

These functions are similar to the trigonometric functions, with two main differences:

- Their first argument reduction is exact, and relatively easy: It consists in removing the integer part of x, as e.g. $\text{sinpi}(x+n) = \pm\,\text{sinpi}(x)$ for $n \in \mathbb{N}$. As an important consequence, their worst-case critical accuracy is known on $\mathbb{F}$ as soon as it is known on the interval $[01]$.
- Their Taylor expansion, on the other hand, has irrational coefficients, which requires more careful handling around zero.

Apart from these two differences, we use the same secondary argument reduction as for the trigonometric functions in chapter 9. Indeed, we even use the same tabulated values: first, decompose the input value as follows:

$$\pi x = k\frac{\pi}{256} + \pi y \tag{13.4}$$

where k is an integer and $|y| \leq 1/512$.

Contrary to the usual trig functions, y so defined is an exact double: this second argument reduction is errorless, too. Actually, it is performed in the same operations that compute the first.

Then, denoting $a = k\pi/256$, we read off a table the following triple-double values:

$$sa_h + sa_m + sa_l \approx sin(a)$$
$$ca_h + ca_m + ca_l \approx cos(a)$$

Only 64 pairs of triple-doubles are tabulated (amounting to $64 \times 8 \times 6 = 3$ Kbytes), the rest is obtained by periodicity and symmetry, implemented as masks and integer operations on the integer k. For instance, $a \mod 2\pi$ is implemented by $k \mod 512$, $\pi/2 - a$ is implemented as $128 - k$, etc.

Then we use the reconstruction steps:

$$\text{sinpi}(x) = \sin(a + \pi y) = \cos(a)\,\text{sinpi}(y) + \sin(a)\,\text{cospi}(y) \tag{13.5}$$

$$\text{cospi}(x) = \cos(a + \pi y) = \cos(a)\,\text{cospi}(y) - \sin(a)\,\text{sinpi}(y) \tag{13.6}$$

$$tanpi(x) = \frac{\text{sinpi}(x)}{\text{cospi}(x)} \tag{13.7}$$

13.2 Special cases for $\cos(\pi x)$

cospi should return a NaN on infinities and NaN.

In all the rounding modes, we have $\text{cospi}(x) = 1$ for all the even integer values of $|x|$, and $\text{cospi}(x) = -1$ for all the odd integer values of $|x|$.

In all the rounding modes, we have $\text{cospi}(x) = 1$ for all the even integer values of $|x|$, and $\text{cospi}(x) = -1$ for all the odd integer values of $|x|$. We have $\text{cospi}(x) = +0$ for all the half-integer values of x. One could discuss whether having alternate $+0$ and -0 would not be better, but there will be a conflict between $\cos(\pi + x) = -\cos(x)$ and $\cos(-x) = \cos(x)$ for e.g. $x = 0.5\pi$. Our choice ($+0$ only) is inspired by the LIA2 standard.

Concerning small inputs, we have the Taylor expansion:

$$\cos(\pi x) = 1 - (\pi x)^2/2 + O(x^4) \tag{13.8}$$

where $O(x^4)$ is positive.

Therefore $\cos(\pi x)$ is rounded to 1 in RN and RU mode if $(\pi x)^2 < 2^{-53}$. We test this with a constant C which is defined as the upper 32 bits of $\sqrt{(2^{-53})}/4$.

In RD and RZ modes, we have $\text{cospi}(0) = 1$ and $\text{cospi}(x) = 1 - 2^{-53}$ for $0 < |x| < C$.

13.2.1 Worst case accuracy

The worst case accuracy has been computed only for $x \in [2^{-24}, 1]$ (the previous discussion shows that it is enough), with a worst case accuracy of 2^{-110}.

13.3 Special cases for $\sin(\pi x)$

sinpi should return a NaN on infinities and NaN.

In all the rounding modes, we return $\text{sinpi}(x) = +0$ for all the positive integer values of $|x|$, and $\text{sinpi}(x) = -0$ for all the negative integer values of $|x|$. We have $\text{sinpi}(x) = \pm 1$ for the half-integer values of $|x|$.

For small numbers, the Taylor expansion is

$$\sin(\pi x) = \pi x - (\pi x)^3/6 + O(x^5) = \pi x(1 - (\pi x)^2/6) + O(x^5) \tag{13.9}$$

where $O(x^5)$ has the sign of x.

The situation is therefore more complex than for the radian trigonometrics.

13.3.1 Worst case accuracy

The worst case accuracy has been computed only for $x \in [2^{-57}, 1]$, with a worst case accuracy of 2^{-111}. For smaller arguments, equation (13.9) shows that the worst case arguments will be the same: the worst-cases (up to a certain limit) become those of πx, and may be deduced for each binade of $x < 2^{-55}$.

13.3.2 Subnormal numbers

This is no longer true for subnormals, however, as the relative error becomes an absolute error there. In the subnormal domain, we use the following argument: the worst-case search over the small normal binades show that πx cannot have more than 58 identical ones or zero after the mantissa. Supposing that there exist a worst case in the subnormal binade, it may not have more than 53 + 58 identical bits.

As computing with denormals is tricky – and, on some systems, very slow –, we chose in this case to evaluate πx in SCS, and to use the SCS-to-double functions to manage the subnormal rounding. SCS accuracy (210 bits) is a large overkill considering the 53+58 required bits. If x is a subnormal number, then πx is an approximation to $\text{sinpi}(x)$ accurate to 2^{-2000} according to (13.9).

13.3.3 Computing πx for small arguments

As π is transcendental, we need a two-step approach even for the small arguments. We therefore want to ensure that the bound on the error of approximating $sin(\pi x)$ with $sin(\pi x)$ is between 2^{-60} and 2^{-64}. This bound is given by (13.9): For $x < 2^{-31}$, we have $(\pi x)^2/6 < 2^{-61.28}$. We may then use an algorithm that efficiently computes an approximation to πx with a relative rounding error smaller than 2^{-74}. The total relative error will be smaller than 2^{-61}.

If the rounding test fails, the accurate computation (of sinpi(x), not of $\pi \times x$) has to be launched.

There exists an algorithm, due to Brisebarre and Muller, which computes the correctly rounded value of πx, for any double-precision number x, in two FMA operations. Its proof is a variation of the Kahan/Douglas algorithm mentionned in Chapter 9. Unfortunately, it is of little use here. A first problem is that it requires an FMA, however an equivalent algorithm using double-double arithmetic should be easy to derive. A more important problem is that it is only relevant if one may prove that the correctly rounded value of πx is also the correctly rounded value of $\sin(\pi x)$. This happens when the relative difference between πx and $\sin(\pi x)$ is smaller than the worst-case critical accuracy, which is 2^{-110} for $x < 2^{-31}$. We conclude, again from (13.9) that this algorithm is useful for $x < 2^{-55}$. As we have a two-step approach anyway, the cost of an additional test is difficult to justify.

However, if an FMA is available, we will use the Brisebarre/Muller sequence of two FMAs to evaluate πx using a double-double approximation to π.

In the general case, we will be contented with an approximation to πx accurate to anything much more than 2^{-60}, as suggested before. Let us start with the straightforward double-double multiplication:
`Mul12(&rh,&rl, x,0, PIH, PIL);`
where x is completed with a zero and `PIH` and `PIL` form a double-double approximation to π. This would provide much too much accuracy, so the algorithm is adapted to the specific case as follows:

- In the previous algorithm, all the multiplications by zero are of course optimised out;
- The previous algorithm first splits `x` into `xh` and `xl`, and does the same for `PIH`. An obvious optimisation is to pre-split `PIH` into `PIHH` and `PIHM`.
- A last optimisation is to neglect the term `xl*PIL`.

The final algorithm is therefore :

Listing 13.1: Multiplication by π

```
1  const double c  = 134217729.; /* 2^27 +1 */
2  double t, xh, xl;
3  /* Splitting of x. Both xh and xl have at least 26 consecutive LSB zeroes */
4  t = x*c;
5  xh = (x-t)+t;
6  xl = x-xh;
7
8  Add12(rh,rl, xh*PIHH, (xl*PIHH + xh*PIHM) + (xh*PIL + xl*PIHM) );
```

The splitting is exact (Dekker). In the Add12, all the multiplications are exact except `xh*PIL`. The `Add12` itself is also exact. The error is therefore purely due to the three additions, and lead to a conservative majoration of the relative error of $2^{-53-22} = 2^{-75}$.

13.4 $\tan(\pi x)$

13.4.1 Worst case accuracy

The worst case accuracy has been computed only for $x \in [2^{-25}, 2^{-5}]$, with a worst case accuracy of 2^{-111}.

13.4.2 Special cases

tanpi should return a NaN on infinities and NaN.

In all the rounding modes, we return tanpi$(x) = 0$ with the sign of x for all integer values of $|x|$. We have tanpi$(x) = \pm\infty$ for the half-integer values of $|x|$.

For small numbers, the Taylor expansion is

$$\tan(\pi x) = \pi x + (\pi x)^3/3 + O(x^5) = \pi x(1 + (\pi x)^2/3) + O(x^5) \tag{13.10}$$

where $O(x^5)$ has the sign of x.

The handling of special cases will be similar to those of sinpi. The first step for small arguments now has an overall relative error bounded by 2^{-60}.

13.5 $\arctan(\pi x)$

13.5.1 Proven correctly-rounded domain

The search for worst cases is not finished yet (work in progress). Correct rounding is currently proven on $[\tan(2^{-25}\pi), \tan(2^{-5}\pi)]$.

13.5.2 Implementation

This function is the inverse of tanpi and is thus defined as follows:

$$\text{atanpi}(x) = \frac{1}{\pi}\arctan(x)$$

Its implementation is very simply derived from that of arctan by a final multiplication by an approximation to $1/\pi$.

- In the first step, we use a double-double approximation to $1/\pi$ (accurate to 2^{-105}) and a Dekker double-double multiplication (which should actually be sped up, in the absence of FMA, by pre-splitting the constant as exposed above for sinpi and tanpi). The overall error of this final multiplication is below 2^{-100}, and practically doesn't even change the rounding constants.
- In the second step (currently still SCS) we similarly multiply by an SCS approximation to $1/\pi$ with an error well below 2^{-200} which barely impacts the overall error (2^{-136}, see chapter 12).

Care has to be taken of the special cases, though. To keep things simple, the SCS accurate phase is launched for small arguments to avoid problems with subnormals. This should also be improved.

Implementations of asinpi and acospi along the same lines should follow soon.

Chapter 14

The hyperbolic sine and cosine

This chapter was initially contributed by Matthieu Gallet under the supervision of F. de Dinechin. The accurate phase was rewritten by Ch. Q. Lauter and F. de Dinechin.

14.1 Overview

Like the algorithms for others elementary functions, we will compute our hyperbolic cosine and sine in one or two steps. The first one, which is called 'fast step' is always executed and provides a precision of about 63 bits. This is a sufficient precision in most cases, but sometimes more precision is needed, and then we enter the second step using the SCS library.

14.1.1 Definition interval and exceptional cases

The hyperbolic cosine and the hyperbolic sine are defined for all real numbers, and then for all floating point numbers. These functions are divergent toward $+\infty$ and $-\infty$, so for $|x| > 710.47586007$, $\cosh(x)$ and $\sinh(x)$ should return $+\infty$ or the greatest representable number (depending ot the choosen rounding mode).

- If $x = NaN$, then $\sin(x)$ and $\cosh(x)$ should return NaN
- If $x = +\infty$, then $\cosh(x)$ and $\sinh(x)$ should return $+\infty$.
- If $x = -\infty$, then $\cosh(x)$ should return $+\infty$.
- If $x = -\infty$, then $\sinh(x)$ should return $-\infty$.

This is true in all rounding modes.

Concerning subnormals, $\cosh(x) \geq 1$, so $\cosh(x)$ can't return subnormal numbers, even for subnormal inputs. For small inputs ($|x| \leq 2^{-26}$), we have $\sinh(x) = x$ with 80 bits of precision, so we can return a result without any computation on subnormals.

14.1.2 Relation between $\cosh(x)$, $\sinh(x)$ and e^x

The hyperbolic sine and cosine are defined by

$$\sinh(x) = \frac{e^x - e^{-x}}{2}$$

and

$$\cosh(x) = \frac{e^x + e^{-x}}{2}$$

respectively.

For large arguments, we will be able to neglect the smaller term.

- $e^{-x} < 2^{-65}e^x$ as soon as $x > 23$ (this is the target precision of the first step)
- $e^{-x} < 2^{-115}e^x$ as soon as $x > 40$ (this is the target precision of the second step)

Note that this has been used in the search for worst cases, too: knowing that correct rounding of exp requires at most 2^{-113} accuracy for $x > 2^{-30}$, and the division by 2 being exact, we deduce that the worst cases for sinh and cosh will be those of the exponential for all the input values greater than 40.

However, due to the division by 2, the domain of sinh and cosh is slightly larger than that of exp, so there is a little additional search to do (precisely between 709.78 and 710.75). For the same reason, this additional search gives worst cases for both sinh and cosh.

14.1.3 Worst cases for correct rounding

The search for the worst-case accuracy required for correct rounding the hyperbolic sine and cosine and their inverses is completed. The cosh function require a relative accuracy of 2^{-142} in the worst case, while sinh requires 2^{-126}. However, for $x > 2^{-12}$, both functions require only 2^{-111}.

14.2 Quick phase

14.2.1 Overview of the algorithm

The algorithm consists of two argument reduction using classical formulae of hyperbolic trigonometry, followed by a polynomial evaluation using a Taylor polynom of degree 6 (for cosh) and 7 (for sinh).

These formulaes are:

- $\sinh(x+y) = \sinh(x)\cosh(y) + \sinh(y)\cosh(x)$
- $\cosh(x+y) = \cosh(x)\cosh(y) + \sinh(x)\sinh(y)$
- $\cosh(k\ln(2)) = 2^{k-1} + 2^{-k-1}$
- $\sinh(k\ln(2)) = 2^{k-1} - 2^{-k-1}$

After having treated special cases (NaN, $+\infty$, $-\infty$), we do a first range reduction to reduce the argument between $\frac{-ln(2)}{2}$ and $\frac{ln(2)}{2}$. So, we write $x = k\ln(2) + y$, where k is given by rounding to the nearest integer $x\frac{1}{ln(2)}$. Now, $\frac{-ln(2)}{2} \leq y \leq \frac{ln(2)}{2}$, but it is even too large to have a sufficient precision during polynomial evaluation with small polynoms, and we do a second range reduction, by writing $y = a + b$, where $a = index.2^{-8}$ (index is an integer) and $|b| \leq 2^{-9}$.

Mathematically, we have:

$$\sinh(x) = (2^{k-1} + 2^{-k-1})\sinh(y) + (2^{k-1} - 2^{-k-1})\cosh(y)$$

and

$$\cosh(x) = (2^{k-1} + 2^{-k-1})\cosh(y) + (2^{k-1} - 2^{-k-1})\sinh(y)$$

The second range reduction allows to compute $\sinh(y)$ and $\cosh(y)$ as $\sinh(y) = \sinh(a)\cosh(b) + \sinh(b)\cosh(a)$ and $\cosh(y) = \cosh(a)\cosh(b) + \sinh(a)\sinh(b)$. In the C code, we have $ch\_hi + ch\_lo \approx \cosh(y)$ and $sh\_hi + sh\_lo \approx \sinh(y)$.

A quick computation shows that $-89 \leq index \leq 89$, and we can pre-compute so few values of $\sinh(a)$ and $\cosh(a)$ and store them in a table as double-doubles.

The constants 2^{k-1} and 2^{-k-1} are constructed by working directly on their binary representation.

$\cosh(b)$ and $\sinh(b)$ are computed with Taylor polynoms. It's well-known that

$$\cosh(b) = \sum_{n\geq 0} \frac{x^{2n}}{(2n)!}$$

and

$$\sinh(b) = \sum_{n\geq 0} \frac{x^{2n+1}}{(2n+1)!}$$

For our needs, a degree 6 polynom for cosh and a degree 7 polynom for sinh give enough accuracy.

We write $\cosh(b) = 1 + tcb$ and $\sinh(b) = b(1 + tsb)$, where

$$tcb = b^2(\frac{1}{2} + b^2(\frac{1}{24} + b^2\frac{1}{720}))$$

$$tsb = b^2(\frac{1}{6} + b^2(\frac{1}{120} + b^2\frac{1}{5040}))$$

We use the Horner scheme for the evaluation of the polynoms, with all the coefficients being coded on double-precision numbers.

If the input is very small (i.e. $|b| \leq 2^{-40}$), tsb and tcb are not calculated but directly set to 0, to avoid any problem with subnormal numbers.

At this stage, we have computed all the needed sub-terms before the final reconstruction, which is done in two steps, corresponding to the two-step range-reduction. The reconstruction is computed in double-double arithmetic. In the first reconstruction, some sub-terms can be ignored without any loss of precision, due to their very small relative values. For this step, it exists a particular case, when $index = 0$, since it is the only case where $|\sinh(a)| < 2^{-9}$ ($\sinh(a) = 0$). Now we have the definitive values of $\cosh(y)$ and $\sinh(y)$.

In the second reconstruction, we begin by computing all needed products before adding their results (i.e. $2^{k-1}\cosh(y)$, $2^{k-1}\sinh(y)$,...). Computations are also done using double double arithmetics, with the Add22 function.

14.2.2 Error analysis

Many of the previous computations can introduce some error.

- First range reduction We have to consider two different cases:
  - $|x| \leq \frac{ln(2)}{2}$
    We have $k = 0$, and there is no reduction, and no term of error.
  - $|x| > \frac{ln(2)}{2}$
    We have $k \neq 0$, and we must compute the term of error introduced by the range reduction. Since k is an integer, we can assume that there is no error on it. $ln(2)$ is a constant which is stored in the function in double-double, and we have $ln(2) = ln2_{hi} + ln2_{lo} + \overline{\delta}_{\text{repr\_ln2}}$, where $|\overline{\delta}_{\text{reprln2}}| \leq 1.94e-31$. The total absolute error of this reduction is $\overline{\delta}_{\text{range\_reduc}} = 3.437e-27$, so the maximum relative error is $\overline{\varepsilon}_{\text{range\_reduc}} = 9.9e-27$ (we have $|x| \geq 0.36$), and that represents about 86.38 bits of precision.

- Second range reduction This range reduction is exact (we only cut y in two parts, with no multiplication nor division), so no new term of error is introduced.

- Error in tabulation Since $\cosh(a)$ and $\sinh(a)$ are stored as double-doubles, and since they are transcendental numbers (when $a \neq 0$), some error is done on their approximation. A simple Maple procedure can compute this error, which is about $\overline{\delta}_{\text{ca}} = 6.08e-33$ for cosh and $\overline{\delta}_{\text{sa}} = 1.47e-33$ for sinh. That is large overkill compared to precision on other values.

- Error in polynomial approximations We use the *errlist_quickphase* and *compute_horner_rounding_error* Maple procedures to compute thes errors on tcb and tsb, which are $\overline{\delta}_{\text{rounding\_cosh}} = 6.35e-22$ and $\overline{\delta}_{\text{rounding\_sinh}} = 1.94e-22$. Then there is the approximation error. The sum of theses errors gives $\overline{\delta}_{\text{tcb}} = 6.35e-22$ and $\overline{\delta}_{\text{tsb}} = 1.11e-21$.

- First reconstruction This reconstruction is done by adding all the pre-calculated terms (tcb, tsb, $ca = \cosh(a)$, $sa = \sinh(a)$), in an order which try to minimize the total error.$\overline{\delta}_{\text{sh}} = 2.10e-25$. Maple scripts are used to compute the error, since there are many terms. There are 2 different cases:

- $a = 0$
  $ch_{hi} + ch_{lo} = \widehat{\cosh(\widehat{y})} + \delta_{\cosh 0}$, where $|\delta_{\cosh 0}| \leq \overline{\delta}_{\cosh 0} = 6.35e-22$, and $\sinh(y) = \widehat{\sinh(\widehat{y})} + \delta_{\sinh 0}$, where $|\delta_{\sinh 0}| \leq \overline{\delta}_{\sinh 0} = 5.4e-20$.
- $a \neq 0$
  $ch_{hi} + ch_{lo} = \widehat{\cosh(\widehat{y})} + \delta_{\cosh 1}$, where $|\delta_{\cosh 1}| \leq \overline{\delta}_{\cosh 1} = 2.39e-20$, and $\sinh(y) = \widehat{\sinh(\widehat{y})} + \delta_{\sinh 1}$, where $|\delta_{\sinh 1}| \leq \overline{\delta}_{\sinh 1} = 1.09e-22$.

- Second reconstruction This reconstruction is based on multiplying the obtained results before adding them. The products are exact since each product has a factor which a power of 2. We have to leave absolute errors for relative errors, since the range of values returned by cosh is too large. We will distinguish three different cases:
  - $|k| \leq 35$ All terms must be computed. We have $ch_{hi} + ch_{lo} = \widehat{\cosh(\widehat{x})}(1+\varepsilon_{\text{ch}})$, where $|\varepsilon_{\text{ch}}| \leq \overline{\varepsilon}_{\text{ch}} = 7.66e-19$
  - $k > 35$ In this case, the terms corresponding to e^{-x} are neglected, with an error smaller than 2^{-68}. We have $ch_{hi} + ch_{lo} = \widehat{\cosh(\widehat{x})}(1+\varepsilon_{\text{ch}})$, where $|\varepsilon_{\text{ch}}| \leq \overline{\varepsilon}_{\text{ch}} = 7.69e-19$
  - $k < -35$ This case is symmetric to the previous one, we just have to remplace k by -k.

14.2.3 Details of computer program

The procedures `cosh_quick` and `sinh_quick` contain the computation respectively shared by the functions `cosh_rn`, `cosh_ru`, `cosh_rd` and `cosh_rz` in one hand, and by the functions `sinh_rn`, `sinh_ru`, `sinh_rd` and `sinh_rz` in the other hand. The eight functions `cosh_rX` and `sinh_rX` call `cosh_quick` or `sinh_quick` with an integer which represent the choosen rounding mode. We will begin to prove the cosh function, and then we will prove the sinh function. Since both functions share a lot a code, only the different part between cosh and sinh will be proven for the sinh.

Exceptional cases and argument reduction

This part is shown for `cosh_rn`, but it is quite identical for the three other functions.

Listing 14.1: Exceptional cases

```
1
2  double cosh_rn(double x){
3   db_number y;
4   y.d = x;
5   y.i[HI] = y.i[HI] & 0x7FFFFFFF;      /* to get the absolute value of the input */
6   if (y.d > max_input_ch.d) { /* out of range */
7     y.i[LO] = 0; y.i[HI] = 0x7FF00000; return (y.d);
8   }
9   if ((y.i[HI] & 0x7FF00000) >= (0x7FF00000)) {     /*particular cases : QNaN, SNaN, +- oo*/
10   return (y.d);
11  }
12  return(cosh_quick(x, RN));
13 }
```

Lines 3	Initialize y
Line 4	Get the absolute value of y by removing the first bit.
Line 5	Test if $\cosh(\|x\|) = \cosh(x)$ is representable as a double.
Line 6	If this test is true, we must return ∞.
Line 8	Test if $\|x\|$ is a special case, like NaN or ∞
Line 9	If this test is true, we must return $\|x\|$
Line 11	x is a correct input, we can return cosh_quick.

Procedure cosh_quick

Listing 14.2: Procedure `cosh_quick` - variables

```
1
2  double cosh_quick(double x, int rounding_mode){
3
4   /*some variable declarations */
5   int k;
6   db_number y;
7   double res_hi, res_lo;
8   double ch_hi, ch_lo, sh_hi, sh_lo;/* cosh(x) = (ch_hi + ch_lo)*(cosh(k*ln(2)) + (sh_hi +
        sh_lo)*(sinh(k*ln(2))) */
9   db_number  table_index_float;
10  int table_index;
11  double temp_hi, temp_lo, temp;/* some temporary variables */
12  double b_hi, b_lo,b_ca_hi, b_ca_lo, b_sa_hi, b_sa_lo;
13  double ca_hi, ca_lo, sa_hi, sa_lo; /*will be the tabulated values */
14  double tcb_hi, tsb_hi; /*results of polynomial approximations*/
15  double square_y_hi;
16  double ch_2_pk_hi, ch_2_pk_lo, ch_2_mk_hi, ch_2_mk_lo;
17  double sh_2_pk_hi, sh_2_pk_lo, sh_2_mk_hi, sh_2_mk_lo;
18  db_number two_p_plus_k, two_p_minus_k; /* 2^(k-1) + 2^(-k-1) */
19  db_number absyh, absyl, u53, u;
```

Here there are all the variables which will be used in the code.

First range reduction

Listing 14.3: Procedure `cosh_quick` - first range reduction

```
19
20  /* Now we can do the first range reduction*/
21  DOUBLE2INT(k, x * inv_ln_2.d)
22    if (k != 0){ /* b_hi+b_lo =  x - (ln2_hi + ln2_lo) * k */
23      temp_hi = x - ln2_hi.d * k;
24      temp_lo = -ln2_lo.d * k;
25      Add12Cond(b_hi, b_lo, temp_hi, temp_lo);
26    }
27    else {
28      b_hi = x;  b_lo = 0.;
29    }
```

Line 20 Put in k the closest integer of x * inv_ln_2.
We use the property of DOUBLE2INT that convert a floating-point number in rouding to nearest mode.
By its definition, k satisfies the following properties:
$\lfloor x \times inv\_ln2 \rfloor \leq k \leq \lceil x \times inv\_ln2 \rceil$
$|k| \leq \frac{x}{2} \times inv\_ln2$
since $|x| \leq 710.475...$, we have $|k| \leq 1025$, so k is coded on at most 11 bits.

Line 21 First case : $k \neq 0$
We have by contruction : $ln2_{hi} + ln2_{lo} = \ln(2) + \delta_{\text{repr\_ln2}}$, where $|\delta_{\text{repr\_ln2}}| \leq \overline{\delta}_{\text{repr\_ln2}} = 1.95e-31$.
the last 11 bits of ln_{hi} are set to zero by its construction

Line 22 the $ln2_{hi}k$ product is exact since k is coded on at most 11 bits and the last 11 bits of $ln2_{hi}$ are zeros
we have to use the properties verified by k: $xinv\_ln2 - 1 \leq k \leq xinv\_ln2 + 1$
if $x \geq 0$
we have $k \geq 1$ and then $x \geq \frac{\ln(2)}{2}$, so $(xinv\_ln2 + 1)ln2_{hi} \leq 2x$
since $|k| \leq \frac{x}{2} \times inv\_ln2$, we have $\frac{x}{2} \leq (xinv\_ln2 - 1)ln2_{hi}$
and then we have $\frac{x}{2} \leq kln2_{hi} \leq 2x$
we can apply the Sterbenz theorem to prove that the result of this line is exact
if $x \leq 0$
we can use the same reasoning and then apply the Sterbenz theorem
and this line of code is always exact.

Line 23 this product is not exact, we can loose at most 11 bits of precision
there is an error of δ_{round} which satisfies $|\delta_{\text{round}}| \leq \overline{\delta}_{\text{round}} = 3.15e-30$ on $ln2_{lo}$
so a bound to the maximal absolute error is $k_{max}\overline{\delta}_{\text{round}}$

Line 24 We do an Add12 to have well-aligned double doubles in b_{hi} and b_{lo}
The conditionnal version is used since temp_hi can be zero if x is very close to $kln(2)$.
The total absolute error is bounded by $\overline{\delta}_{\text{b}} = 3.43e-27$

Line 27 We have $k = 0$. We needn't to do any reduction, so $b_{hi} + b_{lo} = x$ exactly.

At this stage, we have $b_{hi} + b_{lo} = \widehat{y} + \delta_{\text{b}}$, where $|\delta_{\text{b}}| \leq \overline{\delta}_{\text{b}} = 3.43e-24$. Now we will write $y = a + b$, where $a = 2^{-8}index$.

Second range reduction

Listing 14.4: Procedure `cosh_quick` - second range reduction

```
29
30  /*we'll construct 2 constants for the last reconstruction */
31  two_p_plus_k.i[LO] = 0;
32  two_p_plus_k.i[HI] = (k-1+1023) << 20;
33  two_p_minus_k.i[LO] = 0;
34  two_p_minus_k.i[HI] = (-k-1+1023) << 20;
35
36  /* at this stage, we've done the first range reduction : we have b_hi + b_lo  between -ln(2)
        /2 and ln(2)/2 */
37  /* now we can do the second range reduction */
38  /* we'll get the 8 leading bits of b_hi */
39  table_index_float.d = b_hi + two_43_44.d;
40  /*this add do the float equivalent of a rotation to the right, since -0.5 <= b_hi <= 0.5*/
41  table_index = LO(table_index_float.d);/* -89 <= table_index <= 89 */
42  table_index_float.d -= two_43_44.d;
43  table_index += bias; /* to have only positive values */
44  b_hi -= table_index_float.d;/* to remove the 8 leading bits*/
45  /* since b_hi was between -2^-1 and 2^1, we now have b_hi between -2^-9 and 2^-9 */
```

Line 30-33 Put in `two_p_plus_k` and `two_p_minus_k` the exact values of 2^{k-1} and 2^{-k-1}.

Line 38-44 The goal of the second range reduction is to write y as $y = index2^{-8} + b$
We have $|y| \leq \frac{ln(2)}{2} \leq \frac{1}{2}$
so $2^{44} \leq 2^{44} + 2^{43} + y \leq 2^{44} + 2^{43} + 2^{42}$
since the mantissa counts 53 bits, only the part above 2^{-8} si kept in table_index_float
It is easy to show that we have $-89 \leq table\_index \leq 89$
so we can add $bias = 89$ to $table\_index$ to have only positive values.
then we remove this bits of y to obtain the final $b = b_{hi} + b_{lo}$
all these operations are exact, so the final absolute error doesn't increase

Polynomial evaluation - First reconstruction

Listing 14.5: Procedure `cosh_quick` - polynomial evaluation - first reconstruction

```
45    y.d = b_hi;
46    /*   first , y   */
47    square_b_hi = b_hi * b_hi;
48    /* effective computation of the polynomial approximation */
49
50    if (((y.i[HI])&(0x7FFFFFFF)) < (two_minus_30.i[HI])) {
51      tcb_hi = 0;
52      tsb_hi = 0;
53    }
54    else {
55      /*   second , cosh(b) = b  * (1/2 + b  * (1/24 + b  * 1/720)) */
56      tcb_hi = (square_b_hi)* (c2.d + square_b_hi * (c4.d + square_b_hi * c6.d));
57      tsb_hi = square_b_hi * (s3.d + square_b_hi * (s5.d + square_b_hi * s7.d));
58    }
59
60
61    if( table_index != bias) {
62      /* we get the tabulated the tabulated values */
63      ca_hi = cosh_sinh_table[table_index][0].d;
64      ca_lo = cosh_sinh_table[table_index][1].d;
65      sa_hi = cosh_sinh_table[table_index][2].d;
66      sa_lo = cosh_sinh_table[table_index][3].d;
67
68      /* first reconstruction of the cosh (corresponding to the second range reduction) */
69      Mul12(&b_sa_hi,&b_sa_lo, sa_hi, b_hi);
70      temp =  ((((((ca_lo + (b_hi * sa_lo)) + b_lo * sa_hi) + b_sa_lo) + (b_sa_hi * tsb_hi)) +
            ca_hi * tcb_hi) + b_sa_hi);
71      Add12Cond(ch_hi, ch_lo, ca_hi, temp);
72        /* first reconstruction for the sinh (corresponding to the second range reduction) */
73    }
74    else {
75      Add12Cond(ch_hi, ch_lo, (double) 1, tcb_hi);
76    }
```

Line 45	Put in y the value of b_{hi}, so we can use its hexadecimal aspect
Line 47	Put b^2 in $square\_b_{hi}$. We have $square\_b_{hi} = \widehat{b} + \delta_{\text{square\_b}}$, where $\lvert\delta_{\text{square\_b}}\rvert \leq \overline{\delta}_{\text{square\_b}} = 4.23e-22$
Line 50	Match b_{hi} and then b with 2^{-40}
Line 51-52	If $\lvert b\rvert \leq 2^{-40}$, we will have $\lvert tcb\rvert, [tsb] \leq \overline{\delta}_{\text{square\_b}}$, so we can directly set tcb and tsb to zero: converning the mathematical values, we have $\lvert\widehat{tcb}\rvert, \lvert\widehat{tsb}\rvert \leq 2^{-24}$. We can avoid by this way any problem with subnormal numbers.
Line 55-56	Polynomial evaluation of $\cosh(x) - 1$ and $\frac{\sinh(x)}{x} - 1$, following the Hrner scheme. A maple procedure is used to compute the error on this computations There are 2 reasons for the total error : the effective computations, since all operations are done with 53 bits of precision. the mathematical approximation, since we use polynoms finally, we have $tcb = \widehat{\cosh(\widehat{b} - 1)} + \delta_{\text{tcb}}$, where $\lvert\delta_{\text{tcb}}\rvert \leq \overline{\delta}_{\text{tcb}} = 6.35e-22$, and $tsb = (\widehat{\frac{\sinh(\widehat{b})}{\widehat{b}} - 1}) + \delta_{\text{tsb}}$, where $\lvert\delta_{\text{tsb}}\rvert \leq \overline{\delta}_{\text{tsb}} = 1.11e-21$
Line 60	If y is very close to 0, we have the 8 bits of the second range reduction which are null
Line 62-65	We get tabulated values for $\cosh(a)$ and $\sinh(a)$. They are tabulated as double doubles: we have $ca_{hi} + ca_{lo} = \widehat{\cosh(\widehat{a})} + \delta_{\text{ca}}$, where $\lvert\delta_{\text{ca}}\rvert \leq \overline{\delta}_{\text{ca}} = 6.08e-33$, and $sa_{hi} + sa_{lo} = \widehat{\sinh(\widehat{a})} + \delta_{\text{sa}}$, where $\lvert\delta_{\text{sa}}\rvert \leq \overline{\delta}_{\text{sa}} = 1.47e-33$,
Line 68	$b\_sa_{hi} + b\_sa_{lo} = sa_{hi}b_{hi}$. This product is exact.
Line 69-70	it is the reconstruction : $\cosh(y) = \cosh(a)(1 + tcb) + \sinh(a)b(1 + tsb)$ A maple procedure is used to compute the error done in this reconstruction. We have $ch_{hi} + ch_{lo} = \widehat{\cosh(\widehat{y})} + \delta_{\text{cosh1}}$, where $\lvert\delta_{\text{cosh1}}\rvert \leq \overline{\delta}_{\text{cosh1}} = 2.39e-20$
Line 75	If y is very close to 0, we have $a = 0$ and $\cosh(y) = \cosh(b) = 1 + tcb$. This addition is exact, so no error is introduced. We have $ch_{hi} + ch_{lo} = \widehat{\cosh(\widehat{y})} + \delta_{\text{cosh0}}$, where $\lvert\delta_{\text{cosh0}}\rvert \leq \overline{\delta}_{\text{cosh0}} = 6.35e-22$

Second reconstruction

Listing 14.6: Procedure `cosh_quick` - reconstruction

```
77    if(k != 0) {
78      if( table_index != bias) {
79        /* first reconstruction for the sinh (corresponding to the second range reduction) */
80        Mul12(&b_ca_hi , &b_ca_lo, ca_hi, b_hi);
81        temp = (((((sa_lo + (b_lo * ca_hi)) + (b_hi * ca_lo)) + b_ca_lo) + (sa_hi*tcb_hi)) + (
              b_ca_hi * tsb_hi));
82        Add12(temp_hi, temp_lo, b_ca_hi, temp);
83        Add22Cond(&sh_hi, &sh_lo, sa_hi, (double) 0, temp_hi, temp_lo);
84      }
85      else {
86        Add12Cond(sh_hi, sh_lo, b_hi, tsb_hi * b_hi + b_lo);
87      }
88      if((k < 35) && (k > -35) )
89        {
90          ch_2_pk_hi = ch_hi * two_p_plus_k.d;
91          ch_2_pk_lo = ch_lo * two_p_plus_k.d;
92          ch_2_mk_hi = ch_hi * two_p_minus_k.d;
93          ch_2_mk_lo = ch_lo * two_p_minus_k.d;
94          sh_2_pk_hi = sh_hi * two_p_plus_k.d;
95          sh_2_pk_lo = sh_lo * two_p_plus_k.d;
96          sh_2_mk_hi = -1 * sh_hi * two_p_minus_k.d;
97          sh_2_mk_lo = -1 * sh_lo * two_p_minus_k.d;
98
99          Add22Cond(&res_hi, &res_lo, ch_2_mk_hi, ch_2_mk_lo, sh_2_mk_hi, sh_2_mk_lo);
100         Add22Cond(&ch_2_mk_hi, &ch_2_mk_lo , sh_2_pk_hi, sh_2_pk_lo, res_hi, res_lo);
101         Add22Cond(&res_hi, &res_lo, ch_2_pk_hi, ch_2_pk_lo, ch_2_mk_hi, ch_2_mk_lo);
102       }
103     else if (k >= 35)
104       {
105         ch_2_pk_hi = ch_hi * two_p_plus_k.d;
106         ch_2_pk_lo = ch_lo * two_p_plus_k.d;
107         sh_2_pk_hi = sh_hi * two_p_plus_k.d;
108         sh_2_pk_lo = sh_lo * two_p_plus_k.d;
109         Add22Cond(&res_hi, &res_lo, ch_2_pk_hi, ch_2_pk_lo, sh_2_pk_hi, sh_2_pk_lo);
110       }
111     else /* if (k <= -35) */
112       {
113         ch_2_mk_hi = ch_hi * two_p_minus_k.d;
114         ch_2_mk_lo = ch_lo * two_p_minus_k.d;
115         sh_2_mk_hi = -1 * sh_hi * two_p_minus_k.d;
116         sh_2_mk_lo = -1 * sh_lo * two_p_minus_k.d;
117         Add22Cond(&res_hi, &res_lo, ch_2_mk_hi, ch_2_mk_lo, sh_2_mk_hi, sh_2_mk_lo);
118       }
119   }
120   else {
121     res_hi = ch_hi;
122     res_lo = ch_lo;
123   }
```

Line 77	Test if $k = 0$ or not
Line 78-87	We have $k \neq 0$, so we must compute $\sinh(y)$ This computation is done like the computation of $\cosh(h)$ We can use an Add12 (instead of Add12Cond) since $b_{hi}ca_{hi} \geq temp$ A maple script gives $\sinh(y) = \widehat{\sinh(\widehat{y})} + \delta_{\text{sinh1}}$, where $\lvert\delta_{\text{sinh1}}\rvert \leq \overline{\delta}_{\text{sinh1}} = 1.09e-22$ and $\lvert\delta_{\text{sinh1}}\rvert \leq \overline{\delta}_{\text{sinh0}} = 5.4e-20$ (when $\sinh(a) = 0$)
Line 89	We have $k \neq 0$, and $\lvert k\rvert \leq 35$
Line 91-98	we multiply $\sinh(y)$ and $\cosh(y)$ by powers of 2, so these products are exact
Line 100-102	A maple script is used to compute the error: We have $ch_{hi} + ch_{lo} = \widehat{\cosh(\widehat{x})}(1 + \varepsilon_{\text{ch}})$, where $\lvert\varepsilon_{\text{ch}}\rvert \leq \overline{\varepsilon}_{\text{ch}} = 7.66e-19$
Line 104	$k \geq 35$
Line 106-109	we multiply $\sinh(y)$ and $\cosh(y)$ by powers of 2, so these products are exact Some terms are not computed, since they are too little
Line 110	A maple script is used to compute the error: We have $ch_{hi} + ch_{lo} = \widehat{\cosh(\widehat{x})}(1 + \varepsilon_{\text{ch}})$, where $\lvert\varepsilon_{\text{ch}}\rvert \leq \overline{\varepsilon}_{\text{ch}} = 7.69e-19$
Line 112	$k \leq -35$
Line 114-118	this case is symmetric to the previous one. We also have $ch_{hi} + ch_{lo} = \widehat{\cosh(\widehat{x})}(1 + \varepsilon_{\text{ch}})$, where $\lvert\varepsilon_{\text{ch}}\rvert \leq \overline{\varepsilon}_{\text{ch}} = 7.69e-19$
Line 121	we now have $k = 0$ Since we have $1 \leq \cosh(x)$, we have $\overline{\varepsilon}_{\text{ch}} \leq max(\overline{\delta}_{\text{cosh0}}, \overline{\delta}_{\text{cosh1}}) = 2.39e-20$

At this, stage, we have $ch_{hi} + ch_{lo} = \widehat{\cosh(\widehat{x})}(1 + \varepsilon_{\text{ch}})$, where $\lvert\varepsilon_{\text{ch}}\rvert \leq \overline{\varepsilon}_{\text{ch}} = 7.69e-19 = 2^{-60.17}$.

14.2.4 Rounding

Rounding to the nearest

The code for rounding is strictly identical to that of Theorem 21. The condition to this theorem that `res_hi` $\geq 2^{-1022+53}$ is ensured by the image domain of the cosh, since $\cosh(x) \geq 1$. The rounding constant

14.2.5 Directed rounding

Here again, the code is strictly identical to that of Theorem 22, and the conditions to this theorem are ensured by the image domain of the cosh.

14.3 Accurate phase

It is reminded that correct rounding requires an intermediate accuracy of 2^{-142} for cosh and 2^{-126} for sinh. The triple-double exponential function in `crlibm` is sufficiently accurate for computing the cosh, using the equation

$$\cosh(x) = \frac{e^x + e^{-x}}{2} \quad .$$

For sinh, we use the `expm1` triple-double implementation: This is more accurate around 0, as the following equation shows:

$$\sinh(x) = \frac{e^x - e^{-x}}{2} \quad = \frac{(e^x - 1) - (e^{-x} - 1)}{2} \quad .$$

As already noted, the e^{-x} term is optimised out for large arguments. Indeed, for $\lvert x\rvert > 40$ we have $e^{-x} < 2^{-115}e^x$, therefore the relative error due to neglecting e^{-x} is much smaller than the worst case accuracy required to decide rounding, which is smaller than 2^{-111} for both functions in this range.

14.4 Analysis of cosh performance

The input numbers for the performance tests given here are random positive double-precision numbers with a normal distribution on the exponents. More precisely, we take random 63-bit integers and cast them into double-precision numbers.

In average, the second step is taken in 0.13% of the calls.

14.4.1 Speed

Table 14.1 (produced by the `crlibm_testperf` executable) gives absolute timings for a variety of processors and operating systems.

Pentium III / Linux 2.6 / gcc 4.0			
	min time	avg time	max time
default `libm`	242	272	304
`crlibm`	212	344	2639

Table 14.1: Absolute timings for the hyperbolic cosine

Contributions to this table for new processors/OS/compiler combinations are welcome.

Chapter 15

The power function

This chapter is contributed by Ch. Q. Lauter and F. de Dinechin.

15.1 Work in progress

Here is the status of the current implementation of pow in CRLibm:

- Exact and mid-point cases are handled properly [24]. This is especially important because an exact case would mean an infinite Ziv iteration.
- Worst cases are not known for the full input range. This is a deep theoretical issue. Recent research has focussed on x^n for integer n. At the time of release, it has been proved computationally that that an intermediate precision of 2^{-118} is enough to round correctly x^n for all integer n between -180 and +1338. In addition, specific algorithms have been studied for the computation of x^n [23].
- Due to lack of time, only round-to-nearest is implemented. Directed rounding requires additional work, in particular in subnormal handling and in exact case management. There are more exact cases in directed rounding modes, therefore the performance should also be inferior.
- The current implementation computes two Ziv iterations, to 2^{-61} then to 2^{-120}. With current technology, there is little hope to find all the worst cases for the full range of the power function. Should an input require more than 2^{-120} happen (to our knowledge none has been exhibited so far), current implementation will not necessarily return the correctly rounded result. Options are:
  - Ignore silently the problem (this is the current option).
  - perform a second rounding test at the end of the accurate step. If the test fails (with a probability smaller that 2^{-120}),
    * an arbitrary precision computation could be launched, for example MPFR. This requires adding a dependency to MPFR only for this highly improbable case.
    * launch a high, but not arbitrary precision third step (say, accurate to 2^{-3000}. Variations of the SLZ algorithm [36] could provide, at an acceptable computational cost, a certificate that there is no worst case requiring a larger precision. This is the only fully satisfactory solution that seems at reach, but this idea remains to be explored.
    * (in addition to the previous) a message on the standard error could be written, including the corresponding inputs, and inviting anyone who reads it to send us a mail. Considering the probability, we might wait several centuries before getting the first mail.

Bibliography

[1] Open source from Intel. `http://www.intel.com/software/products/opensource/`.

[2] SCS, Software Carry-Save multiprecision library.

[3] Sun Freely Distributable LIBM. `http://www.netlib.org/fdlibm/`.

[4] AMD. AMD athlon processor x86 code optimization guide. Technical report, Advanced Micro Devices,inc, 2001.

[5] ANSI/IEEE. Standard 754-1985 for binary floating-point arithmetic, 1985.

[6] A. Baker. *Transcendental Number Theory*. Cambridge University Press, 1975.

[7] S. Boldo and M. Daumas. A mechanically validated technique for extending the available precision. In *35th Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, California, 2001. IEEE Computer Society Press.

[8] M. Cornea, J. Harrison, and P.T.P Tang. *Scientific Computing on Itanium-based Systems*. Intel Press, 2002.

[9] Marc Daumas and Claire Moreau-Finot. Exponential: implementation trade-offs for hundred bit precision. In *Real Numbers and Computers*, pages 61–74, Dagstuhl, Germany, 2000.

[10] F. de Dinechin and D. Defour. Software carry-save: A case study for instruction-level parallelism. In *Seventh International Conference on Parallel Computing Technologies*, Nizhny Novgorod, Russia, September 2003.

[11] F. de Dinechin, D. Defour, and C. Lauter. Fast correct rounding of elementary functions in double precision using double-extended arithmetic. Technical Report 2004-10, LIP, École Normale Supérieure de Lyon, March 2004. Available at http://www.ens-lyon.fr/LIP/Pub/Rapports/RR/RR2004/RR2004-10.pdf.

[12] F. de Dinechin, A. Ershov, and N. Gast. Towards the post-ultimate libm. In *17th Symposium on Computer Arithmetic*, pages 288–295. IEEE Computer Society Press, June 2005.

[13] F. de Dinechin, Ch. Q. Lauter, and G. Melquiond. Assisted verification of elementary functions using Gappa. In *ACM Symposium on Applied Computing*, 2006. Extended version available as LIP research report RR2005-43, `http://www.ens-lyon.fr/LIP/Pub/Rapports/RR/RR2005/RR2005-43.pdf`.

[14] D. Defour and F. de Dinechin. Software carry-save for fast multiple-precision algorithms. In *35th International Congress of Mathematical Software*, Beijing, China, 2002. Updated version of LIP research report 2002-08.

[15] D. Defour, G. Hanrot, V. Lefèvre, J.-M. Muller, N. Revol, and P. Zimmermann. Proposal for a standardization of mathematical function implementations in floating-point arithmetic. *Numerical algorithms*, 37(1-4):367–375, January 2004.

[16] Theodorus J. Dekker. A floating point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.

[17] P. M. Farmwald. High bandwidth evaluation of elementary functions. In K. S. Trivedi and D. E. Atkins, editors, *Proceedings of the 5th IEEE Symposium on Computer Arithmetic*. IEEE Computer Society Press, Los Alamitos, CA, 1981.

[18] S. Gal. Computing elementary functions: A new approach for achieving high accuracy and good performance. In *Accurate Scientific Computations, LNCS 235*, pages 1–16. Springer Verlag, 1986.

[19] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–47, March 1991.

[20] J. Harrison, T. Kubaska, S. Story, and P.T.P. Tang. The computation of transcendental functions on the IA-64 architecture. *Intel Technology Journal*, Q4, 1999.

[21] R. Klatte, U. Kulisch, C. Lawo, M. Rauch, and A. Wiethoff. *C-XSC a C++ class library for extended scientific computing*. Springer Verlag, 1993.

[22] D. Knuth. *The Art of Computer Programming*, volume 2. Addison Wesley, Reading, MA, 1973.

[23] P. Kornerup, Ch. Lauter, V. Lefèvre, N. Louvet, and J.-M. Muller. Computing correctly rounded integer powers in floating-point arithmetic. Technical Report RR 2008-15, SDU, Odense, Denmark et LIP, CNRS/ENS Lyon/INRIA/Universit de Lyon, Lyon, France, May 2008. To appear in ACM Transactions on Mathematical Software.

[24] Ch. Lauter and V. Lefèvre. An efficient rounding boundary test for pow(x,y) in double precision. Technical Report RR-2007-36, Laboratoire de l'Informatique du Parallélisme, 2007. To appear in IEEE Transactions on Computers.

[25] Ch. Q. Lauter. Basic building blocks for a triple-double intermediate format. Technical Report RR2005-38, LIP, September 2005.

[26] V. Lefèvre, J.M. Muller, and A. Tisserand. Towards correctly rounded transcendentals. *IEEE Transactions on Computers*, 47(11):1235–1243, November 1998.

[27] V. Lefvre. *Moyens arithmétiques pour un calcul fiable*. PhD thesis, École Normale Supérieure de Lyon, Lyon, France, 2000.

[28] R.-C. Li, P. Markstein, J. P. Okada, and J. W. Thomas. The libm library and floating-point arithmetic for HP-UX on Itanium. Technical report, Hewlett-Packard company, april 2001.

[29] IBM Accurate Portable Math. Library. `http://oss.software.ibm.com/mathlib/`.

[30] P. Markstein. *IA-64 and Elementary Functions : Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, 2000. ISBN: 0130183482.

[31] G. Melquiond. Gappa - génération automatique de preuves de propriétés arithmétiques. Available at `http://lipforge.ens-lyon.fr/www/gappa/`.

[32] R.E. Moore. *Interval analysis*. Prentice Hall, 1966.

[33] MPFR. `http://www.mpfr.org/`.

[34] J.-M. Muller. *Elementary Functions, Algorithms and Implementation*. Birkhauser, Boston, 1997.

[35] K. C. Ng. Argument reduction for huge arguments: Good to the last bit. *SunPro*, July 1992. Work in progress.

[36] Damien Stehlé. *Algorithmique de la rduction de rseaux et application la recherche de pires cas pour l'arrondi de fonctions mathmatiques*. PhD thesis, LORIA, 2006.

[37] P. H. Sterbenz. *Floating point computation*. Prentice-Hall, Englewood Cliffs, NJ, 1974.

[38] P. T. P. Tang. Table lookup algorithms for elementary functions and their error analysis. In P. Kornerup and D. W. Matula, editors, *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 232–236, Grenoble, France, June 1991. IEEE Computer Society Press, Los Alamitos, CA.

[39] W. F. Wong and E. Goto. Fast hardware-based algorithms for elementary function computations using rectangular multipliers. *IEEE Transactions on Computers*, 43(3):278–294, March 1994.

[40] A. Ziv. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Transactions on Mathematical Software*, 17(3):410–423, September 1991.