Guaranteed Precision for Transcendental and Algebraic Computation made Easy

by

Zilin Du

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

Courant Institute of Mathematical Sciences

New York University

May 2006

Chee Keng Yap

Dedicated to the friends and families who blessed and supported me

Acknowledgments

Many thanks to my advisor Professor Chee Keng Yap for all of his kind help with my work. He has always been available and enthusiastic to encourage and help me during the last five years. I'm also very thankful to Professor Michael Overton, Richard Pollack and Hervé Brönnimann for their careful reading and generous advice. Special thanks to my collaborator and thesis committee member, Sylvain Pion. He kindly hosted me one fruitful week in February 2006 in INRIA. Also great thanks to Vikram Sharma who worked with me together for our Exact Geometric Computation project.

My graduate school years in New York city cannot be a truly memorable one without all my friends here. Among them are Zhihua Wang, Ziyang Wang, Jinyuan Li, Xiaojian Zhao, Ning Feng, Wei Cheng, Tao Ling, Jiawu Feng, Xiujie Wang, Bing Sun and Shubin Zhao. I also want to thank brothers and sisters from Newtown Church, Chingching Won, Daniel Hu, Davie Chao, Lanjo Wu, Li Jing, Lydia Yang, and Ming for their love and prayers.

Abstract

Numerical non-robustness is a well-known phenomenon when implementing geometric algorithms. A general approach to achieve geometric robustness is Exact Geometric Computation (EGC). This dissertation explores the redesign and extension of the `Core Library`, a C++ library which embraces the EGC approach. The philosophy of the `Core Library` is to make guaranteed precision computation transparent and easily accessible to most users; our redesign strives to keep the original user interface intact while improving the underlying algorithms significantly. The contributions of this thesis are organized into three parts.

In the first part, we discuss the redesign of the `Core Library`, especially the expression (`Expr`) and bigfloat (`BigFloat`) classes. Our new design emphasizes extensibility in a clean and modular way. The three facilities in `Expr`, filter, root bound and bigfloat, are separated into independent modules. This allows new filters, root bounds and some bigfloat substitute to be plugged in. The key approximate evaluation and precision propagation algorithms have been greatly improved. A new bigfloat system based on `MPFR` and interval arithmetic has been incorporated. Our benchmark shows that the redesigned `Core Library` typically has 5-10 times speedup. We also provide tools to facilitate extensions of `Expr` to incorporate new types of nodes, especially transcendental nodes.

Although the `Core Library` was originally designed for algebraic applica-

tions, transcendental functions are needed in many applications. In the second part, we present a complete algorithm for absolute approximation of the general hypergeometric function. Its complexity is also given. The extension of this algorithm to "blackbox numbers" is provided. A general hypergeometric function package based on our algorithm is implemented and integrated into the `Core Library` based on our new design.

Brent has shown that many elementary functions, such as $\exp, \log, \sin$, etc., can be efficiently computed using the Arithmetic-Geometric Mean (AGM) based algorithm. However, he only gave an asymptotic error analysis. The constants in the Big $O(\cdot)$ notation required for implementation are unknown. We provide a non-asymptotic error analysis of the AGM algorithm and the related algorithms for logarithm and exponential functions. These algorithms have been implemented and incorporated into the `Core Library`.

Contents

List of Figures

List of Tables

List of Experiments

List of Appendices

Chapter 1

Introduction

Numbers are normally represented in fixed-point or floating-point format using a fixed number of bits in modern computers. The arithmetic operations and comparisons associated with them are inexact. Numerical errors due to round-off may lead to inconsistent states from which computer programs cannot recover. These non-robustness problems are especially serious in geometric computation since numerical errors can propagate into the combinatorial computations and result in complete failure of the algorithms. A general framework to solve such problems is Exact Geometric Computation (EGC). While current EGC approach can handle algebraic problems successfully, it is an open question whether it is possible for problems involving transcendental functions. This thesis is aimed to redesign our EGC software library, `Core Library`, to make it more efficient, modular and extensible, easy to use for both algebraic and transcendental computation. A complete algorithm for absolute approximating the general hypergeometric function is given. An implementation has been developed and integrated into the redesigned `Core Library`.

1.1 Exact Geometric Computation (EGC)

Computational Geometry is the study of algorithms to solve geometric problems ([77, 25, 71, 6, 66]). In general, geometric computations have two components, a numerical part and combinatorial part. Numerical computations are involved in both the construction of new geometric objects and the evaluation of geometric predicates. To determine the combinatorial relations among geometric objects, geometric predicates are especially critical. For example, for the convex hull problem the predicate asserting that 3 points are oriented counterclockwise is very important. Incorrect evaluation of this predicate can lead to inconsistencies and crash the whole program.

Usually geometric algorithms are designed within a *Real RAM model of computation*, in which real numbers are exact and arithmetic operations and comparisons are performed exactly. However, approximate floating-point arithmetic, which is a form of fixed-precision arithmetic, is widely used in modern computer systems. Almost all of them follow the IEEE 754 standard [44]. Such an arithmetic has serious shortcomings [36, 43] in certain applications. A geometric algorithm implemented using straightforward floating-point arithmetic could easily introduce some undesirable numerical errors. Although numerical errors can sometimes be tolerated and interpreted as small perturbations in inputs, serious problems arise when these errors accumulate and propagate, triggering inconsistencies between numerical and combinatorial data. Numerical non-robustness has long been an important concern in the implementation of geometric algorithms ([26, 43, 83, 5, 27]). In the last decade it has become a central research topic in computational geometry ([4, 52, 42]).

There are many approaches to dealing with this problem - see [73, 84, 95]

for general surveys on robust geometric computation. For example, the "naive" arithmetic solution tries to compute every numerical quantity exactly, without any errors. But researchers soon realized that it is impractical, even for computation over linear geometry [97, 47]. One approach called *Exact Geometric Computation* (EGC, for short) [92] perhaps has been the most successful one. The basic idea of EGC is to compute "exactly" in the geometric sense, i.e., all predicates in a geometric algorithm are evaluated exactly. The EGC principle does not require "exact arithmetic" but arithmetic that has "sufficient accuracy" to guarantee the exactness of geometric predicates. Moreover, the EGC solution is often more general and has better properties. Unlike many other approaches [13, 24] that require new algorithms to be designed for each application, it can take any implementation of any algorithm and achieve full robustness just by replacing its machine number type by a suitable number type ("EGC number").

Two general libraries which provide such EGC numbers are currently available: `LEDA Real`[14, 59] and `Core Library`[46, 21]. The `Core Library`, as the successor of `Real/Expr`[96], is aimed at achieving a user-friendly interface, especially for user's access of numerical accuracy [93]. The EGC number in the `Core Library` is called `Expr`; it is the first EGC number type to incorporate arbitrary real algebraic numbers. In contrast to the `Core Library`, `LEDA` offers, in addition to their EGC number types, also a large collection of algorithms, data structures and related services. The EGC number type in `LEDA` is called `LEDA_real`. Using such libraries, programmers can routinely implement robust programs by using standard algorithms. A large collection of such robust algorithms have been implemented in the major software libraries `CGAL`[19] and `LEDA`[49].

§1. General Terminology and Notation for this Thesis We list some terminology that are used in this thesis:

- $\lg = \log_2$, $\ln = \log_e$.
- $\oplus$, $\ominus$, $\odot$ and $\oslash$ are used to denote the corresponding floating-point arithmetic operations of addition, subtraction, multiplication and division.
- x is a "blackbox number" if we can get arbitrarily many bits of precision of x.
- $\mathbf{a} = \{a_1, a_2, \ldots, a_p\}$ is a multiset if the order of $\mathbf{a}$ is ignored, but the multiplicity is explicitly significant. For example, multisets $\{1, 2, 3\}$ and $\{2, 1, 3\}$ are equivalent, but $\{1, 2, 3\}$ and $\{1, 1, 2, 3\}$ differ.

1.2 Previous work in EGC

Since the early 1990s, many research efforts have been made toward EGC. Among them three key areas are constructive root bounds [16, 54, 75, 85], approximate expression evaluation [86, 52, 53] and filter techniques [28, 31, 12].

§2. Constructive Root Bound Let Ω be a set of real or complex functions (including constants). The set of constant expressions over Ω is denoted by $\texttt{Expr}(\Omega)$. For instance, $\Omega_0 = \{+, -, \times\} \cup \mathbb{Z}$. The value of an expression $e \in \texttt{Expr}(\Omega)$, denoted by $\texttt{Val}_\Omega(e)$, is a real (complex) number, but it may also be undefined. The zero problem for Ω, denoted $\texttt{ZERO}(\Omega)$, is to decide for a given $e \in \texttt{Expr}(\Omega)$, whether $\texttt{Val}_\Omega(e)$ is defined and equal to 0. It is called the **fundamental problem** of EGC.

DEFINITION 1. *Assume e is an expression and b is a positive number in $\mathbb{R}$. We say b is a* ***root bound*** *(or zero bound) if the following holds:*

$$e \text{ is well-defined and } e \neq 0 \Rightarrow |e| \geq b.$$

Note that b is a conditional bound: it is *not* a bound when e is undefined or zero. To determine the sign of e from a root bound b, we compute a numerical approximation $\tilde{e}$ such that if e is undefined then $\tilde{e}$ is undefined; otherwise, $|e - \tilde{e}| < \frac{b}{2}$. Then

$$sign(e) = \begin{cases} sign(\tilde{e}) & \text{if } |\tilde{e}| \geq \frac{b}{2} \text{ or } \tilde{e} \text{ is undefined} \\ 0 & \text{otherwise} \end{cases}$$

It is important to find root bounds that are as large as possible since the amplitude of the root bound directly affects the worse-case complexity in sign determination. On the other hand, we also want to be able to compute them efficiently.

Root bounds have been extensively studied in the classical literature (e.g., [58] or [61]). Many classical results, however, are non-constructive. We are more interested in root bounds which can be inductively computed from the structure of an algebraic expression. Such bounds are called "Constructive Root Bounds". A number of constructive root bounds have been proposed and implemented, such as Degree-Measure bounds, BFMSS bounds [16] and Li-Yap bounds [54]. Some techniques (e.g. [75]) have been developed to improve them as well.

Root bounds greatly depends on the operators in Ω. A hierarchy of some Ω that are important in practice is described in [94]:

- Polynomial basis: $\Omega_0 = \{+, -, \times\} \cup \mathbb{Z}$
- Rational basis: $\Omega_1 = \Omega_0 \cup \{\div\}$

- Radical basis: $\Omega_2 = \Omega_1 \cup \{\sqrt{\cdot}\}$
- Algebraic basis: $\Omega_3 = \Omega_2 \cup \{\texttt{RootOf}(P, i)\}$ where P is an integer polynomial and i is the i-th real root of P.
- Elementary basis: $\Omega_4 = \Omega_3 \cup \{\exp(\cdot), \log(\cdot)\}$
- Hypergeometric basis: $\Omega_5 = \Omega_4 \cup \mathbb{H}$, where $\mathbb{H}$ is the set of real hypergeometric functions (see Chapter 3).

In many areas of computational sciences, non-algebraic operators are needed: Ω_4 is the simplest basis beyond the algebraic case. Basic functions such as the trigonometric functions are captured in Ω_5. It is an open problem whether there exist constructive root bounds for Ω_4 or Ω_5, but Richardson has given an important conditional result in [80, 81, 82]; partial results are also known from transcendental number theory [76, 57, 69].

§3. Diamond Operation In Ω_3, the `RootOf(P,i)` operation allows one to take the root of the integer polynomials. A more general operation which is called **Diamnod Operation** is to allow one to take the root of arbitrary real algebraic polynomials. If $e_d, e_{d-1}, \ldots, e_1, e_0$ are real algebraic expressions and i is a positive integer with $0 \leq i \leq d$, then $\diamond(i, e_d, e_{d-1}, \ldots, e_1, e_0)$ is an expression. If $\texttt{Val}_\Omega(e_i)$ are defined, then the value of $\diamond(i, e_d, e_{d-1}, \ldots, e_1, e_0)$ is the i-th smallest real root of the polynomial

$$\texttt{Val}_\Omega(e_d)X^d + \texttt{Val}_\Omega(e_{d-1})X^{d-1} + \cdots + \texttt{Val}_\Omega(e_0)$$

if the polynomial has at least i real roots. Otherwise, the value is undefined. We also say $\diamond(\cdot)$ is a **Diamond Operator**.

§4. Approximate expression evaluation Essentially, all expression evaluations are done with approximations in any EGC library. This requires algorithms that can compute an approximation $\tilde{e}$ to within precision p for any given expression e and precision p. Mostly we use relative or absolute precision [94]:

DEFINITION 2. *Given $e, \tilde{e} \in \mathbb{R}$, we say that $\tilde{e}$ is a relative p-bit approximation of e if*

$$|\tilde{e} - e| \leq |e| \cdot 2^{-p}.$$

We also say $\tilde{e}$ has relative precision p.

DEFINITION 3. *Given $e, \tilde{e} \in \mathbb{R}$, we say that $\tilde{e}$ is an absolute p-bit approximation of e if*

$$|\tilde{e} - e| \leq 2^{-p}.$$

We also say $\tilde{e}$ has absolute precision p.

The `Core Library` use a **precision-driven approach** to approximate expressions. It was first given in detail and analyzed in [96], then was implemented in the `Real/Expr` package [72] and incorporated into the `Core Library` subsequently. Intuitively, it can be viewed as an iterative "downward-upward process" operating on the input expression DAG (directed acyclic graph). In the downward direction, precision values (starting with p at the root) are propagated down to the leaves. In the upward direction, approximations are propagated up to the root. This upward propagation amounts to a bottom-up evaluation of the expression e.

The optimal method of propagating precision is an open problem. Currently the `Core Library` propagates a "composite precision", which leads to compute

an approximation $\tilde{e}$ of e such that

$$|\tilde{e} - e| \leq 2^{-a} \quad \text{or} \quad |\tilde{e} - e| \leq |e|2^{-r}$$

for a given pair $[a, r]$. In [94], Yap proposed a simpler and more intuitive algorithm, which propagates either absolute or relative precision, but not both.

We should notice that computing absolute approximations is a more basic problem. Yap [94] showed that computing a relative approximation of e can be converted to compute an absolute approximation of e only if we can compute a root bound for e. In terms of our hierarchy $\{\Omega_i : i = 0, \ldots, 5\}$, we only know how to compute root bounds for $e \in \texttt{Expr}(\Omega_i)$ for $i = 0, \ldots, 3$.

§5. Filter techniques Multi-precision arithmetic is used in EGC libraries, which is much slower than machine floating-point arithmetic. To gain efficiency, a technique called "filtering" is used, which has been proved to be very effective in practice [30, 18]. The basic idea is simple: we first perform all the arithmetic computation using machine floating-point arithmetic, and then we "certify" the results, and go for the slower high precision computation only when this fails. For example, to determine the sign of an expression e, we compute an approximation value $\tilde{e}$ using machine floating-point arithmetic and an error bound err on the accumulated numerical errors such that $|e - \tilde{e}| \leq err$. e and $\tilde{e}$ must have the same sign if $|\tilde{e}| > err$.

There are two main categories of numerical filters: static and dynamic. Static filters can be computed at compile time for the most part, and have a low overhead at runtime. However, error bounds may be over-estimated and thus less effective. On the other hand, dynamic filters have higher runtime cost but are much more effective (i.e., fewer false rejections). We can also have

semi-static filters which combine both features.

Error tracking in filters (e.g., the BFS filter [18] used in the `Core Library`) is usually based on the specification of the floating-point arithmetic standard (IEEE 754). Interval arithmetic [1, 62] is a simpler and more traditional way to control the error. It is used in the `CGAL` library, covering all predicates in the geometry kernel [12, 74].

The `Core Library` is a set of C++ classes which aims at making robust programs easily constructed by any programmer. It defines a natural and simple numerical accuracy API with four accuracy levels:

- Level I: Machine Accuracy (i.e., IEEE 754 Standard)
- Level II: Arbitrary Accuracy (e.g., compute to 1000 bits)
- Level III: Guaranteed Accuracy (e.g., guarantee 100 bits)
- Level IV: Mixed Accuracy (i.e., a combination of the three previous levels)

The goal of the `Core Library` is to allow a single program to be run in any of these levels. The flexible choice of accuracy simplifies debugging and experimentation in many applications. The `Core Library` is designed so that most "ordinary C++ programs" (written without knowledge of the `Core Library` in mind) can be adapted to use the `Core Library` with minimal modifications (i.e., just inserting a single directive, `#include <CORE/CORE.h>`). EGC can be easily implemented using Level III accuracy.

1.3 Need for Transcendental Functions

The algebraic problems are the majority of problems treated in contemporary computational geometry. However, some non-algebraic examples, e.g., some kinds of Voronoi diagrams, shortest paths with disc obstacles [48], and non-holonomic motion planning, do arise in computational geometry. Transcendental functions such as $\exp x$, $\log x$, $\sin x$, etc. are needed as primitives. While a program involving only algebraic expressions can be "robustified" if it is recompiled under Level III in the `Core Library`, the exactness of geometric predicates involving transcendental functions cannot be guaranteed since there are no known root bounds for expressions involving transcendental functions. Consequently, we might want to scale back to Level II for such expressions.

Most transcendental functions used in geometric computation are elementary functions [64]. A function is **elementary** if it can be built up from a finite combination of constant functions, field operations and algebraic, exponential, logarithmic functions [87]. Among them the simplest elementary functions are the logarithm, exponential, and trigonometric functions. Many well-known elementary functions are special cases of hypergeometric functions. Hence, a general implementation of hypergeometric functions can be used as a base for implementing others.

The problem of evaluating hypergeometric functions is a highly classical problem (e.g., [22, 55]). The usual modus operandi here is one that is widely used in numerical analysis: the algorithms are based on fixed-precision arithmetic (e.g., IEEE Standard), and the goal is to design algorithms that try to minimize the round-off errors in the final result. With this procedure, one then gives a *posteriori* guarantees on the final precision, either by using error analysis,

or computationally via interval arithmetic. Such a *posteriori* bounds may not be sufficient for an application. In general, it is nontrivial to transform *a posteriori* methods into *a priori* ones (the obvious method of increasing precision iteratively may fail [94]). This has given rise to the "Table Maker's Dilemma" [51], described as the problem of computing correctly rounded values of transcendental functions. For machine-double tables, this problem was solved in [51] for some elementary functions using double-precision format. Nardin et al [68, 67] describe an evaluation method for confluent hypergeometric series (on large complex arguments) which they "verify" to be accurate to at least 9 digits. Their verification consists of a battery of 12 tests, but these do not constitute a proof of correctness. But using results of our chapter 2, we can automatically verify the accuracy of their evaluation.

Jeandel [45] describes a recent effort to provide hypergeometric functions in `GNU`'s multi-precision number package `GMP`. In [23], an implementation of Hypergeometric Function Package in the `Core Library` introduces transcendental functions into EGC for the first time.

1.4 Our Contributions

The basic goal of my thesis research is to improve the efficiency of our EGC approach and to extend it to transcendental computations. This requires a redesign of our current `Core Library` and providing effective algorithms for approximating transcendental functions.

As a part of our research effort, we redesigned the `Core Library`. While we keep the user-level interface intact, the on-top `Expr` class is redesigned as a template class with three parameters, `Filter`, `Rootbd` and `Kernel`. This

makes the implementation of `Expr` class independent from the implementation of these three submodules. Moreover, users can easily plug in different template parameters for experiments or various efficiency trade-offs. The key routines `computeExactSign()` and `computeApprox()` in `Expr` package are separated into five subroutines, which greatly avoid the unnecessary computations. The correctness and performance of these new subroutines have been carefully considered. Our benchmark shows that the redesigned system has 5-10 times speedup. For correctness of our implementation, we have ensured that the fairly extensive suite of sample programs in our `Core Library` produce the same results as the current system.

The underlying approximation "engine", the `BigFloat` subsystem, has also been redesigned. Instead of one class, we now split off from `BigFloat` a new class `BigFloat2`. They are both based on `MPFR` [63], a portable C library providing multi-precision floating point computation. `BigFloat` does not keep track of errors while `BigFloat2` does using interval arithmetic. This change makes our system more flexible since users can now choose one of them depending on their specific applications to get the most efficiency. And by using `MPFR`, we can now focus more exclusively on our EGC research.

Some efforts to extend the `Core Library` have also been made. We now provide a standard interface to help developers or users to write extensions for the `Core Library`, say, adding new operators or functions. Some C++ macros have been developed to simplify the task of writing such extensions. We also developed an interpreted version of our library, *InCore*, to help users prototype and rapidly develop their applications.

In the second part of this thesis, we investigate the absolute approximation of the general hypergeometric function $H(\mathbf{a};\mathbf{b};x) = {}_pF_q(\mathbf{a};\mathbf{b};x)$. We show that it

is solvable by presenting our complete approximation algorithm. Furthermore, we show that we can also approximate H when x is a "blackbox number", i.e., the number x is represented by a procedure that can return an approximation $\widetilde{x}$ to any desired absolute precision. This generalization is necessary for various applications: (a) argument reduction [64, 23], (b) evaluation of hypergeometric functions at irrational values such as $x = \pi$ or $x = \sqrt{2}$, (c) absolute approximation of functions that are derived from hypergeometric functions by irrational transformations of their x argument. An explicit bound on the complexity of our algorithm is given as well. A full implementation of our algorithm has been integrated into the `Core Library`.

Another important aspect of transcendental computation is the AGM algorithm. It has been used by Brent [10, 11] for fast evaluation of most elementary functions. However, he only gave the asymptotic error analysis which is useful for complexity analysis, but his algorithms cannot be implemented directly since the constants in the Big $O(\cdot)$ notation is unknown. We present a full non-asymptotic error analysis on these algorithms and implemented them using our redesigned `Core Library`.

We now summarize the contributions of this thesis:

- We redesigned the `Expr` package in the `Core Library` to increase modularity and significantly improved the efficiency of the key evaluation algorithms.

- We provided two `BigFloat` classes, which are now both based on `MPFR`, to make them more efficient and flexible for different applications.

- We developed a new mechanism for `Expr` to help users extend the functionalities of the `Core Library`.

- We implemented an interpreted version of our library.
- We presented a complete algorithm for absolute approximation of the general hypergeometric function with complexity analysis. Some problems, such as argument reduction, parameter pre-processing and mathematical constants are also addressed.
- We integrated hypergeometric function package with the `Core Library`.
- We gave an error analysis of the AGM algorithms and implementation in the `Core Library`.

The results of this thesis form the basis of the next release of the `Core Library` (version 2.0). All the experiments reported in this thesis are given with source codes and input data as part of the `Core Library` 2.0 distribution. They can be downloaded from

http://www.cs.nyu.edu/exact

Note that the `Core Library` is open source software.

Acknowledgments. The work on redesign is joint with Chee Yap, Sylvain Pion and Hervé Brönnimann. The work on hypergeometric functions is joint with Chee Yap. Some initial work on hypergeometric functions is a result of our collaboration with Jose Moreira and Maria Eleftheriou in IBM research.

Chapter 2

Redesign of the Core Library

The Core Library, as one of two general EGC software libraries, has been extensively developed for about ten years. It provides a collection of C++ classes to support numerical computation of algebraic expressions to arbitrary relative or absolute precision. Its simple, natural, numerical API and unique guaranteed accuracy computation feature make it quite useful in many applications, especially for developing robust geometric software. For example, it has been bundled with CGAL since 2003. However, the current design and development faces many issues. In this chapter, we discuss the redesign of the Core Library.

Overview of this chapter. In Section 2.1, we review the current Core Library design. In Section 2.2, we present the new design for the Expr package and discuss the key algorithms in it. The redesign of the BigFloat subsystem is shown in Section 2.3. In Section 2.4, we describe our new mechanism of writing extensions for the Core Library. Some experimental results and benchmarks are reported in Section 2.5. In Section 2.6, we present our newly designed interpreted version. We summarize in Section 2.7.

2.1 Review of the current `Core Library` design

The `Core Library` features an object-oriented design and is implemented in C++. A basic goal in the design of the `Core Library` is to make EGC techniques transparent and easily accessible to non-specialist programmers [52]. Built upon the `Real/Expr` package from Yap, Dubé and Ouchi [96, 72], it facilitates the rapid development of robust geometric applications.

There are four main subsystems in the current `Core Library` (version 1.7): expressions(`Expr`), real numbers(`Real`), big floating-point numbers(`BigFloat`) and big integer/rational numbers(`BigInt`, `BigRat`). They are built up in a layered structure. The `Expr` package at the top level provides the basic functionalities of exact geometric computation. In theory, this is the only interface that users need to program with. But experienced users can also access the underlying number classes directly, mainly for efficiency.

In the following, we raise design and efficiency issues in the current design of the `Expr` subsystem and the `BigFloat` subsystem.

`Expr` is the most important class of the library. It represents expressions that are constructed from rational constants by repeated application of the four basic arithmetic operations $\{+, -, *, /\}$ and square root. The leaves of expression trees can introduce arbitrary real algebraic numbers via the `rootOf()` operator. Internally, expressions are represented as directed acyclic graphs (DAGs). Each node in the DAG is a pointer to an instance of `ExprRep`. The expression DAG and most functionalities of `Expr` are actually implemented in `ExprRep` first and then simply wrapped under the interface of `Expr`.

There are several issues with the current design of the `Expr` package:

- There are critical facilities in `Expr` that should be modularized and made

extensible. In particular, the filter facility and the root bound facility have grown considerably over the course of library development and are now hard to maintain, to debug and to extend.

- The main evaluation algorithm of `Expr` has essentially three co-recursive subroutines. The current design does not separate their roles clearly, and can lead to costly unnecessary computations. For example, the subroutine for computing the sign of a node is always called even though this may not be necessary.

- The `Core Library` currently support only algebraic expressions. An overhaul of the entire design is needed to add support for non-algebraic expressions.

- Currently, users cannot easily add new operators to `Expr`. For instance, it is desirable to add determinants, summation and product, and diamond operators.

- It can be very inefficient to build certain huge expressions. For example, a naive implementation of the summation $\sum_{i=1}^{n}(1/i)$ would build an unbalanced tree of depth n. For large n, this recursive evaluation can easily cause stack overflow. Often such expressions can be automatically generated and evaluated on the fly. We would like to be able to introduce such kinds of nodes into `Expr`.

We now address the `BigFloat` class. This class is used by `Expr` to approximate real numbers. It is an arbitrary precision floating-point number system that is built on top of big integer package (we use `GNU`'s `GMP`[35] since version 1.3). A `BigFloat` is represented by the triple $\langle m, err, e\rangle$ of integers where m

is the mantissa, $err \geq 0$ is the error bound and e is the exponent. The triple represents the interval $[(m - err)B^e, (m + err)B^e]$ where $B = 2^{14}$. We say the error err is **normalized** when $err < B$. The following issues arise:

- The interval representation of `BigFloat` has performance penalty because the current design requires frequent error normalization. For example, our `sqrt()` function is about 25 times slower than `MPFR` as shown in Figure 2.6.
- The current `BigFloat` class only implements $\{+, -, *, /, \sqrt[2]{}\}$. A lack of implementations for the elementary functions such as $\exp, \log, \sin, \cos$ etc, limits the applications of the current `Core Library`.
- There are applications that do not need to maintain error. For example, in Newton iteration, in AGM computation and when bigfloats are used with exact ring operations only. The current `BigFloat` can be very inefficient for such computations. Although users can manually eliminate the error bounds in current `BigFloat` numbers, this process is error-prone.

The above issues call for a major redesign of the `Core Library`.

2.2 Redesign of `Expr` Package

The goal of our redesign is to increase modularity, introduce extensibility of expression nodes, and significantly improve efficiency. Another design goal is to ensure that these changes do not change the user-level interface. Thus, the original `Core Library` API, which is simple to understand and use, is intact. For instance, any current `CGAL` code using the `Core Library` as its number kernel should be able to run after recompilation.

2.2.1 Incorporation of Transcendental Nodes

We now consider expressions (DAG's) with transcendental operators. We classify a node (and the corresponding subexpression) as **transcendental** if any of its descendant nodes has a transcendental operator. For example, a transcendental node may be a leaf node such as π, or a unary node such as $\sin(\cdot)$.

In the current `Core Library`, we classify nodes into rational or irrational (this is used for certain operations). We now refine this classification into

integer, dyadic, rational, algebraic, transcendental.

Since transcendental numbers do not have constructive root bounds, we need to introduce a user-definable global value called **escape bound**. This bound is used as the root bound at transcendental nodes. Note that there is also another bound called **cutoff bound** which is used for a different purpose, as explained below.

Note that both the escape and cutoff bounds are absolute bounds on $-\lg|E|$ where E is some error that we wish to bound.

2.2.2 New template-based design of `ExprRep`

Our new design retains the basic structure of `Expr`, which is a thin wrapper around `ExprRep`. So we mainly focus on the redesign of `ExprRep` class.

§6. `ExprRep` and `ExprRepT` As noted in Section 2.1, the filter facility and root bound facility are embedded in the old `ExprRep` class. Now we factor them out from `ExprRep` into two functional modules: `Filter` and `Rootbd`. The `Real` class, which is already an independent module, is now viewed as an instance of an abstract number module called `Kernel`. The role of `Kernel` is to provide

approximations to the exact value. Typically we use bigfloats. Thus, we want to parametrize an expression class with these three modules. So our `ExprRep` class is redesigned as a template class with new name `ExprRepT`.

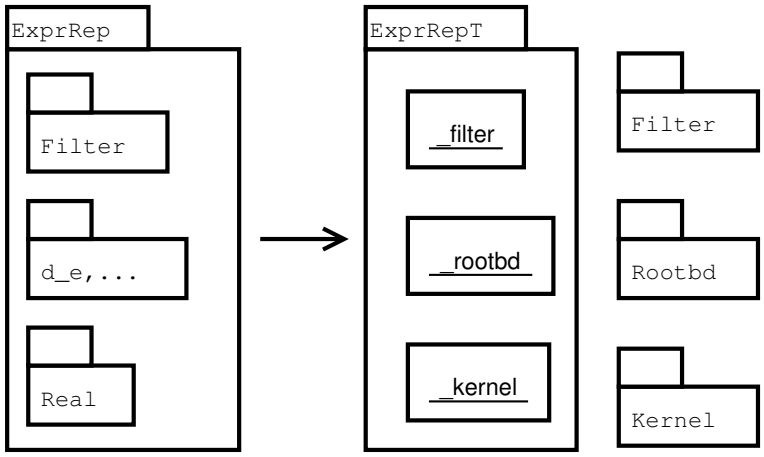

Figure 2.1: Comparison of `ExprRep` and `ExprRepT`

This applies the "delegation pattern" from [89]: delegate certain behaviors of the main class to other objects. The benefit of this new design is that, now we can replace `Filter`, `Rootbd` or `Kernel` without any changes in `ExprRepT`. The C++ prototype of `ExprRepT` is shown as follows:

```
template <typename Filter, typename Rootbd, typename Kernel>
class ExprRepT;
```

§7. Memory layout of `ExprRep` Three type parameters are used to define 3 data fields `_filter`, `_rootbd` and `_kernel` inside `ExprRepT`. However, the nature of their definitions are different: `_filter` is defined as a variable while `_rootbd` and `_kernel` are defined as pointers. We design it in this way because the filter computation will always be done first, but root bound information and

high precision computation (based on `_kernel`) may not be needed. No memory will be allocated for them when they are not needed. The memory layout of `ExprRepT` is shown in Figure 2.2. For a binary node, it uses a total of 48 bytes on a 32-bit architecture system (we assume that the `Filter` class use two IEEE doubles, i.e., 16 bytes).

Field name	Size
Dynamic type information (RTTI)	4
Reference counter	4
Operands : Node * [arity]	4×arity
Filter _filter	16
Kernel *_kernel	4
Rootbd *_rootbd	4
Cache *_cache	4
int _numType	4

Figure 2.2: `ExprRepT` memory layout. Sizes in bytes for a 32-bit architecture.

Note that for efficiency, a data field named `_cache` is added to cache some important and costly information such as `sign, uMSB, lMSB` once they become available. Currently, it is designed to cache `sign, uMSB, lMSB` using 3 integers (12 bytes on 32-bit system). The field `_numType` is used for node type classification.

§8. API specification for `ExprRepT` The C++ API specification for `ExprRepT` is given below.

```
1 template <typename T>
2 class ExprRepT {
```

```
3   T::Filter           _filter;  ///<- filter
4   mutable T::Kernel*  _kernel;  ///<- kernel
5   mutable T::Rootbd*  _rootbd;  ///<- root bound
6   mutable Cache*      _cache;   ///<- cache
7   int                 _numType; ///<- number type
8
9 public:
10  /// \name public accessors
11  //@{
12  /// The following functions are public and const, they are
13  /// used to retrieve the values of internal fields.
14  sign_t       sgn() const;  ///<- return sign
15  msb_t        uMSB() const; ///<- return upper bound of lg|e|
16  msb_t        lMSB() const; ///<- return lower bound of lg|e|
17  T::Kernel    r_approx(prec_t) const;  ///<- relative approx.
18  T::Kernel    a_approx(prec_t) const;  ///<- absolute approx.
19  const T::Filter& filter() const; ///<- return the filter
20  const T::Rootbd& rootbd() const; ///<- return root bound
21  const Cache&     cache()  const; ///<- return cache
22  //@}
23
24 protected:
25  /// \name protected accessors
26  /// the following functions are protected and non-const,
27  /// they are used to update corresponding internal field.
28  //@{
29  sign_t&    sgn();    ///<- return reference of sign field
30  msb_t&     uMSB();   ///<- return reference of uMSB field
31  msb_t&     lMSB();   ///<- return reference of lMSB field
32  T::Kernel& kernel(); ///<- return reference of _kernel field
33  T::Filter& filter(); ///<- return reference of _filter field
34  T::Rootbd& rootbd(); ///<- return reference of _rootbd field
35  Cache&     cache();  ///<- return reference of _cache field
36  //@}
37
38 protected:
39  /// \name evaluation functions
40  //@{
41  /// refine current approximation until the sign is known
```

```
42    void refine();
43    /// compute the sign
44    virtual bool compute_sgn() const;
45    /// compute the upper bound of lg|e|
46    virtual bool compute_uMSB() const;
47    /// compute the lower bound of lg|e|
48    virtual bool compute_lMSB() const;
49    /// compute relative approximation
50    virtual void compute_r_approx(prec_t) const;
51    /// compute absolute approximation
52    virtual void compute_a_approx(prec_t) const;
53    /// compute root bound
54    virtual void compute_rootbd() const;
55    //@}
56
57    /// \name helper functions
58    //@{
59    /// return true if node is algebraic
60    bool is_algebraic() const;
61    /// return true if _cache has been initialized
62    bool cache_initialized() const;
63    /// initialize the cache
64    void initialize_cache();
65    /// return true if _kernel has been initialized
66    bool kernel_initialized() const;
67    /// initialize the kernel
68    void initialize_kernel();
69    /// convert relative precision to absolute precision
70    prec_t rel2abs(prec_t) const;
71    /// convert absolute precision to relative precision
72    prec_t abs2rel(prec_t) const;
73    //@}
74    ...
75  };
```

Note that in the definition of `ExprRepT`, we only have one type parameter `T`. This type is really a combination of the three types `Filter`, `Rootbd`, `Kernel`. This packaging is useful because we can just pass a single type parameter `T` to

other template classes which are derived from `ExprRepT`.

There are two versions for each function such as `sgn()`, `uMSB()`, `lMSB()`, etc.; they are protected and public respectively. The ones with C++ keyword `const` are called *const* versions which can only be put on the right hand side of a C++ expression, while the others are *non-const* versions which can be put on the either side. Lazy evaluation is used in the `Core Library`, and this requires the modification of some internal data fields in those `const` version functions. In order to solve this conflict, data fields `_rootbd, _kernel` and `_cache` are marked as `mutable` to avoid checking done by compilers.

§9. `ExprRepT` class hierarchy `ExprRepT` only defines the abstract structures and operations; the actual implementations are delegated to the derived classes. The new design still keep the same class hierarchy of `ExprRepT` and its derived classes as in the `Core Library` 1.x except that a "T" is appended at each class names. A full hierarchy diagram is shown in Figure 2.3.

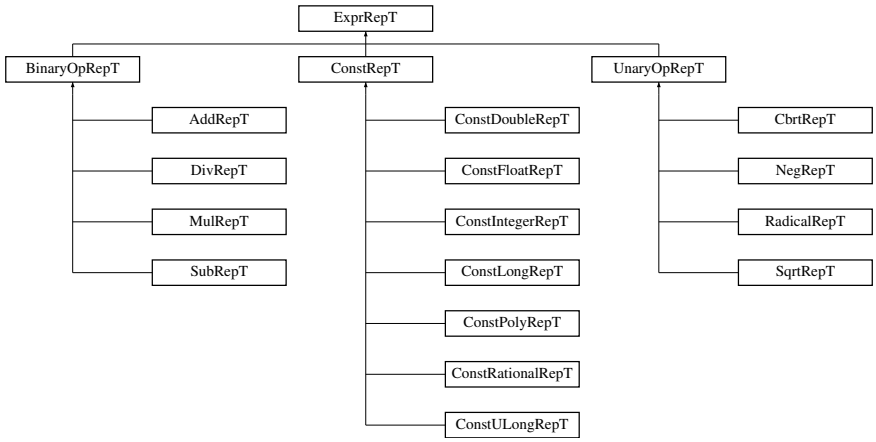

Figure 2.3: `ExprRepT` Class Hierarchy

§10. Design of the `Filter` class The `Filter` class is used to provide some information such as `sign, uMSB, lMSB` using "filter technique" before `ExprRepT` goes high precision computation. In order to make it work for `ExprRepT`, the `Filter` class is required to implement the following interface:

```
1  class Filter {
2    typedef Filter this_cls;
3    // put internal maintained data fields here!
4    //
5  public:
6    // return true if filter succeed
7    bool is_ok() const;
8    // return correct sign (only valid if filter succeed)
9    int sgn() const;
10   // return upper bound of MSB (only valid if filter succeed)
11   long uMSB() const;
12   // return lower bound of MSB (only valid if filter succeed)
13   long lMSB() const;
14   // return true if filter value has given relative precision
15   bool has_rel_prec(prec_t) const;
16   // return relative approximated value from filter value
17   double r_approx(prec_t) const;
18   // return absolute approximated value from filter value
19   double a_approx(prec_t) const;
20
21   // assignment function
22   template <typename T> void set(const T&, bool=true);
23   // special function to make filter failed
24   void invalidate();
25
26   // arithmetic functions
27   void neg(const this_cls&);
28   void add(const this_cls&, const this_cls&);
29   void sub(const this_cls&, const this_cls&);
30   void mul(const this_cls&, const this_cls&);
31   void div(const this_cls&, const this_cls&);
32   void sqrt(const this_cls&);
33   void cbrt(const this_cls&);
```

```
34    void root(const this_cls&, unsigned long);
35  };
```

Some design considerations and issues for this interface include:

- The function `is_ok()` is the most important call in the `Filter` class. It is called by `ExprRepT` to test whether filter succeed or failed by checking its return values `true` or `false`. Only when `is_ok()` returns `true`, the other 6 functions: `sgn()`, `uMSB()`, `lMSB()`, `has_rel_prec()`, `r_approx()` and `a_approx()` will be called to get information from the `Filter`. Thus, a dummy filter can be implement as follows: let `is_ok()` function return `false` and other 6 functions return some dummy values (this is necessary in order to avoid compilation errors).

- The function `has_rel_prec()` is used to test whether the filter can provide enough precision. If not, `ExprRepT` will do higher precision computation.

- The assignment function `set()` is designed for leaf `ExprRepT` node. It will be called whenever a leaf `ExprRepT` node is constructed. Since in the `Core Library` the leaf node can be constructed from the following types:

  `long`, `unsigned long`, `double`, `BigInt`, `BigRat`, `BigFloat`,

  it has to accept all types above. Different versions of `set()` have to be defined if the user's `Filter` class handles those types differently. The second `boolean` parameter is used to indicate whether the value of the first argument is exact or not. Its default value is `true`. A dummy filter can just define one `set()` which takes a template parameter and does nothing inside it.

- The arithmetic functions `neg()`, `add()`, `sub()`, `mul()`, `div()`, `sqrt()`, `cbrt()` and `root()` are designed to be called when a corresponding `ExprRepT` node is constructed. For example, when a `MulRepT` node is created, the function `mul()` will be called. At the same time, the filter instances of the children nodes are passed to those functions so that the current node can update its filter information from them. Again, a dummy filter can have a empty implementation. (The `Filter` class in the `Core Library` 1.7 implements those functions as C++ operators, which is more intuitive, but the new designed interface can avoid temporary memory storages to be more efficient.)

- A global function `set_filter_flag()` is defined to allow users to enable or disable filter facility at run time. So in the implementation of `Filter`, developers may check this flag by calling the global function `get_filter_flag()` (if it returns `false`, then it means users have disabled the filter facility in their applications). However, for a non-dummy filter, the underlying filter computation will still be performed even if the filter facility is disabled. To completely avoid the filter computation, a dummy filter should be used.

- The `invalidate()` function is used to invalidate the filter value of the current node (for example, by setting it to be IEEE value infinite). Note that this applies only to the current node. To invalidate the filter values for all nodes, the users should call `set_filter_flag(false)`.

With the newly designed filter interface, given the recursive rules of a filter algorithm, writing a `Filter` class just requires translating those rules into above corresponding functions. An effective floating-point filter, `BfsFilter`, is

provided in the `Core Library`. It is based on [17] with some extended rules for cubic root and k-th root extraction. The detail rules of the `BfsFilter` and how to translate the rules into code are given in Appendix A.

§11. Design of the `Rootbd` Class The `Rootbd` class is used to provide root bound information when `ExprRepT` cannot determine the sign of the current node. While it was embedded inside `ExprRep` in the `Core Library` 1.x, now it becomes a separate class. The following interface of the `Rootbd` class is required by `ExprRepT`:

```
1  class Rootbd {
2    typedef Rootbd this_cls;
3    // put internal maintained data fields here!
4    //
5  public:
6    // return true if the root bound computation is constructive
7    static bool is_constructive();
8    // return root bound
9    unsigned long get_bound() const;
10   // return true if it is degree based
11   static bool is_degree_based() const;
12   // set the degree bound of the current node
13   void set_degree_bound(unsigned long);
14   // get the degree bound of the current node
15   unsigned long get_degree_bound();
16
17   // assignment function
18   template <typename T> void set(const T&);
19
20   // arithmetic functions
21   void neg(const this_cls&);
22   void add(const this_cls&, const this_cls&);
23   void sub(const this_cls&, const this_cls&);
24   void mul(const this_cls&, const this_cls&);
25   void div(const this_cls&, const this_cls&);
26   void sqrt(const this_cls&);
```

```
27    void cbrt(const this_cls&);
28    void root(const this_cls&, unsigned long);
29  };
```

We use the term "constructive" for root bound computation. We say a root bound is "constructive" to mean the root bound information is computed by some recursive rules. Functions `is_constructive()` is used by `ExprRepT` to test whether the root bound computation is constructive or not. If it returns `false`, `ExprRepT` will skip the recursive calls of `compute_rootbd()` for each of its children. Since there are no known constructive root bounds for non-algebraic expressions, `ExprRepT` will also skip the root bound computation if the current node is non-algebraic.

Moreover, if `is_constructive()` returns `true` and the current node is algebraic, then `get_bound()` will be called by `ExprRepT` to obtain the root bound. Otherwise the global function `get_escape_bound()` will be called and its returned value is used as a substitute root bound. Users can use `set_escape_bound()` to set this bound at the run time.

As we mentioned before, `ExprRepT` contains a pointer instance `_rootbd` of `Rootbd`. We will not allocate memory for `_rootbd` unless we need root bound information. Hence the function `is_constructive()` is designed to be a `static` function which can be called without any `Rootbd` object being constructed, i.e., no memory is allocated.

Most constructive root bounds are based on the degree of the expression node. The function `is_degree_based()` is used to test this property. A `Rootbd` class needs to maintain an upper bound on the degree which is called **degree bound** if `is_degree_based()` returns `true`. Degree bounds can be computed

using some recursive rules [52, Table 2.1]. Yap proposed an algorithm[1] which detects the shared expression nodes in DAG to get a better bound. However, such an algorithm can only be invoked by `ExprRepT` since `Rootbd` class can not access DAG. Hence, two functions `set_degree_bound()` and `get_degree_bound()` are required to allow `ExprRepT` to update degree bound inside `Rootbd`.

As the `Filter` class above, the assignment function `set()` and arithmetic functions `neg(), add()`, etc. are designed for corresponding types of nodes. However, those functions in the `Rootbd` class are only called when root bound information is needed while in the `Filter` class they are called always even if the filter is disabled by users.

To design a `Rootbd` class for a constructive root bound algorithm, we just need to translate its constructive rules and root bound computation formula into corresponding function calls. We implemented 3 different `Rootbd` classes for 3 constructive root bounds: Degree-Measure bound, BFMSS bound [15, 16] and Li-Yap bound [54] in the `Core Library`. See Appendix B for an example of how we implement BFMSS bound.

Two other `Rootbd` classes, `DummyRootbd` and `MaxRootbd` are provided, too. The `DummyRootbd` class just returns `false` in its `is_constructive()` function. The `MaxRootbd` takes two other `Rootbd` classes as parameters and does the root bound computation at the same time, and then returns the maximal bound.

§12. Design of the `Kernel` Class The `Kernel` is used by `ExprRepT` to compute an approximation of the current node. Typically we use bigfloat number for this class. The minimal interface is given below:

[1] This algorithm did not appear in any publication, but it has been implemented in the `Core Library` since version 1.4

```
1  class Kernel {
2    typedef Kernel this_cls;
3    // put internal maintained data fields here!
4    //
5  public:
6    // return true if the sign is known
7    bool has_sgn() const;
8    // return true if within required relative precision
9    bool has_rel_prec(prec_t) const;
10   // return sign (only valid if has_sgn() return true)
11   sign_t sgn() const;
12   // return upper bound of MSB
13   msb_t  uMSB() const;
14   // return lower bound of MSB
15   msb_t  lMSB() const;
16
17   // assignment functions
18   template <typename T> void set(const T&);
19   template <typename T> void set(const T&, prec_t);
20
21   // arithmetic functions
22   void neg(const this_cls&, prec_t);
23   void add(const this_cls&, const this_cls&, prec_t);
24   void sub(const this_cls&, const this_cls&, prec_t);
25   void mul(const this_cls&, const this_cls&, prec_t);
26   void div(const this_cls&, const this_cls&, prec_t);
27   void sqrt(const this_cls&, prec_t);
28   void cbrt(const this_cls&, prec_t);
29   void root(const this_cls&, unsigned long, prec_t);
30 };
31 std::istream& operator>>(std::istream&, this_cls&);
32 std::ostream& operator<<(std::ostream&, const this_cls&);
```

The function `has_sgn()` checks whether the sign of the current approximation is known or not (whether the current approximation is separated from zero). If it returns `true`, the return value of `sgn()` will be used when needed.

The function `has_rel_prec()` is used to test whether current approximation has enough (relative) precision.

The other functions are similar to those in the `Filter` class except that now the assignment function `set()` and arithmetic functions require an extra `prec_t` type parameter. This parameter is used to specify the precision that needed in the corresponding arithmetic operations. Note that it is a relative precision.

Two global operators >> and << have to be overloaded by the `Kernel` to provide reading/writing facilities for `ExprRepT`. By default, only 6 digits are printed out.

We will present our new `BigFloat2` class later as an example implementation of the `Kernel` class. Users are free to use their own number type class as long as they implement the above interface. For example, we can use our old `BigFloat` or `REAL` in `iRRAM` [65].

§13. Number Type inside `Expr` In order to bring transcendental functions or constants into `Expr`, we need to identify the number type of each expression node. The field `_numType` is used for this purpose. For each expression node, its `_numType` can be one of the following enum values:

```
1 enum NODE_NUMTYPE {
2   NODE_NT_INTEGER = 0,        // integer node
3   NODE_NT_DYADIC = 1,         // floating-point node
4   NODE_NT_RATIONAL = 2,       // rational node
5   NODE_NT_ALGEBRAIC = 3,      // algebraic node
6   NODE_NT_TRANSCENDENTAL = 4 // transcendental node
7 }
```

For a leaf node, the `_numType` is determined by its internal data type. For other operations, we can deduct them from the Table 2.1 shown below:

Operator op	A op B
$+/-/\times$	$\max\{A.\texttt{\_numType}, B.\texttt{\_numType}\}$
$\div$	$\max\{\texttt{NODE\_NT\_RATIONAL}, A.\texttt{\_numType}, B.\texttt{\_numType}\}$
$\sqrt[k]{\cdot}$	$\max\{\texttt{NODE\_NT\_ALGEBRAIC}, A.\texttt{\_numType}\}$
Transcendental	$\max\{\texttt{NODE\_NT\_TRANSCENDENTAL}, A.\texttt{\_numType}\}$

Table 2.1: Rules for computing number type

§14. `Expr` with plug-gable Filter, Root Bound and Kernel A new template class `ExprT` is defined as:

```
template <typename Filter, typename Rootbd, typename Kernel>
class ExprT;
```

`Expr` is now a typedef:

```
typedef ExprT<BfsFilter, BfmssRootbd, BigFloat2> Expr;
```

Thus, the default `Expr` class uses BFS filter, BFMSS root bound and our `BigFloat2` kernel. However, now the users are free to plug in their own filter, root bound or kernel classes. For example, we can use a better filter and root bound class for division-free expressions.

2.2.3 Improved Approximate Evaluation Algorithms

`computeExactSign()` and `computeApprox()` are two key evaluation subroutines in the `Core Library` 1.x. The subroutine `computeExactSign()` is used for computing the sign, upper and lower bound on the most significant bits of the current node (actually it even computes the root bound information). The subroutine `computeApprox()` is used to propagate the composite precision and

compute the approximation. These two subroutines recursively call each other, involving unnecessary computations. Now we divided them into five subroutines to be more efficient. A multi-level computation paradigm is designed for optimization.

§15. Algorithms for `sgn()`, `uMSB()` and `lMSB()` The sign of an expression node is important and used in many places in the `Core Library`. For example, division operator needs to check the sign of the child nodes to avoid "divided by zero" error. The upper and lower bound on the most significant bits of an expression node E, denoted by E^+ and E^- respectively, are also important. They are used to convert the precision from an absolute bound to a relative one or vice-versa. These three parameters are maintained in `ExprRepT`. The subroutine `computeExactSign()` computes them simultaneously using the rules shown in Table 2.2, Table 2.3 and Table 2.4.

$E =$	$E.sgn()$
Constant x	$sign(x)$
$E_1 \pm E_2$	$\begin{cases} \pm E_2.sgn() & \text{if } E_1.sgn() = 0 \\ E_1.sgn() & \text{if } E_2.sgn() = 0 \\ E_1.sgn() & \text{if } E_1.sgn() = \pm E_2.sgn() \\ E_1.sgn() & \text{if } E_1.sgn() \neq \pm E_2.sgn() \text{ and } E_1^- > E_2^+ \\ \pm E_2.sgn() & \text{if } E_1.sgn() \neq \pm E_2.sgn() \text{ and } E_1^+ < E_2^- \\ \text{unknown} & \text{otherwise} \end{cases}$
$E_1 \times E_2$	$E_1.sgn() * E_2.sgn()$
$E_1 \div E_2$	$E_1.sgn() * E_2.sgn()$
$\sqrt[k]{E_1}$	$E_1.sgn()$

Table 2.2: Rules for computing sign recursively

We can see these rules are recursive. In order to compute this information for the current node, `computeExactSign()` has to compute its children's

$E =$	E^+
Constant x	$\lceil \log_2 x \rceil$
$E_1 \pm E_2$	$\begin{cases} E_2^+ & \text{if } E_1.sgn() = 0 \\ E_1^+ & \text{if } E_2.sgn() = 0 \\ \max\{E_1^+, E_2^+\} + 1 & \text{if } E_1.sgn() = \pm E_2.sgn() \\ \max\{E_1^+, E_2^+\} & \text{if } E_1.sgn() \neq \pm E_2.sgn() \\ \text{unknown} & \text{otherwise} \end{cases}$
$E_1 \times E_2$	$E_1^+ + E_2^+$
$E_1 \div E_2$	$E_1^+ - E_2^-$
$\sqrt[k]{E_1}$	$\lceil E_1^+ / k \rceil$

Table 2.3: Rules for computing upper bound of MSB recursively

`computeExactSign()` first, which consequently computes this information for all nodes below it in the DAG even we do not need them all.

In contrast to this, in the new design three separated subroutines: `sgn()`, `uMSB()` and `lMSB()` are introduced and they are only called when necessary. The algorithms for them are very similar. Basically, each algorithm consists of the following steps (using `sgn()` as an example):

Sign Evaluation Algorithm

1. Ask the filter if it knows the sign;
2. Else if the cache exists, ask if it cached the sign;
3. Else if the approximation (`_kernel`) exists, ask if it can give the sign;
4. Else if the virtual function `compute_sgn()` return `true`, return `sgn()` (it returns the sign in the cache);
5. Else call `refine()` (to be presented next) to get the sign.

For efficiency, we use five levels of computation here: filter, cache, kernel, recursive rules, `refine()`. Note that we do not put the cache at the first level. We do not even cache the `sign`, `uMSB` and `lMSB` information when the filter

$E =$	E^-
Constant x	$\lfloor \log_2 x \rfloor$
$E_1 \pm E_2$	$\begin{cases} E_2^- & \text{if } E_1.sgn() = 0 \\ E_1^- & \text{if } E_2.sgn() = 0 \\ \max\{E_1^-, E_2^-\} + 1 & \text{if } E_1.sgn() = \pm E_2.sgn() \\ E_1^- - 1 & \text{if } E_1.sgn() \neq \pm E_2.sgn() \text{ and } E_1^- > E_2^+ \\ E_2^- - 1 & \text{if } E_1.sgn() \neq \pm E_2.sgn() \text{ and } E_2^- > E_1^+ \\ \text{unknown} & \text{otherwise} \end{cases}$
$E_1 \times E_2$	$E_1^- + E_2^-$
$E_1 \div E_2$	$E_1^- - E_2^+$
$\sqrt[k]{E_1}$	$\left\lfloor E_1^- / k \right\rfloor$

Table 2.4: Rules for computing lower bound of MSB recursively

succeed because a `Cache` structure is large and we try to avoid costly memory allocation. The pseudo-codes in C++ are given below:

```
1  sign_t sgn() const {
2    // Step 1:
3    if (filter().is_ok())
4      return filter().sgn());
5    // Step 2:
6    if (cache_initialized() && cache().has_sgn())
7      return cache().sgn();
8    // Step 3:
9    if (kernel_initialized() && kernel().has_sgn())
10     return kernel().sgn();
11   // Step 4:
12   if (compute_sgn())
13     return sgn();
14   // Step 5:
15   refine();
16   return kernel().sgn();
```

In the algorithm above, we adapt another design pattern which is called "template method pattern" [32]: define the skeleton of an algorithm in terms of abstract operations which subclasses will override to provide concrete behavior.

In the derived classes of `ExprRepT`, it is sufficient to just override the virtual function `compute_sgn()` when appropriate. For example, `ConstIntegerRepT` may implement `compute_sgn()` as follows:

```
1 virtual bool compute_sgn() {
2   sgn() = value.sgn();
3   return true;
4 }
```

and `MulRepT` may override its `compute_sgn()` function as:

```
1 virtual bool compute_sgn() {
2   sgn() = first.sgn() * second.sgn();
3   return true;
4 }
```

§16. Algorithms for `r_approx()` and `a_approx()` For approximation algorithms, we made two improvements:

1. The new approximation algorithm does not compute the sign before approximating when it is unnecessary; the old algorithm always does.

2. While the the `Core Library` 1.x uses one subroutine `computeApprox()` to do approximation which requires a composite precision, the new design use two subroutines, `r_approx()` for relative approximation and `a_approx()` for absolute approximation. This can avoid unnecessary conversion between them.

The algorithm for computing approximation in relative (or absolute) precision is given below.

Approximation Algorithm

1. Use the filter value if it works and has the required precision.
2. Else if the approximation has not been initialized, then initialize it.
3. Else if the approximation does not have enough precision, then call virtual function `compute_r_approx()` or `compute_a_approx()` to obtain a better approximation with required precision.
4. Return the approximation.

It uses two levels of computation: filter and kernel. The detailed algorithm in C++ for `r_approx()` is as follows:

```
1  Kernel r_approx(prec_t prec) const {
2    // Step 1:
3    if (filter().is_ok() && filter().has_rel_prec(prec))
4      return filter().r_approx(prec);
5    // Step 2:
6    if (!kernel_initialized())
7      initialize_kernel();
8    // Step 3:
9    if (!kernel().has_rel_prec(prec))
10     compute_r_approx(prec);
11   // Step 4:
12   return kernel();
13 }
```

We also applied "template method pattern". The derived classes of `ExprRepT` will override `compute_r_approx()` or `compute_a_approx()` function to implement actual approximating computation. Moreover, only one of them is required to be implemented if `uMSB()` or `lMSB()` is available because two default implementations are already provided in `ExprRepT`:

```
1  virtual bool compute_r_approx(prec_t prec)
2  { return compute_a_approx(rel2abs(prec)); }
```

```
3 virtual bool compute_a_approx(prec_t prec)
4 { return compute_r_approx(abs2rel(prec)); }
```

With the default implementations, if one of them is missing, `ExprRepT` will automatically call another one with converted precision. This simplifies the work of designing derived classes of `ExprRepT`. For instance, `AddRepT` may want to override `compute_a_approx()` while `MulRepT` may prefer to override `compute_r_approx()`.

In order to keep the current interface, we still provide the function `approx()` in `Expr` level that will be redirected to call `r_approx()` or `a_approx()`.

§17. Algorithm for `refine()` The override-able `compute_sgn()`, `compute_uMSB()` and `compute_lMSB()` return `false` when they cannot determine the sign, upper and lower bound on the most significant bit of the current node. Then a refinement of the current approximation is required. The abstract algorithm for such refinement is as follows:

Refinement Algorithm

1. If an approximation exists, use its precision as the start precision, otherwise use 52 bits instead since this is the relative precision that the floating-point filter can provide.
2. Check if the Rootbd is non-constructive or the current node is non-algebraic. If so, use the global escape bound. Otherwise, compute the degree bound if the Rootbd is degree based. Then compute the root bound.
3. Compare the bound that we get in step 2 with the global cutoff bound. Take the minimum of them.
4. Run a loop which doubles the precision each time until it hits the precision that we obtained in step 3. In the loop, compute a better approximation (absolute) on the current node. Once the refined approximation can give the correct sign, return immediately.
5. Upon the loop termination, if either escape bound or cutoff bound was used, print out a warning message in a diagnostic file, otherwise set the current node to be zero.

§18. Cutoff Bound **Cutoff bound** is used in above algorithm. It is a global variable and set to be `CORE_INFTY` by default. It is different from the escape bound we mentioned before. The escape bound is used when we do not have the root bound for the current node, say a transcendental node, while the cutoff bound is used to restrict the root bound to a certain precision. For instance, if users want to speed up their programs for testing, they can set a small cutoff bound using `set_cut_off_bound()`. However, users should realize that sign

computations are not guaranteed any more. The pseudo code for `refine()` is given:

```
1  void refine() {
2    // Step 1:
3    prec_t prec = 52, bound;
4    int bound_type;
5    if (!kernel_initialized())
6      initialize_kernel();
7    else
8      prec = kernel().rel_prec();
9    // Step 2:
10   if (!T::Rootbd::is_constructive() || !is_algebraic()) {
11     bound = get_escape_bound();
12     bound_type = 0;
13   } else {
14     if (T::is_degree_based()) compute_degree_bound();
15     bound = rootbd().get_bound();
16     bound_type = 2;
17   }
18   // Step 3:
19   if (bound > get_cut_off_bound()) {
20     bound = get_cut_off_bound(); bound_type = 1;
21   }
22   // Step 4:
23   do {
24     prec <<= 1;
25     compute_a_approx(prec);
26     if (kernel().has_sgn()) {
27       if (cache_initialized()) {
28         sgn() = kernel().sgn();
29         uMSB() = kernel().uMSB(); lMSB() = kernel().lMSB();
30       }
31       return;
32     }
33   } while (prec < bound);
34   // Step 5:
35   if (bound_type < 2) { // hit escape bound or cut-off bound
36     core_warning("Hit Escape bound or Cut-off Bound");
```

```
37    }
38    kernel().set(0.0)
39    if (cache_initialized()) {
40      sgn() = 0; uMSB() = 0; lMSB() = MSB_MIN;
41    }
42  }
```

Note that `rootbd()` will call `compute_rootbd()`, which consequently computes the root bound information recursively .

§19. Algorithm for `compute_degree_bound()` Most known constructive root bounds are based on the degree of an algebraic expression. In many applications, computing minimal polynomials explicitly is very expensive, so obtaining the exact degree is hard. In practice, we are more interested in computing an upper bound on the degree, say "degree bound". Such a bound can be computed recursively using the rules in Table 2.5:

Expr E	$d(E)$
Rational $\frac{a}{b}$	1
RootOf(P, i)	$deg(P)$
$E = F \pm G$	$d(F)d(G)$
$E = F \times G$	$d(F)d(G)$
$E = F \div G$	$d(F)d(G)$
$E = \sqrt[k]{F}$	$k \cdot d(F)$

Table 2.5: Rules for computing degree bound.

A better bound can be obtained by detecting the shared nodes: if a node E is shared, then $d(E)$ will return 1 for the subsequent calls. The `Core Library` 1.x implemented this algorithm. However, to detect whether a node is shared, an extra flag `visited` is stored in each `ExprRep` and a clean operation is needed for all nodes after $d(E)$ is computed.

For efficiency, we now use a map data structure (`std::map` in `STL`) to re-implement this algorithm[2]: we first create a map M, then for each node E, if its address does not appear in M, use its degree bound value $d(E)$ and save its address in M, otherwise use 1 instead. At the end we just discard the map M.

2.2.4 Improved Propagation of Precision

The centerpiece of an EGC library is the algorithm for approximate expression evaluation. The `Core Library` uses a *precision-driven* algorithm with composite precision. The error analysis of the various bigfloat operations of this algorithm were studied in detail by Ouchi [72]. Using this analysis to propagate composite precision, various small constants appear in the code which caused the current code to be hard to understand and maintain. Here we introduce a simpler and more intuitive solution, where we propagate either absolute or relative precision, but not both. It basically follows the idea from [94] but with some simplification.

§20. Precision Conversion We can transform absolute precision to relative precision or vice-versa if we know the location of the first significant bit in E. It is shown below:

LEMMA 1. *Let $\tilde{E}$ be an approximation of E, while E^+ and E^- are upper and lower bounds on $\lg|E|$, i.e., $2^{E-} \leq |E| \leq 2^{E+}$.*
(i) If we want to guarantee a absolute bits in E, it is enough to guarantee $a+E^+$ relative bits in E.
(i) If we want to guarantee r relative bits in E, it is enough to guarantee $r-E^-$

[2]Sylvain Pion mentioned this idea when I visited INRIA in February 2006.

absolute bits in E.

Proof. It is clear from

$$|\tilde{E} - E| \leq |E|2^{-(a+E^+)} \leq 2^{-a},$$

and

$$|\tilde{E} - E| \leq 2^{-(r-E^-)} \leq |E|2^{-r}.$$

Q.E.D.

We normally restrict the relative precision r to positive values. To see why, note that when $r \leq 0$, then $\tilde{E} = 0$ satisfies $|\tilde{E} - E| \leq |E|2^{-r}$ always. Similarly, we may also assume absolute precision $a > -E^+$. This latter assumption allows us to simplify some of the bounds in [94].

§21. Absolute Approximation for Addition/Subtraction For addition/subtraction, it is easy to guarantee absolute precision:

LEMMA 2. *(cf. [94, Lemma 9, (i)]) Let $z = x + y$. To guarantee p absolute bits in z, it suffices to approximate x and y to $p + 2$ absolute bits and perform the addition in $p+1$ absolute bits of precision (or $p+z^+ +1$ bits relative precision).*

Proof.

$$\begin{aligned} |\tilde{x} \oplus \tilde{y} - (x + y)| &\leq |\tilde{x} \oplus \tilde{y} - (\tilde{x} + \tilde{y})| + |\tilde{x} - x| + |\tilde{y} - y| \\ &\leq 2^{-(p+1)} + 2^{-(p+2)} + 2^{-(p+2)} \\ &= 2^{-p}. \end{aligned}$$

Q.E.D.

§22. Absolute Approximation for Summation We can extend this to summation:

COROLLARY 3. *To guarantee p absolute bits in $z = \sum_{i=1}^{n} x_i$, it suffices to approximate x_i to $(p + \lceil \lg n \rceil + 1)$ absolute bits and perform the addition in $(p + \lceil \lg n \rceil + 1)$ absolute bits of precision.*

Proof. A naive implementation of $\sum_{i=1}^{n} x_i$ can be done by first setting $z = 0$, then approximating x_i for $i = 1, \ldots, n$, and finally do n additions of $z = z + x_i$ for $i = 1, \ldots, n$. Hence the total absolute error is $2n \cdot 2^{-(p+\lceil \lg n \rceil + 1)} \leq 2^{-p}$.

Q.E.D.

§23. Relative Approximation for Multiplication For multiplication, guaranteed relative precision is more natural:

LEMMA 4. *(cf. [94, Lemma 8, (i)]) To approximate $z = x \cdot y$ to within p relative bits of precision, it is enough to approximate x and y to within $p+2$ relative bits of precision and perform the multiplication in $p + 2$ relative bits of precision.*

Proof. Let $\delta = 2^{-(p+2)}$. From $|\tilde{x} - x| \leq |x|\delta$, we have $|\tilde{x}| \leq |x|(1 + \delta)$. Similarly we have $|\tilde{y}| \leq |y|(1 + \delta)$ and $|\tilde{x} \cdot \tilde{y}| \leq |x \cdot y|(1 + \delta)^2$. Hence

$$\begin{aligned}
|\tilde{x} \odot \tilde{y} - x \cdot y| &\leq |\tilde{x} \odot \tilde{y} - \tilde{x} \cdot \tilde{y}| + |\tilde{x}| \cdot |\tilde{y} - y| + |y| \cdot |\tilde{x} - x| \\
&\leq |\tilde{x} \cdot \tilde{y}| \cdot \delta + |\tilde{x} \cdot y| \cdot \delta + |x \cdot y| \cdot \delta \\
&\leq |x \cdot y|(1 + \delta)^2\delta + |x \cdot y|(1 + \delta)\delta + |x \cdot y| \cdot \delta \\
&= |x \cdot y|(3\delta + 3\delta^2 + \delta^3) \\
&\leq |x \cdot y|(4\delta) \\
&= |x \cdot y|2^{-p}.
\end{aligned}$$

Since $p > 0$, $\delta = 2^{-(p+2)} < 1/4$. **Q.E.D.**

§24. Precision Propagation without Exact Ring Operation It should be noticed, from the above error analysis, exact ring operations are not necessary although this is assumed in [94]. To see the the difference, we did the following experiment:

Experiment 1 Precision Propagation for Multiplication

We use two methods to compute $z = \sqrt{2} \cdot \sqrt{3}$ to relative precision p.

Method 1: approximate $\sqrt{2}$ and $\sqrt{3}$ to $(p+2)$ relative bits, then use exact multiplication to multiply the approximate values.

Method 2: do the same as the Method 1 except performing the multiplication in $(p+2)$ bits relative precision.

The timing (in microseconds) is shown in the Figure 2.4 (we use loops to repeat the experiment since the time for a single run is short). Method 2 seems to be more efficient.

Precision	Loops	Method 1	Method 2	Speedup
10	1000000	345	191	45%
100	100000	60	46	23%
1000	10000	72	71	1%
10000	1000	267	219	18%
100000	100	859	760	12%

Figure 2.4: Timing for computing $\sqrt{2} \cdot \sqrt{3}$ w/ and w/o exact multiplication

§25. Relative Approximation for Addition/Subtraction Using Lemma 1, we can do relative approximation for addition/subtraction if we know the lower bound on $\lg|z|$: to guarantee p relative bits in $z = x + y$, we can first convert to guarantee $p - z^-$ absolute bits in z, then applying Lemma 2, we need to guarantee $p - z^- + 2$ absolute bits in x and y. After we convert them again, we need to guarantee $p - z^- + x^+ + 2$ relative bits in x and $p - z^- + y^+ + 2$ bits in y. With the similar precision transformation, the relative precision we need for performing addition is $p + 2$ bits. Note a small error in [94, Lemma 9, (ii)] where the corresponding terms "$+x^+$" and "$+y^+$" are missing.

LEMMA 5. *(cf. [94, Lemma 9, (ii)]) Let $z = x + y$. To guarantee p relative bits in z, it suffices to approximate x to $p - z^- + x^+ + 2$ relative bit and y to $p - z^- + y^+ + 2$ relative bits, and perform the addition in $p + 2$ relative bits.*

§26. Absolute Approximation for Multiplication Similarly we can do absolute approximation for multiplication using transformation of precision. However, such transforming might over-estimate the precision we actually need. For example, to guarantee p absolute bits in $z = x \cdot y$, we can transform the operation to guarantee $(p + z^+)$ relative bits in z first, and then we can apply Lemma 4, i.e., we need approximate x and y to $(p + z^+ + 2)$ relative bits. After we transform them to absolute precision, we find that we need to guarantee $(p + z^+ - x^- + 2)$ absolute bits in x and $(p + z^+ - y^- + 2)$ absolute bits in y. Similarly, the relative precision for performing multiplication is $(p + z^+ + 2)$. However, the lemma below gives a better estimation:

LEMMA 6. *(cf. [94, Lemma 8, (ii)]) To approximate $z = x \cdot y$ to within p absolute bits of precision, it is enough to approximate x to within $(p + y^+ + 2)$*

absolute bits of precision, y to within $(p + x^+ + 2)$ absolute bits of precision and perform the multiplication with $p + 2$ bits absolute of precision (or $p + z^+ + 2$ relative bits).

Proof. Let $\delta = 2^{-(p+2)}$, $\delta_x = 2^{-(p+y^+ +2)}$ and $\delta_y = 2^{-(p+x^+ +2)}$. From $|\tilde{x} - x| \leq \delta_x$, we have $|\tilde{x}| \leq |x| + \delta_x$. Hence

$$\begin{aligned}
|\tilde{x} \odot \tilde{y} - x \cdot y| &\leq |\tilde{x} \odot \tilde{y} - \tilde{x} \cdot \tilde{y}| + |\tilde{x}| \cdot |\tilde{y} - y| + |y| \cdot |\tilde{x} - x| \\
&\leq \delta + |\tilde{x}| \cdot \delta_y + |y| \cdot \delta_x \\
&\leq \delta + (|x| + \delta_x) \cdot \delta_y + |y| \cdot \delta_x \\
&= \delta + |x| \cdot \delta_y + |y| \cdot \delta_x + \delta_x \cdot \delta_y \\
&\leq \delta + \delta + \delta + \delta_x \cdot \delta_y \\
&\leq 4\delta \\
&= 2^{-p}.
\end{aligned}$$

where $\delta_x \cdot \delta_y = 2^{-(2p+4+x^+ +y^+)} \leq 2^{-(2p+4+z^+)} \leq 2^{-(p+4)} \leq \delta$ since we assume $p > -z^+$. **Q.E.D.**

It is not hard to see that $p + y^+ + 2 \leq p + z^+ - x^- + 2$. Moreover, this propagation avoids the computation of x^- and y^- which could be costly for some nodes.

§27. Relative Approximation for Division For division, guaranteed relative precision is easier:

LEMMA 7. *(cf. [94, Lemma 10]) To approximate $z = x/y$ to within p relative bits precision, it is enough to approximate x and y to within $p + 2$ bits relative precision and perform the division in $p + 2$ relative bits precision.*

Proof. Let $\delta = 2^{-(p+2)}$. From $|\tilde{x} - x| \leq |x|\delta$, we have $|x|(1-\delta) \leq |\tilde{x}| \leq |x|(1+\delta)$. Similarly we have $|y|(1-\delta) \leq |\tilde{y}| \leq |y|(1+\delta)$ and $|\tilde{x}/\tilde{y}| \leq |x/y| \cdot \frac{1+\delta}{1-\delta}$. Hence

$$\begin{aligned}
|\tilde{x} \oslash \tilde{y} - x/y| &\leq |\tilde{x} \oslash \tilde{y} - \tilde{x}/\tilde{y}| + |\tilde{x}/\tilde{y} - x/y| \\
&\leq |\tilde{x}/\tilde{y}| \cdot \delta + \frac{|\tilde{x} - x| \cdot |y| + |\tilde{y} - y| \cdot |x|}{|y \cdot \tilde{y}|} \\
&\leq |x/y| \cdot \frac{1+\delta}{1-\delta} \cdot \delta + \frac{|x| \cdot \delta \cdot |y| + |y| \cdot \delta \cdot |x|}{|y \cdot y|(1-\delta)} \\
&\leq \left|\frac{x}{y}\right| \cdot \left(\frac{1+\delta}{1-\delta} \cdot \delta + \frac{2\delta}{1-\delta}\right) \\
&\leq \left|\frac{x}{y}\right| \cdot \frac{3+\delta}{1-\delta} \cdot \delta \\
&\leq \left|\frac{x}{y}\right| \cdot (4\delta) \\
&\leq \left|\frac{x}{y}\right| \cdot 2^{-p}
\end{aligned}$$

Since $p \geq 1$, $\delta = 2^{-(p+2)} \leq 1/8 < 1/5$. **Q.E.D.**

§28. Absolute Approximation for Division We give the following lemma for absolute approximation for division:

LEMMA 8. *(cf. [94, Lemma 11]) To approximate $z = x/y$ to within p bits absolute precision, it is enough to approximate x to $p + z^+ - x^- + 2$ bits absolute and y to $p + z^+ - y^- + 2$ bits absolute precision and perform the division in $p + z^+ + 2$ relative bits precision.*

§29. Relative Approximation for k-th Root Extraction For k-th root extraction, we prefer to guarantee relative precision:

LEMMA 9. *(cf. [94, Lemma 12, (i)]) To approximate $z = \sqrt[k]{k}$ to within p relative bits precision, it is enough to approximate x within $p - \lfloor \lg k \rfloor + 1$ relative bits precision and perform the root extraction in $p + 2$ relative bits precision.*

Proof. Let δ_x be the relative error for approximating x and δ be the relative error in root extraction. If $\delta_x \leq 2^{-(p-\lfloor \lg k \rfloor + 1)}$ and $\delta \leq 2^{-(p+2)}$, then we have

$$\begin{aligned}
|\tilde{z} - z| &= |\sqrt[k]{x(1+\delta_x)}(1+\delta) - \sqrt[k]{x}| \\
&= |\sqrt[k]{x}| \cdot |\sqrt[k]{(1+\delta_x)}(1+\delta) - 1| \\
&\leq |\sqrt[k]{x}| \cdot |(1+\delta_x/k)(1+\delta) - 1| \\
&= |\sqrt[k]{x}| \cdot |\delta_x/k + \delta + \delta_x\delta/k| \\
&\leq |\sqrt[k]{x}| \cdot |2^{-(p+1)} + 2^{-(p+2)} + 2^{-(p+1)}2^{-(p+2)}| \\
&\leq |\sqrt[k]{x}| \cdot 2^{-p}.
\end{aligned}$$

Here we use the fact: $\sqrt[k]{(1+\delta_x)} \leq 1 + \delta_x/k$ for $\delta_x < 1$ and $k \geq 1$. **Q.E.D.**

§30. Absolute Approximation for k-th Root Extraction We give the following lemma for absolute approximation for k-th root extraction using precision transformation:

LEMMA 10. *(cf. [94, Lemma 12, (ii)]) To approximate $z = \sqrt[k]{x}$ to within p bits absolute precision, it is enough to approximate x to $p + z^+ - x^- - \lfloor \lg k \rfloor + 1$ bits absolute and perform the root extraction in $p + z^+ + 2$ relative bits precision.*

§31. Propagation Rules Now we show all these propagation rules in the following Table 2.6 and Table 2.7:

	Precision in x	Precision in y	Precision for Operation
$z = x \pm y$	$p + 2$	$p + 2$	$p + 1$
$z = x \cdot y$	$p + y^+ + 2$	$p + x^+ + 2$	$p + 2$
$z = x/y$	$p + z^+ - x^- + 2$	$p + z^+ - y^- + 2$	$p + 2$
$z = \sqrt[k]{x}$	$p + z^+ - x^- - \lceil \lg k \rceil + 1$		$p + 2$

Table 2.6: Propagating Rules in Absolute Precision.

	Precision in x	Precision in y	Precision for Operation
$z = x \pm y$	$p - z^- + x^+ + 2$	$p - z^- + y^+ + 2$	$p + 2$
$z = x \cdot y$	$p + 2$	$p + 2$	$p + 2$
$z = x/y$	$p + 2$	$p + 2$	$p + 2$
$z = \sqrt[k]{x}$	$p - \lceil \lg k \rceil + 1$		$p + 2$

Table 2.7: Propagating Rules in Relative Precision.

2.3 Redesign of `BigFloat` system

In the `Core Library`, underlying the `Expr` package is the `BigFloat` system, the "engine" that performs actual numerical approximation. Hence, the performance of the `BigFloat` system is crucial to the whole system. As we mentioned in Section 2, `BigFloat` uses `GMP` as its big integer kernel (but with its own internal representation) and implements arithmetic operations from scratch. While many research efforts have been made on improving performance of `Expr` level, less attention is paid to improving `BigFloat`. Moreover, as we want to bring transcendental functions into the `Core Library`, this would require us provide those functions in the `BigFloat` class as well.

We do not want to reinvent the wheel but would prefer to use existing implementations. Two multiple-precision floating-point computation libraries, `mpf` in `GMP` and `MPFR`, were both considered for adoption. We finally decided to use `MPFR` because it provides exact rounding and implements more transcendental functions. `MPFR` is also efficient and under active development.

The interface of `BigFloat` in the `Core Library` has certain requirements that are not directly available in `MPFR`. Hence a wrapper is necessary. Furthermore, `BigFloat` in the `Core Library` has two roles: as an exact number ring and as an approximate interval of real numbers. For example, when we study the intersections of curves and surfaces, we are interested in evaluating polynomials at exact `BigFloat` points using only exact ring operations. Another example is Newton iteration. Although it involves division, we do not want to keep track of errors since the Newton method is self-correcting.

To gain greater efficiency, we now split off from `BigFloat` a new class called `BigFloat2`. `BigFloat` is designed to be a C++ wrapper around `MPFR` with some new features, for instance, exact ring operations are implemented for C++ arithmetic operators $+, -, \times$. Since it does not keep track of errors, it can be used efficiently in the applications that we mentioned above. `BigFloat2` is now going to take the role of the old `BigFloat` in providing interval bounds for expression nodes. We use interval arithmetic to maintain errors inside `BigFloat2`. Internally, each `BigFloat2` actually uses two `BigFloat` numbers, which also explains the "2" in the new name.

2.3.1 MPFR Overview

MPFR is a portable C library which provides efficient arbitrary precision arithmetic on floating-point numbers [63, 29, 50]. It uses ideas from the IEEE-754 standard [44] for double-precision floating-point arithmetic. In particular, with a precision of 53 bits, it is able to exactly reproduce all computations with double-precision machine floating-point numbers. It also provides the four rounding mode in the IEEE 754 standard. It has been used in many applications, such as CGAL and iRRAM.

§32. Internal representation of MPFR Internally, a floating-point number f in MPFR is represented by a triple $\langle s, m, e\rangle$ where s is the sign, m is the mantissa with $0.5 \leq m < 1$ and e is the exponent such that

$$f = (-1)^s \cdot m \cdot 2^e.$$

Functions `mpfr_sgn()` and `mpfr_get_exp()` return the sign and the exponent respectively.

§33. Signed zero Zero in MPFR is signed, i.e., there are both $+0$ and -0. However `mpfr_sgn()` will return 0 for either one.

§34. Precision According to [38], "the precision is the number of bits used to represent the mantissa of a floating-pointer number". Extra zeros may appear at the tail of the mantissa. `mpfr_get_prec()` and `mpfr_set_prec()` are used to get and set this precision.

§35. Mantissa in integer form Sometimes we want to manipulate the mantissa directly. But the mantissa in MPFR is a fraction. We are more interested

in the mantissa in integer form. A call of `mpfr_get_z_exp()` for a variable f will return z and e' such that $f = z \cdot 2^{e'}$. However, the returned mantissa z may have powers of 2 as factors. Actually z is scaled by the precision in f, i.e., if $f = (-1)^s \cdot m \cdot 2^e$ has a precision p, then $z = (-1)^s \cdot m \cdot 2^p$ and $e' = e - p$. Thus, if there are n extra zeros at the tail of mantissa m, then z contains a factor of 2^n. It is clear that $2^{p-1} \leq |z| < 2^p$ since $0.5 \leq m < 1$. Thus, the precision p is $\lceil \log_2 z \rceil$.

§36. Rounding mode Four rounding modes, *round to nearest*, *round towards* 0, *round towards* $+\infty$ and *round towards* $-\infty$, are supported in `MPFR`. Under these rounding modes, if f is a `MPFR` variable with the precision p, then the difference between its rounded mantissa and its exact one is less than 2^{-p}.

2.3.2 Design of class `BigFloat`

The class `BigFloat` is designed to a C++ wrapper class on `mpfr_t` provided by `MPFR`. But we enhanced it with some new features.

§37. Relative precision and precision in `MPFR` `MPFR` follows the IEEE standard in its arithmetic: it requires the result of arithmetic operations be exactly rounded. That is, the result must be computed exactly and then rounded to the nearest floating-point number. In particular, if the result can be exactly represented, then this result will be output. So the error in the result depends on the number of bits of its mantissa, i.e., the precision of the result variable. The `Core Library` use relative precision or absolute precision to measure the approximation errors. The precision in `MPFR` is very close to our notion of relative precision. For each arithmetic operations, users are required to set the output

precision manually. For example, a call of `mpfr_mul(c, a, b, GMP_RNDN)` in `MPFR` will compute the product of a and b with rounding to nearest and put the result into c. The number of correct bits of this product depends on the number of bits in the mantissa of c, so users have to set it explicitly before the `mpfr_mul()` call. However, how many bits do we need if we want to guarantee the result has relative precision p? The following lemma shows that $p + 1$ bits is enough:

LEMMA 11. *To guarantee relative p-bit precision in an arithmetic operation in MPFR, it is sufficient that mantissa of the result variable has at least $p + 1$ bits (using any of above four rounding modes).*

Proof. Assume $f = (-1)^s \cdot m \cdot 2^e$ is the real result and $\tilde{f}$ is the result with $p + 1$ bits in mantissa returned by `MPFR`, then

$$\tilde{f} = (-1)^s \cdot \tilde{m} \cdot 2^e$$

and

$$|\tilde{m} - m| = 2^{-(p+1)}.$$

Hence,

$$|\tilde{f} - f| = |\tilde{m} - m|2^e = 2^{-(p+1)}2^e \leq 2^{-p} \cdot m \cdot 2^e = |f| \cdot 2^{-p}$$

since $1/2 \leq m < 1$. **Q.E.D.**

This Lemma shows that the precision used in `MPFR` is basically relative precision (but off by 1 bit) if all bits in the mantissa are valid. It also shows that the type IEEE double which has 53 bits in mantissa (52 bits + one implicit bit) can only guarantee up to relative 52 bits of precision.

§38. Estimation of precision for exact ring operations `MPFR` requires users to specify the precision for each variable. This design is efficient, but sometime inconvenient for users. In our redesigned `BigFloat`, we want to guarantee exactness of ring operations for our overloaded C++ arithmetic operators $(+, -, \times)$.

To achieve this, we estimate the precision for the final results. The following two lemmas are given for such precision estimation:

LEMMA 12 (Addition/Subtraction). *Let* $f_1 = (-1)^{s_1} \cdot m_1 \cdot 2^{e_1}$ *and* $f_2 = (-1)^{s_2} \cdot m_2 \cdot 2^{e_2}$ *be two variables in* `MPFR`*. If* f_1 *has precision* p_1*,* f_2 *has precision* p_2 *and* $\delta = (e_1 - p1) - (e_2 - p2)$*, then for computing* $f = f_1 \pm f_2$*, it is enough to set the precision of* f *to be*

$$\begin{cases} 1 + \max\{p_1 + \delta, p_2\} & \text{if } \delta \geq 0 \\ 1 + \max\{p_1, p_2 - \delta\} & \text{if } \delta < 0 \end{cases}$$

in order to guarantee that all bits in the mantissa of f *correct.*

Proof. We can write f_1 and f_2 with the mantissa in integer forms as follows:

$$f_1 = z_1 \cdot 2^{e_1 - p_1} \quad \text{and} \quad f_2 = z_2 \cdot 2^{e_2 - p_2}.$$

where $\lceil \log_2 |z_1| \rceil = p_1$ and $\lceil \log_2 |z_2| \rceil = p_2$. Then,

$$|f| = |f_1 \pm f_2| \leq |z_1| \cdot 2^{e_1 - p_1} + |z_2| \cdot 2^{e_2 - p_2}$$

If $\delta \geq 0$, then

$$|f| \leq \left(|z_1| 2^{\delta} + |z_2|\right) 2^{e_2 - p_2}.$$

The mantissa z in integer form of f satisfies

$$|z| \leq |z_1| 2^{\delta} + |z_2|$$

Thus, the precision p that we need in f satisfies

$$\begin{aligned} p &= \lceil \log_2 |z| \rceil \\ &\leq 1 + \max\{\lceil \log_2 |z_1| \rceil + \delta, \lceil \log_2 |z_2| \rceil\} \\ &= 1 + \max\{p_1 + \delta, p_2\}. \end{aligned}$$

If $\delta < 0$, then we obtain

$$p \leq 1 + \max\{p_1, p_2 - \delta\}.$$

Q.E.D.

LEMMA 13 (Multiplication). *Let $f_1 = (-1)^{s_1} \cdot m_1 \cdot 2^{e_1}$ and $f_2 = (-1)^{s_2} \cdot m_2 \cdot 2^{e_2}$ be two variables in MPFR. If f_1 has precision p_1, f_2 has precision p_2, then for computing $f = f_1 \cdot f_2$, it is enough to set the precision of f to be*

$$p_1 + p_2$$

in order to guarantee that f can be computed without error.

Proof. As above, we have

$$|f| = |f_1 \cdot f_2| \leq |z_1 \cdot z_2| \cdot 2^{e_1 - p_1 + e_2 - p_2}$$

so the mantissa of f satisfies:

$$|z| \leq |z_1 \cdot z_2|.$$

Hence, $p = \lceil \log_2 |z| \rceil \leq \lceil \log_2 |z_1| \rceil + \lfloor \log_2 |z_2| \rfloor = p_1 + p_2$. **Q.E.D.**

It should be noticed that over-estimation can occur. Even more, the over-estimated precision could grow very fast (see column 2 of Figure 2.5) which will

cause MPFR have to allocate large unnecessary memory space. To avoid this, we provide a function named mpfr_remove_trailing_zeros() to remove the unnecessary tailing zeros (for efficiency, it only removes zeros in chunks). It is called after each corresponding arithmetic operation. The following experiment shows the difference of performing a summation with and without this call:

Experiment 2 Optimization of Precision Estimation

We compute $\sum_{n=1}^{n} i$ using two methods:

Method1: initialize $s = 1$, then for $i = 1, 2, \cdots, n$, increase the precision of s by the precision of i, and then call MPFR to multiply s and i and put the result in s.

Method2: same as Method 1, but call mpfr_remove_trailing_zeros() for s after multiplication in the loop.

A time comparison between them is given in Figure 2.5 (timings are measured in microseconds and the precision is in bits). From this experiment, we can see that without removing trailing zeros, the estimated precision could grow fast and the time for performing corresponding arithmetic operations could be longer.

§39. Variants of Operations in BigFloat We provide four versions for each arithmetic operations:

1. Raw version: functions starting with "r_"; the user has to set the precision explicitly.

2. Fixed version: the last parameter before rounding mode is the precision that the user specified for result variables. It calls set_prec() first, and

$n =$	Estimated Precision	Timing	Optimized Precision	New Timing
100	469	1	32	0
1000	7599	4	32	4
10000	108458	90	32	32
100000	1414677	16305	32	308
1000000			64	3118

Figure 2.5: Timing for computing $\sum_{i=1}^{n} i$ w/ and w/o removing trailing zeros.

then calls the raw version.

3. Auto version: `BigFloat` will estimate a precision for the result, then after computation, adjusts the final precision (i.e., eliminating the tailing zeros using `mpfr_remove_trailing_zeros()`).

4. C++ operators: The overloaded C++ operators will use the auto version, so that users don't have to worry about the precision estimation. However for division, it will use default global precision.

As an example, C++ declarations of four versions for addition operations are shown below:

```
1 int BigFloat::r_add(const BigFloat& x, const BigFloat& y);
2 int BigFloat::add(const BigFloat& x, const BigFloat& y, prec_t);
3 int BigFloat::add(const BigFloat& x, const BigFloat& y);
4 BigFloat operator+(const BigFloat& x, const BigFloat& y);
```

Note that the first 3 functions are member functions of `BigFloat`. So the results of these function calls will be stored in the calling `BigFloat` objects. The returned integer values in these functions show the exactness of the result: a positive value means the result is larger than the exact value, negative means

less than and zero means exactly. The last overloaded + operator is a global function which returns the result as a new object. Hence, it is less efficient than the first 3 functions.

§40. Benchmark of new `BigFloat` By adapting `MPFR`, not only can `BigFloat` provide more arithmetic operations, such as `cbrt()`, elementary functions `sin()`, `cos()` etc., but also the performance of the `BigFloat` has also been greatly improved. We tested the performance of `sqrt()` in the current `Core Library` 1.x and the new implementation based on `MPFR` using the following experiment:

Experiment 3 Comparison of the Performance of `sqrt()`

Computing `sqrt(`i`)` for $i = 2, \ldots, 100$ with precision p using `BigFloat` class in the `Core Library` 1.x and `Core Library` 2.0:

The timing comparison is shown in Figure 2.6. We have about 25 times speedup with the new implementations.

Precision	`Core Library` 1.7	`Core Library` 2.0	Speedup
1000	25	1	25
10000	716	32	22
100000	33270	1299	25

Figure 2.6: Timing Comparisons for `sqrt()`.

2.3.3 Design of class `BigFloat2`

`BigFloat2` is the new number type that we split off from `BigFloat`. It is mostly used by `Expr` to perform the underlying approximation. However, experienced

users can use it directly for efficiency (to avoid building expression DAG). Internally, we use interval arithmetic.

§41. Interval representations To approximate the real numbers, an interval is usually used to represent the approximation of the exact value when it cannot be presented (we use floating-point representation) exactly in computers. There are 3 different approaches:

- $(m \pm err) \cdot 2^e$: mantissa, error bits and exponents parts are maintained and error bits are limited to a certain range. A computation involves one operation on mantissa, a few operations on error bits, then a final normalization is needed in order to maintain error bits within certain ranges. `BigFloat` in the `Core Library` 1.x uses this approach.

- $(d \pm err)$: less restriction on error bits, which can have different exponents. so normalization is not necessary for each operation. `REAL` in `iRRAM`[65] uses this approach.

- (d_l, d_r): two multiple precision numbers are used to represent an interval, and interval arithmetic is used for all kinds of arithmetic operations.

We use (d_l, d_r) as our new approach for `BigFloat2` because this implementation is simpler and the approximation interval is more precise. Another motivation is that `MPFR` has provided the necessary rounding modes for each arithmetic operations and has already implemented many elementary functions.

§42. Interval Arithmetic `BigFloat2` maintains an interval $[\underline{x}, \overline{x}]$ for a real number x where $\underline{x}$ and $\overline{x}$ are `BigFloat` numbers. It uses interval arithmetic

to implement its arithmetic operations. The standard rules for some basic operations are shown in Table 2.8 [1, 62]:

z	$\underline{z}$	$\overline{z}$
$z = x + y$	$\underline{x} + \underline{y}$	$\overline{x} + \overline{y}$
$z = x - y$	$\underline{x} - \overline{y}$	$\overline{x} - \underline{y}$
$z = x \times y$	$\min\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}$	$\max\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}$
$z = x \div y$	$\min\{\underline{x}/\underline{y}, \underline{x}/\overline{y}, \overline{x}/\underline{y}, \overline{x}/\overline{y}\}$	$\max\{\underline{x}/\underline{y}, \underline{x}/\overline{y}, \overline{x}/\underline{y}, \overline{x}/\overline{y}\}$
$z = \sqrt[k]{x}$	$\sqrt[k]{\underline{x}}$	$\sqrt[k]{\overline{x}}$

Table 2.8: Rules for interval arithmetic.

Multiplication and division involve 4 arithmetic operations and 3 comparisons if we use the rules in the fourth and fifth row for multiplication and division. In practice, we can reduce them to fewer operations as shown below in Table 2.9 and Table 2.10:

2.4 Extending `Expr` class

While our `Expr` class provides four arithmetic operations and radical operations which can satisfy most applications, there is a need for supporting new operations, for example, transcendental functions.

2.4.1 How to Add Your Own Operation for `Expr`

As we mentioned in the Section 2.1, `Expr` is a wrapper around `ExprRep`. Adding a new operation for `Expr` requires one to design a new type of `ExprRep` node, i.e., design a subclass of `ExprRep`. In the `Core Library` 1.x, it is possible for users

x	y	min	max
$0 \le \underline{x} \le \overline{x}$	$0 \le \underline{y} \le \overline{y}$	$\underline{xy}$	$\overline{xy}$
$0 \le \underline{x} \le \overline{x}$	$\underline{y} \le \overline{y} \le 0$	$\overline{x}\underline{y}$	$\underline{x}\overline{y}$
$0 \le \underline{x} \le \overline{x}$	$\underline{y} \le 0 \le \overline{y}$	$\overline{x}\underline{y}$	$\overline{xy}$
$\underline{x} \le \overline{x} \le 0$	$0 \le \underline{y} \le \overline{y}$	$\underline{x}\overline{y}$	$\overline{x}\underline{y}$
$\underline{x} \le \overline{x} \le 0$	$\underline{y} \le \overline{y} \le 0$	$\overline{xy}$	$\underline{xy}$
$\underline{x} \le \overline{x} \le 0$	$\underline{y} \le 0 \le \overline{y}$	$\underline{x}\overline{y}$	$\underline{xy}$
$\underline{x} \le 0 \le \overline{x}$	$0 \le \underline{y} \le \overline{y}$	$\underline{x}\overline{y}$	$\overline{xy}$
$\underline{x} \le 0 \le \overline{x}$	$\underline{y} \le \overline{y} \le 0$	$\overline{x}\underline{y}$	$\underline{xy}$
$\underline{x} \le 0 \le \overline{x}$	$\underline{y} \le 0 \le \overline{y}$	$\min\{\overline{x}\underline{y}, \underline{x}\overline{y}\}$	$\max\{\underline{xy}, \overline{xy}\}$

Table 2.9: Rules for multiplication using interval arithmetic.

to design their own `Expr` nodes. However, the procedure is complicated and error-prone. Now, adding a new type of `Expr` node becomes easier. Basically, it involves the following steps:

1. Derive a class from `ExprRepT` or a subclass of `ExprRepT` (see Figure 2.3 for pre-defined subclasses).

2. Implement a constructor for this new class and call the following two functions:

   `compute_filter()`, `compute_numtype()`

   to compute the filter value and set the number type.

3. Implement the following functions:

   `compute_filter()`, `compute_numtype()`,

   `compute_sign()`, `compute_uMSB()`, `compute_lMSB()`,

x	y	min	max
$\underline{x} \leq \overline{x}$	$\underline{y} = \overline{y} = 0$	NaN	NaN
$\underline{x} \leq \overline{x}$	$\underline{y} \leq 0 \leq \overline{y}$	$-\infty$	$+\infty$
$0 \leq \underline{x} \leq \overline{x}$	$0 \leq \underline{y} \leq \overline{y}$	$\underline{x}/\overline{y}$	$\overline{x}/\underline{y}$
$0 \leq \underline{x} \leq \overline{x}$	$\underline{y} \leq \overline{y} \leq 0$	$\overline{x}/\overline{y}$	$\underline{x}/\underline{y}$
$\underline{x} \leq \overline{x} \leq 0$	$0 \leq \underline{y} \leq \overline{y}$	$\underline{x}/\underline{y}$	$\overline{x}/\overline{y}$
$\underline{x} \leq \overline{x} \leq 0$	$\underline{y} \leq \overline{y} \leq 0$	$\overline{x}/\underline{y}$	$\underline{x}/\overline{y}$
$\underline{x} \leq 0 \leq \overline{x}$	$0 < \underline{y} \leq \overline{y}$	$\underline{x}/\underline{y}$	$\overline{x}/\underline{y}$
$\underline{x} \leq 0 \leq \overline{x}$	$\underline{y} \leq \overline{y} < 0$	$\overline{x}/\overline{y}$	$\underline{x}/\overline{y}$

Table 2.10: Rules for division using interval arithmetic.

`compute_r_approx()`, `compute_a_approx()`, `compute_rootbd()`

4. Define a function (can be global function/operator or constructor/member functions/operator in `Expr`) which can construct this new type node.

Now we follow these steps to show how to add a new operation for `Expr`. Say, we want to add the following "neg" function for `Expr`:

```
1 Expr neg(const Expr& e);
```

It will take one `Expr` parameter e and construct a new `Expr` node with the negated value $-e$.

We first derive a new class, `NegRepT`, from `ExprRepT`. For efficiency, `ExprRepT` uses a reference counting technique, so each `ExprRepT` has a reference counter inside. If we directly derive our class from `ExprRep`, it will require us to maintain the reference counter. The three directly derived class `ConstRepT`, `UnaryOpRepT`, and `BinaryOpRepT` handle the reference counting automatically.

So we derive our new class from `UnaryOpRepT` since negation operation is a "unary" operation:

```
1  template <typename T>
2  class NegRepT : public UnaryOpRepT<T> {
3  };
```

Then we add a constructor that takes one parameter of an `ExprRepT` pointer (pointing to its child node). In the constructor initializer, we pass the pointer parameter to the parent class `UnaryOpRepT`. We compute the filter value and set the number type inside the constructor.

```
1    NegRepT(ExprRepT<T>* c) : UnarayOpRepT<T>(c) {
2      compute_filter();
3      compute_numtype();
4    }
```

The next step is to implement the required functions[3]:

```
1    // functions for computing filter and number type
2    void compute_filter() const {
3      filter().neg(child->filter());
4    }
5    void compute_numtype() const {
6      _numType = child->_numType;
7    }
8
9    // virtual functions for computing sign, uMSB, lMSB
10   virtual bool compute_sign() const {
11     sign() = -child->sign(); return true;
12   }
13   virtual bool compute_uMSB() const {
14     uMSB() = child->uMSB(); return true;
15   }
16   virtual bool compute_lMSB() const {
17     lMSB() = child->lMSB(); return true;
```

[3]Note that `compute_filter()` and `compute_numtype()` are not virtual functions since we cannot call virtual functions in the constructor of the base class.

```
18    }
19
20    // virtual functions for r_approx, a_approx
21    virtual void compute_r_approx(prec_t prec) const {
22      kernel().neg(child->r_approx(prec), prec);
23    }
24    virtual bool compute_a_approx(prec_t prec) const {
25      kernel().neg(child->a_approx(prec), abs2rel(prec));
26    }
27
28    // virtual function for computing root bound
29    virtual void compute_rootbd() const {
30      rootbd().neg(child->rootbd());
31    }
```

Now we have `NegRepT` class. The last step is to define the negation function. It just constructs our new designed `NegRepT` node:

```
1  template <typename T>
2  ExprT<T> neg(const ExprT<T>& e) {
3    return new NegRepT<T>(e.rep());
4  }
```

Remarks:

1. We do not have to implement all functions in step 3. For `compute_filter()`, users can just call `filter().invalidate()` if the users do not know how to obtain an filter value. For `compute_sgn()`, `compute_uMSB()`, and `compute_lMSB()`, users can just return `false` when it is not applicable. In that case, `ExprRepT` will get those information from the approximation. As we mentioned before, users do not have to implement both `compute_r_approx()` and `compute_a_approx()`. However, if the users only implement `compute_r_approx()`, then users need provide a valid `compute_uMSB()`; if the users only implement `compute_a_approx()`, `compute_lMSB()` is needed as well.

2. We must implement `compute_numtype()` since `ExprRepT` need it to classify each node. Moreover, in case the expression node is algebraic, we must implement `compute_rootbd()`.

3. We can put the implementations of `compute_filter()` and `compute_numtype()` into the constructor directly without defining separated functions.

2.4.2 Adding Your Own Operation using Pre-defined Macros

Although now adding a new operation for `Expr` is easier and straightforward, users need to know all the details of implementing in C++. For instance, they need to know the exact signatures for each functions that they are required to implement. In order to help the `Core Library` developers and other advanced users, a new mechanism is designed to simplify such task. We provide a set of pre-defined C++ macros to hide some implementation details from the users.

To see how these macros works, we still use the `NegRepT` in previous section as an example. With pre-defined macros, we can simply define `NegRepT` as follows:

```
1 BEGIN_DEFINE_UNARY_NODE(NegRepT)
2
3 END_DEFINE_UNARY_NODE
```

`BEGIN_DEFINE_UNARY_NODE` and `END_DEFINE_UNARY_NODE` are two C++ macros which are defined as follows:

```
1 #define BEGIN_DEFINE_UNARY_NODE(cls_name)                \
2   template <typename T>                                  \
3   class cls_name : public UnaryOpRepT<T> {               \
4   public:                                                \
5     cls_name(ExprRepT<T>* c) : UnarayOpRepT<T>(c) {      \
```

```
6         compute_filter();                                    \
7         compute_numtype();                                   \
8       }
9
10  #define END_DEFINE_UNARY_NODE };
```

Hence, once our new implementation of `NegRepT` are compiled by C++ compilers, those macros are first expanded by **cpp** (C/C++ preprocessor provided by every C/C++ compilers) into:

```
1  template <typename T>
2  class NegRepT: public UnaryOpRepT<T> {
3  public:
4    NegRepT(ExprRepT<T>* c) : UnarayOpRepT<T>(c) {
5      compute_filter();
6      compute_numtype();
7    }
8  };
```

Similarly, we define the macro `DEFINE_UNARY_FUNCTION()` in the `Core Library` as:

```
1  #define DEFINE_UNARY_FUNCTION(fun_name, cls_name)      \
2  template <typename T>                                  \
3  Expr<T> fun_name(const Expr<T>& e)                     \
4  {  return new cls_name<T>(e.rep()); }
```

With this macro, we can define our negation function in just one line:

```
1  DEFINE_UNARY_FUNCTION(neg, NegRepT)
```

A few other macros are pre-defined. See the source code of the `Core Library` 2.0 for details. With these new macros, we can now implement `NegRepT` as:

```
1  BEGIN_DEFINE_UNARY_NODE(NegRepT)
2    // define rule for filter computation
3    BEGIN_DEFINE_RULE(filter)
4      filter().neg(child->filter());
5    END_DEFINE_RULE
```

```
6     // define rule for setting number type
7     BEGIN_DEFINE_RULE(numtype)
8       _numType = child->_numType;
9     END_DEFINE_RULE
10
11    // define rules for computing sign, uMSB, lMSB
12    BEGIN_DEFINE_RULE(sign)
13      sign() = -child->sign(); return true;
14    END_DEFINE_RULE
15    BEGIN_DEFINE_RULE(uMSB)
16      uMSB() = child->uMSB(); return true;
17    END_DEFINE_RULE
18    BEGIN_DEFINE_RULE(lMSB)
19      lMSB() = child->lMSB(); return true;
20    END_DEFINE_RULE
21
22    // define rules for r_approx, a_approx
23    BEGIN_DEFINE_RULE(r_approx)
24      kernel().neg(child->r_approx(prec), prec);
25    END_DEFINE_RULE
26    BEGIN_DEFINE_RULE(a_approx)
27      kernel().neg(child->a_approx(prec), abs2rel(prec));
28    END_DEFINE_RULE
29
30    // define rules for computing root bound
31    BEGIN_DEFINE_RULE(rootbd)
32      rootbd().neg(child->rootbd());
33    END_DEFINE_RULE
34  END_DEFINE_UNARY_NODE
35
36  DEFINE_UNARY_FUNCTION(neg, NegRepT)
```

They will be expanded to the exactly same code that we designed in the previous section. We can see that these pre-defined macros simplify the work of extending `Expr`.

2.4.3 Summation operation for `Expr`

The DAG constructed in `Expr` could be very large in some computation. However, when these expressions are defined using simple recursive rules, we can avoid the explicit construction. This can lead to considerable speedup. For example, computing harmonic series $\sum_{i=1}^{n} \frac{1}{i}$ can be done using the following function:

```
1 Expr harmonic(int n) {
2   Expr r(0);
3   for (int i=1; i<=n; ++i)
4     r = r + Expr(1)/Expr(i);
5   return r;
6 }
```

This function will build an unbalanced DAG of depth n. We did the following experiment to test the performance of this function:

Experiment 4 Computing harmonic series using loop

Computing harmonic series $\sum_{i=1}^{n} \frac{1}{i}$ using above `harmonic()` function for different n.

The timing (in microseconds) for this experiment is given in the second column in Figure 2.7. The segmentation fault is caused by stack overflow in most systems. However, we find an optimization can be done: instead of using $n-1$ binary operation (+) nodes, we can introduce a single "anary" node to do the entire summation in one level.

With the new mechanism we described in previous section, a new type of node called `Sum` can be easily implemented as follows:

```
1 BEGIN_DEFINE_ANARY_NODE(Sum)
2   // define rule for filter computation
3   BEGIN_DEFINE_RULE(filter)
```

```
4       filter().set(children[0]->filter());
5       for (size_t i=1; i<children.size(); ++i)
6         filter().add(filter(), children[i]->filter());
7     END_DEFINE_RULE
8     // define rule for setting number type
9     BEGIN_DEFINE_RULE(numtype)
10      _numType = children[0]->_numType
11      for (size_t i=1; i<children.size(); ++i)
12        _numType = std::max(_numType, children[i]->_numType);
13    END_DEFINE_RULE
14
15    // define rules for computing sign, uMSB, lMSB
16    BEGIN_DEFINE_RULE(sign)
17      return false;
18    END_DEFINE_RULE
19    BEGIN_DEFINE_RULE(uMSB)
20      uMSB() = children[0]->uMSB();
21      for (size_t i=1; i<children.size(); ++i)
22        uMSB() = std::max(uMSB(), children[i]->uMSB()) + 1;
23      return true;
24    END_DEFINE_RULE
25    BEGIN_DEFINE_RULE(lMSB)
26      lMSB() = children[0]->lMSB();
27      for (size_t i=1; i<children.size(); ++i)
28        lMSB() = std::max(lMSB(), children[i]->lMSB());
29      return true;
30    END_DEFINE_RULE
31
32    // define rules for r_approx, a_approx
33    BEGIN_DEFINE_RULE(r_approx)
34      int n = ceillg(children.size()) + 1;
35      kernel().set(0);
36      for (size_t i=0; i<children.size(); ++i)
37        kernel().add(kernel(),
38          children[i]->r_approx(prec+n), prec+n);
39    END_DEFINE_RULE
40    BEGIN_DEFINE_RULE(a_approx)
41      int n = ceillg(children.size()) + 1;
42      kernel().set(0);
```

```
43      for (size_t i=0; i<children.size(); ++i)
44        kernel().add(kernel(),
45          children[i]->a_approx(prec+n), abs2rel(prec+n));
46    END_DEFINE_RULE
47
48    // define rules for computing root bound
49    BEGIN_DEFINE_RULE(rootbd)
50      rootbd().set(0L);
51      for (size_t i=1; i<children.size(); ++i)
52        rootbd().add(rootbd(), child->rootbd());
53    END_DEFINE_RULE
54  END_DEFINE_ANARY_NODE
55
56  DEFINE_ANARY_FUNCTION(sum, Sum)
```

With this new operation, we can rewrite the routine `harmonic()`:

```
1  Expr harmonic_term(int i) {
2    return Expr(1)/Expr(i);
3  }
4  Expr harmonic(int n) {
5    return sum(harmonic_term, 1, n);
6  }
```

A new experiment with the redesigned `harmonic()` function is shown below:

Experiment 5 Computing harmonic series using `sum()`

Computing harmonic series $\sum_{i=1}^{n} \frac{1}{i}$ using above redesigned `harmonic()` function for different n.

The timing (in microseconds) for this new implementation is shown in the third column in Figure 2.7 (see `t_sum.cpp` under `benchmark/sum`). We can see with the new implemented `sum()` operations, we can compute the summation for larger n and the new operation speeds up 3-58 times.

n=	Using Loops	Using `sum()`	Speedup
1000	24	7	3.4
10000	3931	67	58.6
100000	(segmentation fault)	752	N/A
1000000	(segmentation fault)	12260	N/A

Figure 2.7: Timing for computing harmonic series.

2.4.4 Transcendental Node π

Now we present our *first* transcendental node π. π is a constant, so we design it to be a leaf node and derive it from `ConstRepT`:

```
1  template <typename T>
2  class PiRepT: public ConstRepT<T> {
3  public:
4    PiRepT() : ConstRepT<T>() {
5      compute_filter();
6      compute_numtype();
7    }
8    // functions for computing filter and number type
9    void compute_filter() const {
10     filter().set(3.14, true); //true means the value is inexact
11   }
12   void compute_numtype() const
13   { _numType = NODE_NT_TRANSCENDENTAL; }
14
15   // virtual functions for computing sign, uMSB, lMSB
16   virtual bool compute_sign() const
17   { sign() = 1; return true; }
18   virtual bool compute_uMSB() const
19   { uMSB() = 2; return true; }
20   virtual bool compute_lMSB() const
21   { lMSB() = 1; return true; }
22
```

```
23    // virtual functions for r_approx, a_approx
24    virtual void compute_r_approx(prec_t prec) const
25    { kernel().pi(prec); }
26    virtual bool compute_a_approx(prec_t prec) const
27    { kernel().pi(abs2rel(prec)); }
28  };
```

A global function in `Expr` can be:

```
1  template <typename T>
2  ExprT<T> pi()
3  { return new PiRepT<T>(); }
```

2.5 Benchmarks

§43. `compare.cpp` Figure 2.8 shows the timing of comparing

$$\sqrt{x}+\sqrt{y} \quad \text{and} \quad \sqrt{x+y+2\sqrt{xy}}$$

where x and y are rational numbers with various bit length L. Our redesigned

bit length L	`Core Library` 1.7	`Core Library` 2.0	Speedup
1000	0.82	0.59	1.4
2000	6.94	1.67	4.2
8000	91.9	11.63	7.9
10000	91.91	30.75	3.0

Figure 2.8: Timing Comparisons for `compare.cpp`

library speeds up about 2-8 times.

§44. `testFilter.cpp` Figure 2.9 and Figure 2.10 shows the timing of computing the determinants of a set of matrices with the filter facility on and off (see

`testFilter.cpp` under `benchmark/testFilter`). The numbers in first column in each table has the following format:

$$N \times d \times b$$

where N is the number of matrices, d is the dimension of each matrix and b is the bit length of each entry in the matrix (entries are rationals).

MATRIX	1.7 w/ filter	2.0 w/ filter	Speedup
1000x3x10	9	19	0.5
1000x4x10	26	43	0.6
500x5x10	449	204	2.2
500x6x10	1889	597	3.2
500x7x10	4443	1426	3.1
500x8x10	8100	2658	3.0

Figure 2.9: Timing `testFilter.cpp` w/ filter (in microseconds)

MATRIX	1.7 w/o filter	2.0 w/o filter	Speedup
1000x3x10	621	232	2.7
1000x4x10	1666	530	3.1
500x5x10	1728	488	3.5
500x6x10	3493	894	3.9
500x7x10	6597	1580	4.2
500x8x10	11367	2820	4.0

Figure 2.10: Timing `testFilter.cpp` w/o filter (in microseconds)

2.6 *InCore*: an Interactive `Core Library`

While the `Core Library` provides a very simple programming interface, it still requires users to have C++ programming experience. Moreover since the `Core Library` is written in C++, the library itself as well as users' programs have to been compiled first before they are run. The compilation is slow. It is even slower because of C++ template classes. Some users will be more interested in using the `Core Library` for prototyping and testing, so an interpreted version like `Maple` can be very useful.

Python is an interpreted language with an object oriented design. It is very easy to use. It provides an interface for wrapping C/C++ codes.

§45. Tools for Writing Python Binding To wrap C++ code, there are different solutions [78, 79, 7]:

Python/C API: Official C APIs for writing Python extension.

SWIG: Uses interfaces file to generate wrapper codes automatically. It can generate wrapper codes for other script languages such as Perl, Tcl/Tk and Ruby.

Boost.Python: a C++ library which enables seamless interoperability between C++ and Python.

SIP: a tool for automatically generating Python bindings for C and C++ libraries. SIP was originally developed in 1998 for PyQt —the Python bindings for the Qt GUI toolkit—but is suitable for generating bindings for any C or C++ library.

We developed our first Python binding using SWIG. However, we found the speed was slow because it generated a very big Python interface program to call underlying C++ codes. We tried Boost.Python as well, but had some problems with function and operator overloading and the compilation time for building binding was long. We finally turned to SIP. It is easy to use and has very good performance [34] although the documentation is sparse.

InCore is the name of our Python binding for our `Core Library`. It exposes `Core Library`'s classes such as `BigInt`, `BigRat`, `BigFloat`, `Expr`, so that the users can use them as Python objects in Python programs while providing the same features as our C++ library. Here is an example of a Python program using *InCore*. It calls Python's built-in math functions and our `Core Library` functions (exposed via *InCore*) to test the following two identities:

$$\sqrt{2} \cdot \sqrt{3} - \sqrt{6} = 0$$

and

$$\sqrt{1+y^2} - y - \frac{1}{\sqrt{1+y^2}+y} = 0$$

for $y = 5000, 5001, \ldots, 5010$.

```
1  import corelib
2  import math
3
4  def simpleTest(fn, cls, x, y, z):
5    "test x*y-z"
6    name = fn.__name__
7    result = fn(cls(x))*fn(cls(y))-fn(cls(z))
8    print "%s(%d) * %s(%d) - %s(%d) = " % (name, x, name, y, name, z), result
9
10 def kahan(fn, cls, first, last):
11   "test (sqrt(1+y*y)-y) - 1/(sqrt(1+y*y)+y)"
12   for n in range(first, last):
13     y = cls(n)
```

```
14      yy = y * y
15      g = (fn(yy+1) - y) - (1/(y + fn(yy+1)))
16      print "n = %d; G(n) = " % n, g
17
18  def testFloating():
19    print "Test Python Floating point computation:"
20    simpleTest(math.sqrt, int, 2, 3, 6)
21    kahan(math.sqrt, int, 5000, 5010)
22
23  def testCore():
24    print "Test Core Library Level 3 computation:"
25    simpleTest(corelib.sqrt, corelib.Expr, 2, 3, 6)
26    kahan(corelib.sqrt, corelib.Expr, 5000, 5010)
27
28  if __name__ == "__main__":
29    testFloating()
30    testCore()
```

Currently, a Python binding for `CGAL` is also under development. It uses Boost.Python.

Chapter 3

Absolute Approximation of General Hypergeometric Function

The redesigned `Core Library` provides a better mechanism for integrating transcendental functions. We need algorithms to approximate such functions to absolute precision since we do not have root bounds for them. Many important results were obtained three decades ago by Brent (e.g., [10, 11]). In particular, he showed that all the common elementary functions can be approximated efficiently using Newton-like schemes.

Most common elementary functions [64] are hypergeometric functions. In this chapter, we present our research effort on the general hypergeometric functions. We show that the absolute approximation of the general hypergeometric function $H(\mathbf{a};\mathbf{b};x) = {}_pF_q(\mathbf{a};\mathbf{b};x)$ is solvable. An explicit bound for the complexity of our algorithm is given. We further address the problem of evaluating H when x is a "blackbox number", i.e., the number x is represented by a pro-

cedure that can return an approximation $\widetilde{x}$ to any desired absolute precision. This generalization allows us to extend our approximability results to most of the familiar transcendental functions of classical analysis that can be derived from H. In particular, this solves the so-called Table Maker's Dilemma [51] for such functions.

Overview of this chapter In Section 3.1, we give basic facts about hypergeometric functions. In Section 3.2, we show that H is absolutely approximable. Section 3.3 shows that H is absolutely approximable when x is a black box number. An explicit bound on the complexity of H is given in Section 3.4. In section 3.5, we address the problems of argument reduction, parameter preprocessing and mathematical constants for efficient evaluation. Some details of the implementation and integration of hypergeometric function package are presented in Section 3.6. In Section 3.7, we give our final remarks and propose some open problems.

3.1 Hypergeometric series and functions

3.1.1 Hypergeometric series

A function $r(n)$ is a **rational function** of n if there are polynomials $p(n)$ and $q(n)$ with

$$r(n) = \frac{p(n)}{q(n)}.$$

Consider a sequence with terms t_n and the associated series

$$\sum_{n=0}^{\infty} t_n.$$

If

$$\frac{t_{n+1}}{t_n} = x,$$

then t_n is called a **geometric term** and $\sum_n t_n$ a **geometric series**.

If

$$\frac{t_{n+1}}{t_n} = r(n)x$$

where $r(n)$ is a rational function, then t_n is called a **hypergeometric term** and $\sum_n t_n$ a **hypergeometric series**. So hypergeometric term and series are a generalization of geometric term and series, respectively. See the second column of Table 3.1 for some examples.

A given term and its corresponding series can be **normalized**: (1) by renumbering the terms such that the index of the first nonzero term is 0, (2) by factoring out the first term such that the new first term is equal 1. After this normalization, every geometric series can be written in its normal form

$$\sum_{n=0}^{\infty} x^n$$

and every hypergeometric series can be written in its normal form

$$\sum_{n=0}^{\infty} \frac{(a_1)_n \cdots (a_p)_n}{(b_1)_n \cdots (b_q)_n n!} x^n$$

where the rising factorial or Pochhammer symbol $(a)_k$ is given by $(a)_k = a(a+1)(a+2)\cdots(a+k-1)$ for $k \geq 1$ and $(a)_0 = 1$.

The above normal form of hypergeometric series may be denoted by the following F**-notation**:

$${}_pF_q(a_1, \cdots, a_p; b_1, \cdots, b_q; x).$$

Thus, a hypergeometric series is completely defined by the sequences $\mathbf{a} = (a_1, a_2, \ldots, a_p)$ and $\mathbf{b} = (b_1, b_2, \ldots, b_q)$. These are called the **upper** and **lower parameters** of the hypergeometric series. x is called the **argument**.

Note the presence of the parameter $1(n! = (1)_n)$ in the denominator; this is a standard convention. Many common hypergeometric series have this factor; but in case denominator does not naturally contain such a factor, we artificially attach this factor to numerator and denominator and denote it as b_0.

3.1.2 Hypergeometric function and convergence

A **hypergeometric function** is a hypergeometric series in which the evaluation point is a variable. It is defined only for those x in which the hypergeometric series converges. If one of the upper parameters is a non-positive integer, then the hypergeometric series has finite number of terms and the hypergeometric function become a polynomial. Otherwise the convergence is given by:

number of parameter	convergence radius
$p < q + 1$	∞
$p = q + 1$	1
$p > q + 1$	0

The interesting case is $p = q + 1$. Think of 1 as a hidden lower parameter, this means that the number of upper parameters is equal to the number of lower parameters. In the case of $p = q + 1$ the convergence depends on x and on the value of

$$s := \sum_{k=1}^{q} b_k - \sum_{k=1}^{p} a_k$$

called the **parameter excess** ([3]). The following Lemma 14 shows their relations.

LEMMA 14. *Let* $\mathbf{a} = (a_1, a_2, \ldots, a_p)$ *and* $\mathbf{b} = (b_1, b_2, \ldots, b_q)$ *where* $p, q \leq 0$. *If* $p = q + 1$, *then the hypergeometric series*

$$\sum_{n=0}^{\infty} \frac{(a_1)_n \cdots (a_p)_n}{(b_1)_n \cdots (b_q)_n n!} x^n$$

is convergent for $|x| < 1$ *and divergent for* $|x| > 1$. *When* $x = 1$, *it converges if* $s > 0$; *when* $x = -1$, *it is absolutely convergent when* $s > 0$, *and convergent but not absolutely when* $-1 \leq s \leq 0$, *where* s *is the parameter excess defined above.*

Proof. The term ratio of the series is

$$\frac{t_{n+1}}{t_n} = \frac{(a_1 + n) \cdots (a_p + n)}{(b_1 + n) \cdots (b_q + n)(1 + n)} x = \frac{(1 + a_1/n) \cdots (1 + a_p/n)}{(1 + b_1/n) \cdots (1 + b_q/n)(1 + 1/n)} x$$

so that as $n \to \infty$, the ratio

$$\left| \frac{t_{n+1}}{t_n} \right| \to |x|.$$

By D'Alembert's test, the series is convergent for all values of x, real or complex such that $|x| < 1$, and divergent for all values of x, real or complex, such that $|x| > 1$. When $|x| = 1$,

$$\begin{aligned} \left| \frac{t_{n+1}}{t_n} \right| &= \left| \left\{ 1 + \frac{\sum_{k=1}^{p} a_k}{n} + O(1/n^2) \right\} \left\{ 1 - \frac{1 + \sum_{k=1}^{q} b_k}{n} + O(1/n^2) \right\} \right| \\ &= \left| 1 - \frac{1 + \sum_{k=1}^{q} b_k - \sum_{k=1}^{p} a_k}{n} + O(1/n^2) \right| \\ &= \left| 1 - \frac{1 + s}{n} + O(1/n^2) \right| \end{aligned}$$

Thus, when $x = 1$, by Raabe's test, it converges if $s > 0$. When $x = -1$, the series is absolutely convergent when $s > 0$, and convergent but not absolutely when $-1 \leq s \leq 0$. **Q.E.D.**

From now on, we assume that $p \leq q + 1$ and $|x| \leq 1$ if $p = q + 1$.

3.1.3 Elementary Functions in Hypergeometric Form

Table 3.1 lists the hypergeometric series representation of some elementary functions. For each function, we list (i) the usual power series representation of that function, (ii) the ratio t_{k+1}/t_k between two consecutive terms of that power series, and (iii) the corresponding hypergeometric series. In some cases, the first term t_0 of the power series is not 1, and has to be factored out from the series (e.g., $\sin x, \arcsin x$).

Table 3.1: Some elementary functions in terms of hypergeometric series.

Functions	Power series	Ratio t_{k+1}/t_k	Hypergeometric series
$\exp(x)$	$\sum_{k=0}^{\infty} \frac{x^k}{k!}$	$\frac{1}{(k+1)}x$	${}_0F_0(;;x)$
$\mathrm{erf}(x)$	$\frac{2x}{\sqrt{\pi}} \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)k!}x^{2k}$	$\frac{(k+\frac{1}{2})}{(k+\frac{3}{2})(k+1)}(-x^2)$	$\left(\frac{2x}{\sqrt{\pi}}\right) {}_1F_1(\frac{1}{2};\frac{3}{2};-x^2)$
$\sin(x)$	$\sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)!}x^{2k+1}$	$\frac{1}{(k+\frac{3}{2})(k+1)}(-\frac{x^2}{4})$	$x \cdot {}_0F_1(;\frac{3}{2};\frac{-x^2}{4})$
$\cos(x)$	$\sum_{k=0}^{\infty} \frac{(-1)^k}{(2k)!}x^{2k}$	$\frac{1}{(k+\frac{1}{2})(k+1)}(-\frac{x^2}{4})$	${}_0F_1(;\frac{1}{2};\frac{-x^2}{4})$
$\arcsin(x)$	$\sum_{k=0}^{\infty} \frac{(2k)!}{2^{2k}(2k+1)(k!)^2}x^{2k+1}$	$\frac{(k+\frac{1}{2})^2}{(k+\frac{3}{2})(k+1)}x^2$	$x \cdot {}_2F_1(\frac{1}{2},\frac{1}{2};\frac{3}{2};x^2)$
$\arctan(x)$	$\sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}x^{2k+1}$	$\frac{(k+\frac{1}{2})}{(k+\frac{3}{2})}(-x^2)$	$x \cdot {}_2F_1(\frac{1}{2},1;\frac{3}{2};-x^2)$
$\log(1+x)$	$\sum_{k=0}^{\infty} \frac{(-1)^k}{k+1}x^{k+1}$	$\frac{(k+1)(k+1)}{(k+2)(k+1)}(-x)$	$x \cdot {}_2F_1(1,1;2;-x)$
$\mathrm{ellipticK}(x)$	$\frac{\pi}{2} \sum_{k=0}^{\infty} \left[\frac{(2k-1)!!}{(2k)!!}\right]^2 x^{2k}$	$\frac{(k+\frac{1}{2})(k+\frac{1}{2})}{(k+1)(k+1)}x^2$	$\frac{\pi}{2} \cdot {}_2F_1(\frac{1}{2},\frac{1}{2};1;x^2)$
$\mathrm{ellipticE}(x)$	$\frac{\pi}{2}\left\{1 - \sum_{k=0}^{\infty}\left[\frac{(2k-1)!!}{(2k)!!)}\right]^2 \frac{x^{2k}}{2k-1}\right\}$	$\frac{(k-\frac{1}{2})(k+\frac{1}{2})}{(k+1)(k+1)}x^2$	$\frac{\pi}{2} \cdot {}_2F_1(-\frac{1}{2},\frac{1}{2};1;x^2)$
$\mathrm{erfi}(x)$	$\frac{2xe^{x^2}}{\sqrt{\pi}} \sum_{k=0}^{\infty} \frac{(-1)^k 2^k}{(2k+1)!}x^{2k}$	$\frac{(k+1)}{(k+\frac{3}{2})(k+1)}(-x^2)$	$\left(\frac{2xe^{x^2}}{\sqrt{\pi}}\right) {}_1F_1(1;\frac{3}{2};-x^2)$
$\mathrm{dilog}(x)$	$\sum_{k=0}^{\infty} \frac{x^k}{k^2}$	$\frac{k^2}{(k+1)(k+1)}x$	${}_2F_2(0,0;1,1;x)$
x^y	$\sum_{k=0}^{\infty} \frac{(k-y)!}{(-y)!k!}(1-x)^k$	$\frac{k-y}{k+1}(1-x)$	${}_1F_0(-y;;1-x)$
$\mathrm{BesselJ}[p,x]$	$\frac{1}{p!}(\frac{x}{2})^p \sum_{k=0}^{\infty} \frac{(-1)^k p!}{k!(k+p)!}(\frac{x}{2})^{2k}$	$\frac{1}{(k+p+1)(k+1)}(-\frac{x^2}{4})$	$\frac{1}{p!}(\frac{x}{2})^p {}_0F_1(;p+1;-\frac{x^2}{4})$
$\mathrm{LaguerreL}[p,x]$	$\sum_{k=0}^{\infty} \frac{(k-p)!}{(-p)!k!k!}x^k$	$\frac{(k-p)}{(k+1)(k+1)}x$	${}_1F_1(-p;1;x)$
$\mathrm{LaguerreP}[p,x]$	$\sum_{k=0}^{\infty} \frac{(k-p)!(k+p+1)!}{(-p)!(p+1)!k!k!}(\frac{1-x}{2})^k$	$\frac{(k-p)(k+p+1)}{(k+1)(k+1)}\frac{1-x}{2}$	${}_2F_1(-p,p+1;1;\frac{1-x}{2})$
$\mathrm{HermiteH}_{2n}(x)$	$(-1)^n(2n)! \sum_{k=0}^{\infty} \frac{(-1)^k(2x)^{2k}}{(2k)!(n-k)!}$	$\frac{(k-n)}{(k+\frac{1}{2})(k+1)}x^2$	$\frac{(-1)^n(2n)!}{n!} {}_1F_1(-n;\frac{1}{2};x^2)$
$\mathrm{HermiteH}_{2n+1}(x)$	$(-1)^n(2n+1)! \sum_{k=0}^{\infty} \frac{(-1)^k(2x)^{2k+1}}{(2k+1)!(n-k)!}$	$\frac{(k-n)}{(k+\frac{3}{2})(k+1)}x^2$	$\frac{2x(-1)^n(2n+1)!}{n!} {}_1F_1(-n;\frac{3}{2};x^2)$

The standard series for $\log(1+x)$ in Table 3.1 has poor convergence prop-

erties. By subtracting the standard series for $\log(1+x)$ from $\log(1-x)$, we obtain

$$\log\frac{1-x}{1+x} = -2x\left[1+\frac{x^2}{3}+\frac{x^4}{5}+\cdots\right] = -2x\cdot{}_2F_1(1,\frac{1}{2};\frac{3}{2};x^2).$$

Changing variables, we obtain

$$\log y = -2\left(\frac{1-y}{1+y}\right){}_2F_1\left(1,\frac{1}{2};\frac{3}{2};\left(\frac{1-y}{1+y}\right)^2\right),$$

for $0 < y < \infty$. In addition to the functions in Table 3.1, which are mostly computed through direct evaluation of listed hypergeometric series, we also compute $\tan(x)$, $\cot(x)$, and $\arccos(x)$ as

$$\tan(x) = \frac{\sin(x)}{\cos(x)}, \tag{3.1}$$

$$\cot(x) = \frac{\cos(x)}{\sin(x)}, \tag{3.2}$$

$$\arccos(x) = \arcsin\left(\sqrt{1-x^2}\right),\ 0 \le x \le 1. \tag{3.3}$$

3.2 General Approximate Evaluation Algorithm

We regard the parameters $\mathbf{a}, \mathbf{b}$ as fixed in ${}_pF_q(\mathbf{a};\mathbf{b};x)$. In this thesis, we only consider the case when $\mathbf{a}, \mathbf{b}$ are rationals. Let $H(\mathbf{a};\mathbf{b};x) = {}_pF_q(\mathbf{a};\mathbf{b};x)$ denote the *general hypergeometric function* where the parameters as well as p, q can now vary. It is conceivable that each hypergeometric function ${}_pF_q(\mathbf{a};\mathbf{b};x)$ is absolutely approximable, but the function H is not absolutely approximable. In other word, the absolute approximation of H requires a uniform algorithmic method that is applicable to arbitrary values of the parameters. We give this approximate evaluation problem formally as follows:

Given $\mathbf{a} = (a_1, \ldots, a_p)$ and $\mathbf{b} = (b_1, \ldots, b_q)$ where $p, q \geq 0$ and $p \leq q+1$ and the argument x and precision ℓ, compute an approximation $\widetilde{H}(\mathbf{a}; \mathbf{b}; x; \ell)$ of

$$H = H(\mathbf{a}; \mathbf{b}; x)$$

such that

$$|\widetilde{H} - H| \leq 2^{-\ell}.$$

In this section, we present an algorithm for computing $\widetilde{H}$. Let $H(\mathbf{a}; \mathbf{b}; x) = S_n + R_n$ where $S_n = \sum_{k=0}^{n-1} t_k$ and $R_n = \sum_{k=n}^{\infty} t_k$. The algorithm proceeds in two stages: in STAGE A, we compute an n_3 such that $|R_n| \leq 2^{-\ell-1}$ for all $n > n_3$. In STAGE B, we compute an approximation $\widetilde{S}_n$ such that $|\widetilde{S}_n - S_n| \leq 2^{-\ell-1}$. Since STAGE B is clearly computable, it remains to prove that STAGE A is computable.

This basic strategy follows [23]. Among the improvements are: (1) The original paper only addresses the case where the series $\sum_{k\geq 0} t_k$ satisfies $|t_k| > |t_{k+1}|$ for all $k \geq 0$, while our current method is completely general. (2) Originally, STAGE A (in the non-alternating case) determines n_3 by a computational procedure that evaluates t_n for increasing values of n until $n = n_3$. The STAGE A presented below is able to compute n_3 directly.

§46. STAGE A We introduce some notations: Let $a^+_{\max}$ denote the maximum value of the positive a_i's, and $a^-_{\max}$ denote the maximum of $-a_i$'s where $a_i < 0$. If all the a_i's are positive, we set $a^-_{\max}$ to 0, and similarly, if all a_i's are negative, we set $a^+_{\max}$ to 0. Also, set $a_{\max} = \max\{a^+_{\max}, a^-_{\max}\}$. We write $s^+_k(\mathbf{a})$ for the sum $\sum_{i:a_i>0} a_i^k$ which range over positive elements in $\mathbf{a}$. Also $s^-_k(\mathbf{a})$ is the corresponding sum ranging over negative a_i's. Finally, let $\overline{a} = \left(\sum_i^p a_i\right)/p$.

Each term t_{n+1} in our hypergeometric series $H(\mathbf{a};\mathbf{b};x)$ can be written as $t_{n+1} = t_n f(n)x$ where

$$f(n) = P(n)/Q(n), \qquad P(n) = \prod_{i=1}^{p}(n+a_i), \quad Q(n) = \prod_{j=0}^{q}(n+b_j). \tag{3.4}$$

and $b_0 = 1$. Write $\mathbf{b}'$ for $(b_0, b_1, \ldots, b_q)$ where $b_0 = 1$ is the implicit lower parameter.

§47. Finding n_0 Note that $R_n = \sum_{k=n}^{\infty} t_k$; it is clear that t_k has to be strictly decreasing in order to guarantee $|R_n| \le 2^{-\ell-1}$. Therefore, the first step of our algorithm is to compute a n_0 such that $|\frac{t_{n+1}}{t_n}| = |x|f(n) < 1$ holds for $n \ge n_0$. The following lemma gives this formula:

LEMMA 15. *$|x|f(n) < 1$ holds for $n > n_0$ where*

$$n_1 = \begin{cases} \max\{\bar{a} + 2b^-_{max}, (2^p|x|)^{\frac{1}{q+1-p}}\} & \text{if } \ p < q+1, \\ \frac{\bar{a}+b^-_{max}}{1-\sqrt[p]{|x|}} + b^-_{max} & \text{if } \ p = q+1. \end{cases} \tag{3.5}$$

Proof.

$$\begin{aligned} P(n) &= \prod_{i=1}^{p}(a_i+n) \\ &\le \left[\frac{(a_1+n)+(a_2+n)+\cdots+(a_p+n)}{p}\right]^p = (\bar{a}+n)^p \end{aligned}$$

and

$$Q(n) = \prod_{j=0}^{q}(b_j+n) \ge (n - b^-_{max})^{q+1}.$$

So we have

$$\left|\frac{t_{n+1}}{t_n}\right| = \frac{P(n)}{Q(n)}|x| \le \frac{(\bar{a}+n)^p}{(n-b^-_{max})^{q+1}}|x|.$$

We consider the following two cases:

1. $p < q + 1$. Thus

$$\left|\frac{t_{n+1}}{t_n}\right| \leq \left(1 + \frac{\bar{a} + b^-_{max}}{n - b^-_{max}}\right)^p \frac{|x|}{n^{q+1-p}} \leq \frac{2^p|x|}{n^{q+1-p}} \leq 1$$

provided $n \geq \max\{\bar{a} + 2b^-_{max}, (2^p|x|)^{\frac{1}{q+1-p}}\}$.

2. $p = q + 1$. Thus

$$\left|\frac{t_{n+1}}{t_n}\right| \leq \left(1 + \frac{\bar{a} + b^-_{max}}{n - b^-_{max}}\right)^p |x| \leq 1$$

provided $n \geq \frac{\bar{a}+b^-_{max}}{1-\sqrt[p]{|x|}} + b^-_{max}$.

Q.E.D.

Recall that a series $\sum_{n=0}^{\infty} t_n$ is called alternating if $t_n t_{n+1} < 0$ for all $n \in \mathbb{N}$. For alternating series, we have the following lemma:

LEMMA 16. *if $|t_i| \geq |t_{i+1}|$ for all $i \geq n$, then $|R_n| \leq |t_n|$ and $R_n t_n \geq 0$.*

For $n > n_0$, $|R_n|$ is bounded by $|t_n|$. For non-alternating series, we obtain the following bounds for $|R_n|$:

LEMMA 17. *If $|x|f(n) < 1$, then*

1. *if starting from some n, $f(n)$ decreases, i.e., $f(n) \geq f(n+1)$, then*

$$|R_n| \leq \frac{|t_n|}{1 - |x|f(n)}.$$

2. *if starting from some n, $f(n)$ increases, i.e., $f(n) \leq f(n+1)$, then*

$$|R_n| \leq \frac{|t_n|}{1 - |x|}.$$

Proof.

1. Note that if $f(n)$ decreases, then for $i \geq 0$,

$$|t_{n+i}| = |t_n| \cdot |x|^i \prod_{j=0}^{i-1} f(n+j) \leq |t_n| \cdot |x|^i \prod_{j=0}^{i-1} f(n) \leq |t_n| \cdot |x|^i f(n)^i.$$

Summing,

$$|R_n| = |\sum_{i\geq 0} t_{n+i}| \leq \sum_{i\geq 0} |t_{n+i}| \leq |t_n| \sum_{i\geq 0} |x|^i f(n)^i = \frac{|t_n|}{1-|x|f(n)}.$$

2. If $f(n)$ increases, then $f(n) < 1$ since $\lim_{n\to\infty} f(n) = 1$ if $p = q+1$ and $\lim_{n\to\infty} f(n) = 0$ if $p < q+1$. Hence,

$$|t_{n+i}| = |t_n| \cdot |x|^i \prod_{j=0}^{i-1} f(n+j) \leq |t_n| \cdot |x|^i.$$

Summing,

$$|R_n| = |\sum_{i\geq 0} t_{n+i}| \leq \sum_{i\geq 0} |t_{n+i}| \leq |t_n| \sum_{i\geq 0} |x|^i = \frac{|t_n|}{1-|x|}.$$

Q.E.D.

We notice that the geometric case would also apply for the alternating series as well. However, it is obvious that the upper bound ($|t_n|$) of $R(n)$ given by the alternating case is better than here ($\frac{|t_n|}{1-|x|f(n)}$ or $\frac{|t_n|}{1-|x|}$).

§48. Determining n_1 for $p < q+1$ Now we need to determine a value n_1 beyond which $f(n)$ is monotone increasing or decreasing. Write "$f(n) \nearrow n_1$" to mean that for all $n > n_1$, $f(n) < f(n+1)$. Similarly, "$f(n) \searrow n_1$" means that for all $n > n_1$, $f(n) > f(n+1)$. In either case, we can write "$f(n) \updownarrow n_1$". We may also write "$f(n) \nearrow$" if $f(n) \nearrow n_1$ for some n_1. Similarly, for "$f(n) \searrow$". The following Lemma 18 and Lemma 20 gives such a condition depends on $p < q+1$ and $p = q+1$.

LEMMA 18. *Let $p < q+1$ and $\{a_1, \ldots, a_p\} \neq \{b_0, b_1, \ldots, b_q\}$ viewed as multi-sets. Denote*

$$r = \frac{s_1(\mathbf{b}') - \frac{1}{2}s_1^+(\mathbf{a}) - 2s_1^-(\mathbf{a})}{q+1-p},$$

then $f(n) \searrow n_1$ where

$$n_1 = \max\{a_{\max}^+, 2a_{\max}^-, b_{\max}^-, r\} \tag{3.6}$$

Proof. We have

$$\begin{aligned} f(n) &= \frac{\prod_{i=1}^p (a_i + n)}{\prod_{j=0}^q (b_j + n)} \\ &= \frac{\prod_{i=1}^p (\frac{a_i}{n} + 1)}{\prod_{j=0}^q (\frac{b_j}{n} + 1)} \left(\frac{1}{n}\right)^{q+1-p} \\ \log f(n) &= \sum_{i=1}^p \log(\frac{a_i}{n} + 1) - \sum_{j=0}^q \log(\frac{b_j}{n} + 1) \\ &+ (q+1-p)\log(\frac{1}{n}). \end{aligned}$$

Let $\nu = 1/n$ and define $h(\nu)$ via

$$\begin{aligned} h(\nu) &= \log f(1/\nu) \\ &= \sum_{i=1}^p \log(a_i\nu + 1) - \sum_{j=0}^q \log(b_j\nu + 1) \\ &+ (q+1-p)\log\nu \\ h'(\nu) &= \sum_{i=1}^p \frac{a_i}{a_i\nu + 1} - \sum_{j=0}^q \frac{b_j}{b_j\nu + 1} + \frac{q+1-p}{\nu}. \end{aligned}$$

From the following facts:

$$\begin{aligned}
\frac{a_i}{a_i\nu+1} &> \frac{a_i}{2} \quad \text{for} \quad a_i>0, n>a^+_{max},\\
\frac{a_i}{a_i\nu+1} &> 2a_i \quad \text{for} \quad a_i<0, n>2a^+_{max},\\
-\frac{b_j}{b_j\nu+1} &> -b_j \quad \text{for} \quad b_j>0, n>0,\\
-\frac{b_j}{b_j\nu+1} &> -b_j \quad \text{for} \quad b_j<0, n>b^-_{max},
\end{aligned}$$

for $n > \max\{a^+_{\max}, 2a^-_{\max}, b^-_{\max}\}$, we have

$$h'(\nu) \geq \frac{1}{2}s_1^+(\mathbf{a}) + 2s_1^-(\mathbf{a}) - s_1(\mathbf{b}) + \frac{q+1-p}{\nu}.$$

Hence, if we choose

$$n_1 = \max\{a^+_{\max}, 2a^-_{\max}, b^-_{\max}, r\},$$

we conclude that $h'(\nu) > 0$. This means as $h(\nu)$ increases with ν. But as $n \to \infty$, ν decreases towards 0 and so $h(\nu)$ is decreasing, i.e., $f(n) \searrow$ as claimed. **Q.E.D.**

§49. Determining n_1 for $p = q+1$ For the case $p = q+1$, we need a lemma from the Symmetric Polynomial theorem. Let

$$s_k(\mathbf{a}) = \sum_{i=1}^{p} a_i^k, \qquad k = 1, \ldots, p$$

and $\sigma_k(\mathbf{a})$ be the kth elementary symmetric function on $\mathbf{a}$. Thus

$$\sigma_1(\mathbf{a}) = s_1(\mathbf{a}), \quad \sigma_2(\mathbf{a}) = \sum_{1\leq i<j\leq p} a_i a_j,$$

$$\sigma_3(\mathbf{a}) = \sum_{1\leq i<j<k\leq p} a_i a_j a_k, \quad \ldots, \quad \sigma_p(\mathbf{a}) = \prod_{i=1}^{p} a_i.$$

Also, define the polynomials

$$P_{\mathbf{a}}(n) = \prod_{i=1}^{p}(a_i + n), \qquad P_{\mathbf{b}}(n) = \prod_{i=1}^{p}(b_i + n) \tag{3.7}$$

LEMMA 19. *Let* $\mathbf{a} = (a_1, \ldots, a_p)$ *and* $\mathbf{b} = (b_1, \ldots, b_p)$ *where* $a_1 \leq \cdots \leq a_p$ *and* $b_1 \leq \cdots \leq b_p$. *The following three statements are equivalent:*

(1) $(\forall k = 1, \ldots, p)[s_k(\mathbf{a}) = s_k(\mathbf{b})]$.

(2) $(\forall k = 1, \ldots, p)[\sigma_k(\mathbf{a}) = \sigma_k(\mathbf{b})]$.

(3) The polynomials $P_{\mathbf{a}}(n)$ *and* $P_{\mathbf{b}}(n)$ *are equal.*

(4) $\mathbf{a} = \mathbf{b}$.

Proof. The equivalence of (1) and (2) is a simple consequence of the Newton-Girard formulas which show that the $s_k(\mathbf{a})$'s and the $\sigma_k(\mathbf{a})$'s are derived from each other:

$$\begin{aligned} s_1(\mathbf{a}) &= \sigma_1(\mathbf{a}) \\ s_2(\mathbf{a}) &= (\sigma_1(\mathbf{a}))^2 - 2\sigma_2(\mathbf{a}) \\ s_3(\mathbf{a}) &= (\sigma_1(\mathbf{a}))^3 - \sigma_1(\mathbf{a})\sigma_2(\mathbf{a}) + 3\sigma_3(\mathbf{a}) \\ &\ldots \end{aligned}$$

The equivalence of (2) and (3) follows since $\sigma_k(\mathbf{a})$ is the coefficient of n^k in $P_{\mathbf{a}}(n)$ for all $k = 1, \ldots, n$. The equivalence of (3) and (4) uses the following remark: if the ith derivative $P_{\mathbf{a}}^{(i)}(n)$ vanishes at $n = a$ for all $i = 0, 1, \ldots, r-1$ but $P_{\mathbf{a}}^{(r)}(a) \neq 0$, then a occurs exactly $r \geq 0$ times in $\mathbf{a}$. Thus, $P_{\mathbf{a}}(n) = P_{\mathbf{a}}(n)$ implies that a occurs exactly the same number of times in $\mathbf{a}$ as in $\mathbf{b}$. This shows that $P_{\mathbf{a}}(n) = P_{\mathbf{a}}(n)$ implies $\mathbf{a} = \mathbf{b}$. The converse implication is immediate: $\mathbf{a} = \mathbf{b}$ implies $P_{\mathbf{a}}(n) = P_{\mathbf{a}}(n)$. **Q.E.D.**

Now we can prove the following lemma:

LEMMA 20. *Let $p = q + 1$ and $\{a_1, \ldots, a_p\} \neq \{b_0, b_1, \ldots, b_q\}$ viewed as multi-sets, and $k = 1, \ldots, p$ be the smallest such index such that $s_k(\mathbf{a}) \neq s_k(\mathbf{b}')$.*

(a) $f(n) \searrow$ iff $(-1)^k(s_k(\mathbf{b}') - s_k(\mathbf{a})) > 0$.

(b) Denote

$$r_1 = \sqrt[k]{\frac{s_k(\mathbf{b}')}{s_k(\mathbf{a})}}, \quad r_2 = \sqrt[k]{\frac{s_k^+(\mathbf{b}') - s_k^-(\mathbf{a})}{s_k^+(\mathbf{a}) - s_k^-(\mathbf{b}')}},$$

then we have:

(i) if k is even, then $f(n) \searrow n_1$ where

$$n_1 = \max\left\{a_{max}, b_{max}, b_{max}\frac{1}{r_1 - 1}\right\}. \tag{3.8}$$

provided that $s_k(\mathbf{b}') > s_k(\mathbf{a})$, and $f(n) \nearrow n_1$ where

$$n_1 = \max\left\{a_{max}, b_{max}, a_{max}\frac{r_1}{r_1 - 1}\right\}. \tag{3.9}$$

provided that $s_k(\mathbf{b}') < s_k(\mathbf{a})$,

(ii) if k is odd, then $f(n) \nearrow n_1$, where

$$n_1 = \max\left\{a_{max}, b_{max}, \frac{b_{max}^+ + b_{max}^- r_2}{r_2 - 1}\right\}. \tag{3.10}$$

provided that $s_k(\mathbf{b}') > s_k(\mathbf{a})$, and $f(n) \searrow n_1$ where

$$n_1 = \max\left\{a_{max}, b_{max}, \frac{a_{max}^+ r_2 + a_{max}^-}{1 - r_2}\right\}. \tag{3.11}$$

provided that $s_k(\mathbf{b}') < s_k(\mathbf{a})$,

Proof. (a) In this case, we have

$$h'(\nu) = \sum_{i=1}^{p} \frac{a_i}{a_i\nu + 1} - \sum_{j=1}^{q} \frac{b_j}{b_j\nu + 1}.$$

In general, for $r \geq 1$, the rth derivative is

$$h^{(r)}(\nu) = (r-1)!(-1)^r \left(\sum_{j=0}^{q}\left(\frac{b_j}{b_j\nu + 1}\right)^r - \sum_{i=1}^{p}\left(\frac{a_i}{a_i\nu + 1}\right)^r\right).$$

Also, $h^{(0)}(\nu)$ is simply $h(\nu)$. Evaluating at $\nu = 0$, we get

$$\begin{aligned} h^{(r)}(\nu)\big|_{\nu=0} &= (r-1)!(-1)^r \left(\sum_{j=0}^{q} b_j^r - \sum_{i=1}^{p} a_i^r \right) \\ &= (r-1)!(-1)^r \left(s_r(\mathbf{b}') - s_r(\mathbf{a}) \right). \end{aligned}$$

By Lemma 19, we know that there exists a smallest index k such that $s_k(\mathbf{a}) \neq s_k(\mathbf{b}')$. Then $h^{(r)}(\nu)\big|_{\nu=0} = 0$ for $r = 1, \ldots, k-1$ and $h^{(k)}(\nu)\big|_{\nu=0} \neq 0$. So for $\varepsilon > 0$ small enough, $h(\varepsilon)$ and $h'(\varepsilon)$ has the sign of $h^{(k)}(\nu)\big|_{\nu=0}$. As in the proof of Lemma 18, $h'(\varepsilon) > 0$ means that $f(n) \searrow$. This proves $f(n) \searrow$ iff $(-1)^k \left(s_k(\mathbf{b}') - s_k(\mathbf{a}) \right) > 0$.

(b) Denoting

$$S_k(\nu) = \sum_{j=0}^{q} \left(\frac{b_j}{b_j \nu + 1} \right)^k - \sum_{i=1}^{p} \left(\frac{a_i}{a_i \nu + 1} \right)^k,$$

then we have $f(n) \searrow$ if $(-1)^k S_k(\nu) > 0$ and $f(n) \nearrow$ if $(-1)^k S_k(\nu) < 0$.

If k is even, then b_j^k and a_i^k are positive. For $n > \max\{a_{max}, b_{max}\}$, we have

$$0 < \frac{1}{1 + a_{max}\nu} \leq \frac{1}{1 + a_i \nu} \leq 1, \quad 0 < \frac{1}{1 + b_{max}\nu} \leq \frac{1}{1 + b_i \nu} \leq 1.$$

Hence,

$$\begin{aligned} S_k(\nu) &\geq \left(\frac{1}{1 + b_{max}\nu} \right)^k \sum_{j=0}^{q} {b_j}^k - \sum_{i=1}^{p} {a_i}^k \\ &= \left(\frac{1}{1 + b_{max}\nu} \right)^k \left(s_k(\mathbf{b}') - s_k(\mathbf{a}) \right). \end{aligned}$$

We can choose

$$n > b_{max} \frac{1}{r_1 - 1}$$

to ensure $S_k(\nu) > 0$ provided $s_k(\mathbf{b}') > s_k(\mathbf{a})$. In this case, $f(n) \searrow n_1$ where

$$n_1 = \max\left\{a_{max}, b_{max}, b_{max}\frac{1}{r_1 - 1}\right\}.$$

Similarly we can choose another n_1 for the case $s_k(\mathbf{b}') < s_k(\mathbf{a})$.

If k is odd, then we can rewrite $S_k(\nu)$ as

$$\begin{aligned} S_k(\nu) &= \sum_j \left(\frac{b_j^+}{1 + b_j^+\nu}\right)^k + \sum_j \left(\frac{b_j^-}{1 + b_j^-\nu}\right)^k \\ &- \sum_i \left(\frac{a_i^+}{1 + a_i^+\nu}\right)^k - \sum_i \left(\frac{a_i^-}{1 + a_i^-\nu}\right)^k. \end{aligned}$$

where a_i^+, b_j^+ are positive a_i, b_j and a_i^-, b_j^- are negative a_i, b_j.

For $n > \max\{a_{max}, b_{max}\}$, we have

$$0 < \frac{1}{1 + b_{max}^+\nu} \leq \frac{1}{1 + b_j^+\nu} \leq \frac{1}{1 + a_i^-\nu},$$

$$0 < \frac{1}{1 + a_i^+\nu} \leq \frac{1}{1 + b_i^-\nu} \leq \frac{1}{1 - b_{max}^-}.$$

Hence,

$$\begin{aligned} S_k(\nu) &\geq \left(\frac{1}{1 + b_{max}^+\nu}\right)^k \left(\sum_j (b_j^+)^k - \sum_i (a_i^-)^k\right) \\ &- \left(\frac{1}{1 - b_{max}^-\nu}\right)^k \left(\sum_i (a_i^+)^k - \sum_j (b_j^-)^k\right) \\ &= \left(\frac{1}{1 + b_{max}^+\nu}\right)^k \left(s_k^+(\mathbf{b}') - s_k^-(\mathbf{a})\right) \\ &- \left(\frac{1}{1 - b_{max}^-\nu}\right)^k \left(s_k^+(\mathbf{a}) - s_k^-(\mathbf{b}')\right). \end{aligned}$$

We can choose

$$n > \frac{b^+_{max} + b^-_{max} r_2}{r_2 - 1}$$

to ensure $S_k(\nu) > 0$ provided $s_k(\mathbf{b}') > s_k(\mathbf{a})$. In this case, $f(n) \nearrow n_1$ where

$$n_1 = \max\left\{a_{max}, b_{max}, \frac{b^+_{max} + b^-_{max} r_2}{r_2 - 1}\right\}.$$

Similarly we can choose another n_1 for the case $s_k(\mathbf{b}') < s_k(\mathbf{a})$. **Q.E.D.**

§50. Defining n_2 Let us further define

$$n_2 := \max\{n_0, n_1\} \tag{3.12}$$

for the remainder of this section (this definition of n_2 will be refined in the next section). Let

$$u_n = \frac{(a_1)_n (a_2)_n \cdots (a_p)_n}{(b_1)_n (b_2)_n \cdots (b_q)_n} \frac{1}{n!},$$

then

$$t_n = u_n x^n$$

§51. Computing n_3

LEMMA 21. *Denote*

$$r = -\ell - 1 - \lg|u_{n_2}|, \quad s = n_2 \lg f(n_2),$$

then the bound $|R_n| < 2^{-\ell-1}$ for all $n > n_3$ where

$$n_3 = \begin{cases} \max\{n_2, \frac{r}{\lg|x|}\} & \text{if } x < 0 \text{ and } f(n) \nearrow, \\ \max\{n_2, \frac{r+s}{\lg(|x| f(n_2))}\} & \text{if } x < 0 \text{ and } f(n) \searrow, \\ \max\{n_2, \frac{r+\lg(1-x)}{\lg|x|}\} & \text{if } x > 0 \text{ and } f(n) \nearrow, \\ \max\{n_2, \frac{r+s+\lg(1-x f(n_2))}{\lg(|x| f(n_2))}\} & \text{if } x > 0 \text{ and } f(n) \searrow. \end{cases} \tag{3.13}$$

Proof. If $x < 0$, then

$$\begin{aligned} |R(n)| &\leq |t_n| \\ &= |t_{n_2}||x|^{n-n_2} \prod_{j=0}^{n-n_2-1} f(n_2+j) \\ &= |u_{n_2}||x|^{n} \prod_{j=0}^{n-n_2-1} f(n_2+j) \end{aligned}$$

for $n > n_2$.

If $f(n) \nearrow n_2$, then $f(n) < 1$ for $n > n_2$. To ensure

$$|R_n| \leq |u_{n_2}||x|^n < 2^{-\ell-1},$$

we require

$$n > n_3 = \frac{-\ell - 1 - \lg |u_{n_2}|}{\lg |x|}.$$

If $f(n) \searrow n_2$, then $f(n) > f(n+1)$ for $n > n_2$. To ensure

$$|R_n| \leq |u_{n_2}||x|^n f(n_2)^{n-n_2} < 2^{-\ell-1},$$

we require

$$n > n_3 = \frac{-\ell - 1 - \lg |u_{n_2}| + n_2 \lg f(n_2)}{\lg (|x| f(n_2))}.$$

If $x > 0$, then

$$\begin{aligned} |R(n)| &= |\sum_{i=0}^{\infty} t_{n+i}| \\ &\leq \sum_{i=0}^{\infty} |t_{n+i}| \\ &\leq |t_n| \sum_{i=0}^{\infty} |x|^i \prod_{j=0}^{i-1} f(n+j)|. \end{aligned}$$

If $f(n) \nearrow n_2$, then $f(n) < 1$ for $n > n_2$. To ensure

$$|R_n| \leq |t_n| \sum_{i=0}^{\infty} |x|^i = |t_n| \frac{1}{1-x} < |u_{n_2}||x|^n \frac{1}{1-x} < 2^{-\ell-1}$$

we require

$$n > n_3 = \frac{-\ell - 1 - \lg |u_{n_2}| + \lg(1-x)}{\lg |x|}.$$

If $f(n) \searrow n_2$, then $f(n) > f(n+1)$ for $n > n_2$. To ensure

$$\begin{aligned} |R_n| &\leq |t_n| \sum_{i=0}^{\infty} |x|^i f(n)^i = |t_n| \frac{1}{1-xf(n)} \\ &< |u_{n_2}||x|^n f(n_2)^{n-n_2} \frac{1}{1-xf(n_2)} < 2^{-\ell-1}, \end{aligned}$$

we require

$$n > n_3 = \frac{-\ell - 1 - \lg |u_{n_2}| + n_2 \lg f(n_2) + \lg(1-xf(n_2))}{\lg (|x| f(n_2))}.$$

Q.E.D.

§52. Algorithm of computing n_3 We are now ready to present the algorithm to compute n_3.

- STEP 0: PRELIMINARY. We first sort $\mathbf{a}$ and $\mathbf{b}$ in nondecreasing order. By a merge-like process, we can delete pairs (a_i, b_j) of identical upper and lower parameters where $a_i = b_j$. Continue to let $\mathbf{a}, \mathbf{b}$ denote the parameters after they are processed in this way. If $p = q = 0$ then we are reduced to the evaluation of $exp(x)$. Otherwise, we know that $\mathbf{a} \neq \mathbf{b}$. If $p = q + 1$, verify that $|x| < 1$. If $x = 0$, return 1.

- STEP 1: Compute a value $n_0 = n_0(\mathbf{a}, \mathbf{b}, x)$ such that $|x| f(n) < 1$.

  – CASE $p < q + 1$: compute

$$n_0 = \max\{\overline{a} + 2b^-_{max}, (2^p|x|)^{1/(q+1-p)}\}.$$

– CASE $p = q + 1$: compute

$$n_0 = \frac{\overline{a} + b^-_{max}}{1 - \sqrt[p]{|x|}} + b^-_{max}.$$

- STEP 2: Compute a value $n_1 = n_1(\mathbf{a}, \mathbf{b})$ such that $f(n) \updownarrow n_1$.

  – CASE $p < q + 1$: compute

  $$r = \frac{s_1(\mathbf{b}') - \frac{1}{2}s_1^+(\mathbf{a}) - 2s_1^-(\mathbf{a})}{q + 1 - p}$$

  Then compute

  $$n_1 = \max\{a^+_{\max}, 2a^-_{\max}, b^-_{\max}, r\}$$

  In this case, $f(n) \searrow n_1$.

  – CASE $p = q+1$: we find the smallest $k = 1, \ldots, p$ such that $\sum_i a_i^k \neq \sum_j b_j^k$. Such a k exists since $\mathbf{a} \neq \mathbf{b}$.

  Then for k even, compute

  $$r_1 = \sqrt[k]{\frac{s_k(\mathbf{b}')}{s_k(\mathbf{a})}}.$$

  If $s_k(\mathbf{b}') > s_k(\mathbf{a})$, compute

  $$n_1 = \max\left\{a_{max}, b_{max}, b_{max}\frac{1}{r_1 - 1}\right\}.$$

  If $s_k(\mathbf{b}') < s_k(\mathbf{a})$, compute

  $$n_1 = \max\left\{a_{max}, b_{max}, a_{max}\frac{r_1}{r_1 - 1}\right\}.$$

  For k odd, compute

  $$r_2 = \sqrt[k]{\frac{s_k^+(\mathbf{b}') - s_k^-(\mathbf{a})}{s_k^+(\mathbf{a}) - s_k^-(\mathbf{b}')}}.$$

If $s_k(\mathbf{b}') > s_k(\mathbf{a})$, compute

$$n_1 = \max\left\{a_{max}, b_{max}, \frac{b^+_{max} + b^-_{max} r_2}{r_2 - 1}\right\}.$$

If $s_k(\mathbf{b}') < s_k(\mathbf{a})$, compute

$$n_1 = \max\left\{a_{max}, b_{max}, \frac{a^+_{max} r_2 + a^-_{max}}{1 - r_2}\right\}.$$

- STEP 3: Compute $n_2 = \max\{n_0, n_1\}$.
- STEP 4: Compute n_3 such that $|R_n| < 2^{-\ell-1}$ for $n \geq n_3$. Compute

$$r = -\ell - 1 - \lg|u_{n_2}|, \quad s = n_2 \lg f(n_2)$$

then

  - CASE $x < 0$: compute

$$n_3 = \begin{cases} \max\{n_2, \frac{r}{\lg|x|}\} & \text{if } f(n) \nearrow, \\ \max\{n_2, \frac{r+s}{\lg(|x|f(n_2))}\} & \text{if } f(n) \searrow. \end{cases}$$

  - CASE $x > 0$: compute

$$n_3 = \begin{cases} \max\{n_2, \frac{r+\lg(1-x)}{\lg|x|}\} & \text{if } f(n) \nearrow, \\ \max\{n_2, \frac{r+s+\lg(1-xf(n_2))}{\lg(|x|f(n_2))}\} & \text{if } f(n) \searrow. \end{cases}$$

This give us:

THEOREM 22. *The general hypergeometric function H is absolutely approximable.*

Proof. Given $\mathbf{a}, \mathbf{b}, x, \ell$, we compute n_3 as given in the above algorithm. We then compute an approximation $\widetilde{S_{n_3}}$ which approximates S_{n_3} to $\ell + 1$ absolute bits. This value also approximates $H(\mathbf{a}; \mathbf{b}; x)$ to ℓ absolute bits. **Q.E.D.**

Table 3.2: Transformations of hypergeometric functions.

$E(x) = A(x)F(B(x))$	$F(x)$	$A(x)$	$B(x)$
$\exp(x)$	${}_0F_0(;;x)$	1	1
$\cos(x)$	${}_0F_1(;\frac{1}{2};x)$	1	$-x^2/4$
$\sin(x)$	${}_0F_1(;\frac{3}{2};x)$	x	$-x^2/4$
$\cosh(x)$	${}_0F_1(;\frac{1}{2};x)$	1	$x^2/4$
$\sinh(x)$	${}_0F_1(;\frac{3}{2};x)$	x	$x^2/4$
$\mathrm{erf}(x)$	${}_1F_1(\frac{1}{2};\frac{3}{2};x)$	x	$-x^2$
$(1+x)^{-v}$	${}_1F_0(v;x)$	1	$-x$
$\ln(1+x)$	${}_2F_1(1,1;2;x)$	x	$-x$
$\arcsin(x)$	${}_2F_1(\frac{1}{2},\frac{1}{2};\frac{3}{2};x)$	x	x^2
$\arctan(x)$	${}_2F_1(\frac{1}{2},1;\frac{3}{2};x)$	x	$-x^2$

Many well-known functions in analysis are obtained by the simple transformations of hypergeometric functions. For instance, if $A(x)$ and $B(x)$ are functions (typically simple polynomials), we transform $F(x)$ to $E(x) := A(x)F(B(x))$ to obtain familiar functions. This is illustrated in Table 3.2 (see [56, p. 42ff] for more examples).

We can extend this result to simple functions that are derived from H (e.g., Table 3.2):

THEOREM 23. *If $A(x), B(x) \in \mathbb{Z}[x]$ then the function of the form $F(x) = A(x){}_pF_q(\mathbf{a};\mathbf{b};B(x))$ is absolutely approximable.*

Furthermore, it is clear that these results also hold when we view the functions as complex.

3.3 Evaluation at a Blackbox Number

This section seeks to generalize the algorithm of Section 3 to the case where x is a black-box number. In [23], such a solution is given for common elementary functions, exploiting well-known properties of such functions. We shall now solve this in complete generality.

In the following, we assume the usual rational parameters $\mathbf{a}, \mathbf{b}$, the precision $\ell \in \mathbb{Z}$ and a blackbox number x. Our main result says that $H(\mathbf{a}; \mathbf{b}; x)$ is absolutely approximable. We can make this precise with the help of oracle Turing machines. In general, if $f(x_1, \ldots, x_p; y_1, \ldots, y_q)$ is a real function where the x_i's are representable reals and y_j's are black box numbers, then we say that f is **absolutely approximable** if there is an oracle Turing machine taking representable real inputs $x_1, \ldots, x_p$ and ℓ, and q oracles for $y_1, \ldots, y_q$, such that within a finite time, it halts with an absolute ℓ-bit approximation to $f(x_1, \ldots, x_p, y_1, \ldots, y_q)$. We call an oracle for y_i by writing a pair (i, ℓ') on a special tape, and in the next instant, an absolute ℓ'-bit approximation of y_i appears on another special output tape.

When $\mathbf{a}, \mathbf{b}$ are understood, we will simply write $F(x)$ or $\widetilde{F}(x; \ell)$ for $H(\mathbf{a}; \mathbf{b}; x)$ and $\widetilde{H}(\mathbf{a}; \mathbf{b}; x; \ell)$ respectively. For any $n \geq 0$, let $F(x) = S_n(x) + R_n(x)$ where $S_n(x) = \sum_{k=0}^{n-1} t_k$ and $R_n(x) = \sum_{k \geq n} t_k$.

Let $[u]$ denote an interval $[\underline{u}, \overline{u}]$ where $\underline{u}, \overline{u}$ are bigfloats. We further assume that

$$0 \notin [u], \text{ and } \quad \text{if } \; p = q + 1 \; \text{ then } \; [u] < 1. \tag{3.14}$$

If $f(x)$ is a real function, let $f([u]) = \{f(\theta) : \underline{u} \leq \theta \leq \overline{u}\}$. Write $f([u]) < M$ if each $y \in f([u])$ is less than M.

LEMMA 24. *Given* $[u]$ *as in (3.14), we can compute* $\overline{n}_3$ *such that* $|R_{\overline{n}_3}([u])| < 2^{-\ell}$.

Proof. Observe that all the inequalities required in STAGE A can be extended by replacing x by $[u]$ since $[u]$ satisfies (3.14). **Q.E.D.**

LEMMA 25. *Given* $[u]$ *as in (3.14), we can compute an* $M \in \mathbb{F}_1$ *such that* $|F'([u])| \leq M$ *where* $F'(x)$ *denotes differentiation with respect to* x.

Proof. Note that $F'(x)$ is just another hypergeometric function multiplied by a rational number since

$$F'(x) = \frac{P_{\mathbf{a}}(0)}{P_{\mathbf{b}}(0)} H(\mathbf{a}+1; \mathbf{b}+1; x)$$

where $P_{\mathbf{a}}(n)$ is given by (3.7). Thus by Lemma 24, we can determine an $\overline{n}_3$ such that truncating the series for $F'(x)$ after the first $\overline{n}_3$ terms incurs an error of at most $1/2$. Then compute an approximation $\widetilde{S}$ of the sum of the first $\overline{n}_3$ terms, error at most $1/2$. Take $M = \widetilde{S} + 1$. **Q.E.D.**

§53. Blackbox Evaluation Algorithm The following algorithm will absolutely approximate H relative to a blackbox number x:

BLACKBOX EVALUATION ALGORITHM

Input: Rational $\mathbf{a}, \mathbf{b}, \ell$ and blackbox number x.

Output: A bigfloat $\widetilde{y}$ such that $|\widetilde{y} - H(\mathbf{a}; \mathbf{b}; \widetilde{x})| \leq 2^{-\ell}$.

0. Initialize $s \leftarrow \ell + 1$.

1. Compute $\widetilde{x}$ such that $|\widetilde{x} - x| < 2^{-s}$.

   This is just one call to the black box x.

   Let $[u] = [\underline{u}, \overline{u}]$

   where $\underline{u} = \widetilde{x} - 2^{-s}$ and $\overline{u} = \widetilde{x} + 2^{-s}$.

2. While $[u]$ does not satisfy (3.14),

   keep doubling the precision s in $\widetilde{x}$ until it does.

3. Using Lemma 25, compute M such that $|F'([u])| < M$.

4. [Repeat Step 1]

   Recompute $\widetilde{x}$ such that $|\widetilde{x} - x| < 2^{-s-\max\{0, \lg M\}}$

   Let $[v]$ be the interval based on this new value of $\widetilde{x}$.

5. Using Theorem 22, compute $\widetilde{y} = \widetilde{H}(\mathbf{a}; \mathbf{b}; \widetilde{x}; \ell + 1)$.

6. Return $\widetilde{y}$.

THEOREM 26. *If $x \neq 0$ and ($p = q + 1$ implies $|x| < 1$), then the Blackbox Evaluation Algorithm halts and gives a correct output:*

$$|\widetilde{y} - H(\mathbf{a}; \mathbf{b}; x)| \leq 2^{-\ell}.$$

Proof. Halting of the algorithm is guaranteed because of the assumption that $x \neq 0$ and ($|x| < 1$ if $p = q + 1$). Next, we must show that $|\widetilde{y} - F(x)| \leq 2^{-\ell}$. But Step 5 implies $|\widetilde{y} - F(\widetilde{x})| \leq 2^{-\ell-1}$. Hence it suffices to show that

$$F(\widetilde{x}) - F(x)| \leq 2^{-\ell-1}. \tag{3.15}$$

By Step 4, $\widetilde{x} = x + \delta$ where $|\delta| \leq 2^{-s}/\max\{1, M\} \leq 2^{-\ell-1}/M$. Therefore

$$|F(\widetilde{x}) - F(x)| = |F(x+\delta) - F(x)| \leq |\delta \cdot F'(x+\theta\delta)|$$

for some $0 \leq \theta \leq 1$ (intermediate value theorem). According to Step 2,

$$x + \theta\delta \in [u] = [\widetilde{x} - 2^{-s}, \widetilde{x} + 2^{-s}] = [x + \delta - 2^{-s}, x + \delta + 2^{-s}],$$

and thus $|F'(x+\theta\delta)| \leq M$ by Step 3. Hence

$$|F(\widetilde{x}) - F(x)| \leq |\delta \cdot F'(x+\theta\delta)| \leq (2^{-\ell-1}/M)(M) = 2^{-\ell-1}.$$

Q.E.D.

Remark: the precision of $\widetilde{x}$ in Step 1 is quite arbitrary. However, it seems to be best to choose a precision (like ℓ) that is close to, but not exceeding, the precision s in Step 4. We want it to be close to s so that our estimate of M is not too far off, but we do not want to exceed s so that Step 1 will not expend more effort than the effort in Step 4. Also, after Step 4, we might wish to compute an updated (smaller) value of M based on $[v]$. But this requires a corresponding update on $\widetilde{x}$. This iteration can be repeated, but it seems to give diminishing returns after the second iteration.

The power of the above blackbox algorithm is seen in the following application:

COROLLARY 27. *Fixing x, define $G(\mathbf{a}; \mathbf{b}) := H(\mathbf{a}; \mathbf{b}; x)$. If x is any approximable real number, then G is absolutely approximable.*

The approximable real numbers include all algebraic numbers and most of the common constants of analysis such as π and e. They are also closed under rational operations.

3.4 Complexity

We next derive the complexity of the procedure in Section 3.2. This is quite involved, and we must further sharpen the algorithm of Section 3.2 before an explicit bound is possible.

§54. Bigfloat Representation Our algorithm, however, will usually compute using bigfloat approximations. We represent a bigfloat x by a pair e, f of integers in binary notation. The pair (e, f) represents the value $x = f2^{e-\mathrm{msb}(f)}$ where $\mathrm{msb}(f) = \lfloor \lg f \rfloor$ To emphasize the representation of x by (e, f), we write $x \sim \langle e, f \rangle$. Here, e and f are the **exponent** and **fraction** of the representation. Note that $\langle e, f \rangle$ and $\langle e - 1, 2f \rangle$ represent the same number. A representation $\langle e, f \rangle$ is **normal** if f is odd, or if $e = f = 0$. Clearly every bigfloat in $\mathbb{F}$ has a unique normal representation. Let $\langle f \rangle$ be a shorthand for $f2^{-\mathrm{msb}(f)} \sim \langle 0, f \rangle$. Thus for $f \neq 0$, we have $\langle f \rangle \in [1, 2)$, representing the fraction obtained by placing a binary point immediately to the left of the most significant bit in the binary notation for f.

§55. Size of the Input We now give our input and representation of numbers more explicitly. If a is a rational number p/q then $size(a)$ is defined as $\max\{\lg |p|, \lg |q|\}$. If a is a bigfloat then it is also a rational number, and hence $size(a)$ is well-defined. In fact, if $a \sim \langle e, f \rangle$, then $size(a) = \max\{e, \lg |f|\}$. If the input to $\widetilde{H}$ is the usual $(\mathbf{a}, \mathbf{b}, x, \ell)$ (all rationals), we say the **size** of this input is the quadruple (q', n, m, ℓ) where $q' = q + 1$, $n = \max_{i,j}\{size(a_i), size(b_j)\}$ and $m = size(x)$.

§56. Brent's Complexity Bounds Our complexity bounds will contain terms involving $M(n)$, defined as the complexity of multiplying two n-bit numbers. We can plug in the Schönhage-Strassen bound $M(n) = O(n \log n \log\log n)$, but it may also be useful to substitute $M(n) = n^2$ to estimate the complexity of using naive multiplication.

The following result is taken from [20] with a correction [1]:

LEMMA 28. *Let $x = \langle e_x, f_x\rangle, y = \langle e_y, f_y\rangle$ be bigfloats, and s be a positive number. Then there exists an algorithm to*

(1) evaluate x to s relative bits in $O(s)$ time.

(2) evaluate $x \pm y$ to s relative bits in $O(s + \lg|e_x e_y|)$ time

(3) evaluate $x \cdot y, 1/x, \sqrt{x}$ to s relative bits in $O(M(s) + \lg|e_x e_y|)$ time.

(4) evaluate $\lg(x)$ to s relative bits in $O(M(s)\lg s + \lg|e_x|)$ time.

Using the above results, we can easily get the complexity bound of approximating a rational number and adding of two or more rational numbers:

LEMMA 29. *Let $a = \frac{N}{D}$ be a rational number and $n = size(a) = \max\{\lg N, \lg D\}$. There exist algorithms to approximate a to s bits relative precision in $O(M(s) + \lg n)$ time.*

Proof. We can express N and D by bigfloats $\langle msb(N), N\rangle$ and $\langle msb(D), D\rangle$, then by Lemma 28, we can evaluate $\frac{N}{D}$ to s bits in $O(M(s)+\lg n)$ time. **Q.E.D.**

COROLLARY 30. *(i) Let a, b be rational numbers with at most n bits in their numerators and denominators. Then we can evaluate $a + b$ to s relative bits in*

[1] The results in [20] say we can evaluate $x \pm y$ to s relative bits in $O(s + \max\{e_x, e_y\})$ time, which is not true.

$O(M(s)+\lg n)$ *time. (ii) We can compute* $s_1(\mathbf{a})$ *to* s *relative bits in* $O(q'(M(s+\lg q')+\lg n))$ *or* $O(q'(M(s+q')+\lg n))$ *time.*

Proof. (i) Evaluating $a+b$ to s relative bits can be done by first approximating a and b to $s+2$ relative bits, truncating them to $s+2$ bits, and then adding those values up to s relative bits. The first step takes $O(M(s)+\lg n)$ time and the addition takes $O(s+\lg n)$ time, so the total running time is $O(M(s)+\lg n)$. (ii) The first complexity comes from evaluating a balanced binary tree whose leaves are $a_1,\ldots,a_p$. The second comes from evaluating a linear expression. **Q.E.D.**

§57. Complexity of Computing n_0 The upper bound for n_0 in our algorithm and the complexity of computing n_0 is shown below:

LEMMA 31. *We have* $n_0 \le 2^{3q'(n+m)}$. *Computing* n_1 *takes* $O(M(q)+m)$ *time for the case* $p<q+1$ *and* $O(M(n)+\lg m+\lg q')$ *time for the case of* $p=q+1$.

Proof. We note that $(2^p|x|)^{1/(q+1-p)} \le 2^p|x| \le 2^{np+m}$ and $\frac{\bar{a}}{1-\sqrt[p]{|x|}} \le 2^{n+mp}$, so $n_0 \le 2^{(n+m)(q'+2)} \le 2^{3q'(n+m)}$. For the case of $p<q+1$, it is enough to compute $\sqrt[q+1-p]{|x|}$ to q' relative precision, which takes $O(M(q)+m)$ time. For the case of $p=q+1$, we compute $\frac{1}{1-\sqrt[p]{|x|}}$ to n relative precision, which takes $O(M(n)+\lg m+\lg q')$ time. **Q.E.D.**

§58. Complexity of Computing n_1 **for** $p<q+1$ In the case of $p<q+1$, the complexity of computing n_1 is easy and shown as follows:

LEMMA 32. *If* $p<q+1$, *then* $n_1 \le 3q'2^n$. *Moreover, we can compute it in time* $O(q'M(n+\lg q'))$.

Proof. To bound n_1, we note that

$$|s_1(\mathbf{b}')|, |s_1^+(\mathbf{a})|, |s_1^-(\mathbf{a})|$$

are each bounded by $q'2^n$. Hence $|r| \leq 3q'2^n$. We can compute

$$r' = \frac{\lceil s_1(\mathbf{b}') \rceil - \frac{1}{2} \left\lfloor s_1^+(\mathbf{a}) \right\rfloor - 2 \left\lfloor s_1^-(\mathbf{a}) \right\rfloor}{q + 1 - p}$$

and

$$n_0 = \max\{\left\lceil a_{max}^+ \right\rceil, 2 \left\lceil a_{max}^- \right\rceil, \left\lceil b_{max}^- \right\rceil, r'\}.$$

Since $s_1(\mathbf{b}'), s_1^+(\mathbf{a}), s_1^-(\mathbf{a})$ has at most $(n + \lg q')$ bits, we can evaluate them to $(n+\lg q')$ relative bits to get $\lceil s_1(\mathbf{b}') \rceil, \left\lfloor s_1^+(\mathbf{a}) \right\rfloor, \left\lfloor s_1^-(\mathbf{a}) \right\rfloor$ in time $O(q'M(n+\lg q'))$ by Corollary 30. **Q.E.D.**

§59. Complexity of Computing n_1 for $p = q + 1$ However, for the case of $p = q + 1$, we need first analyze the complexity of finding the smallest k, which is shown in the following three lemmas:

LEMMA 33. *Let $\alpha = \sqrt[k]{a} - \sqrt[k]{b}$ where $a, b > 0$ are rational numbers of size t. Then $\lg |\alpha| \geq -(2t + 1)k^2$. If a, b are integers then $\lg |\alpha| \geq -(t + k)k$.*

Proof. We use the BFMSS rule [16] to compute $u(\sqrt[k]{a}) \leq 2^t$ and $\ell(\sqrt[k]{a}) \leq 2^t$, with the same bound when a is replaced by b. Hence $u(\alpha) = 2^{2t+1}$. Since the degree of α is $\leq k^2$, we get $|\alpha| \geq u(\alpha)^{-k^2}$ and $\lg |\alpha| \geq -(2t + 1)k^2$. When a, b are integers, we obtain $u(\sqrt[k]{a}) \leq 2^{t/k}$ and $\ell(\sqrt[k]{a}) = 1$, giving us the improvement stated. **Q.E.D.**

COROLLARY 34. *Let $\alpha = s_k(\mathbf{b}') - s_k(\mathbf{a})$, $\beta = \sqrt[k]{s_k(\mathbf{a})} - \sqrt[k]{s_k(\mathbf{b})}$. Then $\lg |\alpha| \geq -2(kn + \lg q')$, $\lg |\beta| \geq -(2q'(kn + \lg q') + 1)k^2$.*

Proof. We note that $s_k(\mathbf{a})$ and $s_k(\mathbf{b})$ have size at most $(kn+\lg q')$. **Q.E.D.**

LEMMA 35. *Deciding the sign of $s_k(\mathbf{b}') - s_k(\mathbf{a})$ takes $O(kn + \lg q')$ time and deciding the smallest k such that $s_k(\mathbf{b}') \neq s_k(\mathbf{a})$ takes $O(q'^2 n + q' \lg q')$ time.*

Proof. We can evaluate $s_k(\mathbf{b}') - s_k(\mathbf{a})$ to $2(kn + \lg q')$ relative precision to get the sign in time $O(kn + \lg q')$. In at most q' steps, we can find the smallest index k, which takes $O(q'^2 n + q' \lg q')$ time. **Q.E.D.**

Now we have the complexity of computing n_1:

LEMMA 36. *If $p = q+1$, then $n_1 \leq 2^{4q'^3 n}$. Moreover, we can compute it in time $O(M(q^3 n))$.*

Proof. To see the upper bound for n_1, we note from the definition in (3.10) and (3.11) that the numerator is at most $2^{2n+\lg q'+1}$ and the denominator is at least $2^{-q'^2(2(nq'+\lg q')+1)}$, hence $n_1 \leq 2^{4q'^3 n}$. Then we can compute n_1 to $4q'^3 n$ relative bits which will take $O(M(q'^3 n))$ time. **Q.E.D.**

§60. Complexity of Computing n_2 If we follow the scheme of the previous section, it remains to bound n_3. However, no finite bound is possible under that scheme. To see this, note that for $n > n_2$, although we know that $|xf(n)| < 1$, it can be arbitrarily close to 1, which implies n_3 can be arbitrarily large. Moreover, this difficulty only arises when $p = q+1$. Hence, when $p = q+1$, we shall modify the definition in (3.12).

LEMMA 37. *If $p = q + 1$, redefine $n_2 := \max\{n_0, n_1, n_2'\}$ where*

$$n_2' := \frac{a^+_{\max} + b^-_{\max}}{\sqrt[q]{1 + 2^{-m}} - 1} + b^-_{\max}. \tag{3.16}$$

Then,

$$1 - |xf(n_2)| \geq 2^{-2m}.$$

and hence $-\lg(|x|f(n_2)) \leq 2m$.

Let us note that for $0 < x < 1$ and $p \geq 1$,

$$(1+x)^{1/p} - 1 \geq \frac{x}{2p}. \tag{3.17}$$

LEMMA 38. $n_2 \leq 2^{q'(7q'^2 n+3m)}$. *The computation of* n_2 *takes* $O(q^3 n + m)$ *time.*

Proof. We have that $n_0 \leq 2^{4q'^3 n}$, and $n_1 \leq 2^{3q'(n+m)}$. When $p < q+1$, our lemma is clearly true in view of (3.12). When $p = q+1$, we need to estimate n_2' in (3.16). Using the bound (3.17), we conclude that $\sqrt[q]{1+2^{-m}} - 1 \geq 2^{-m-1-\lg q'}$. Hence

$$n_2' \leq \frac{2^{n+1}}{2^{-m-1-\lg q'}} + 2^n \leq 2^{n+3+m+\lg q'}.$$

But this bound is dominated by the bound for n_1. The computation time is dominated by the computation of n_0 and n_1 in the case of $p = q+1$, which takes at most $O(q^3 n + m)$ time. **Q.E.D.**

§61. Complexity of Computing n_3

LEMMA 39. *We have the bound*

$$\begin{aligned} n_3 &\leq 4^m \left(\ell + 1 + 2q'^2(7q'^2 n + 3m)2^{q'(7q'^2 n+3m)}\right) \\ &\leq 4^m \left(\ell + 2^{4q'(2q'^2 n+m)}\right) \end{aligned}$$

The computation of n_3 *takes time*

$$O(M(\lg(\ell) + q'^3 n + qm)).$$

Proof. Using the fact

$$|(a_i)_{n_2}| \leq (a_i + n_2)^{n_2} \leq (2^n + n_2)^{n_2}$$

and

$$|(b_j)_{n_2}| \geq |b_j|^{n_2} \geq 2^{-nn_2},$$

we first note that

$$\begin{aligned} 1 + \lg |u_{n_2}| &\leq 1 + 2q' n_2 \lg(2^n + n_2) \\ &\leq 1 + 2q' 2^{q'(7q'^2 n + 3m)}(n + q'(7q'^2 n + 3m)) \\ &\leq 2^{4q'(2q'^2 n + m)}. \end{aligned}$$

Moreover, if $f(n) \nearrow$, then p must be equal to $q + 1$. In this case we require $|x| < 1 - 2^{-m}$, i.e., $\lg |x| \leq -2^{-m}$. So

$$n_3 \leq 2^m(\ell + 2^{4q'(2q'^2 n + m)}).$$

If $f(n) \searrow$, by Lemma 37 we have $\lg |x f(n_2)| \leq -2^{-2m}$. Hence

$$n_3 \leq 4^m(\ell + 2^{4q'(2q'^2 n + m)}).$$

Therefore, the running time of computing n_3 is

$$O(M(\lg(\ell) + q'^3 n + qm))$$

assuming that we use bigfloat approximations. **Q.E.D.**

§62. Complexity of Computing the Summation S_N Let us define

$$N = 4^m \left(\ell + 2^{4q'(2q'^2 n + m)}\right). \tag{3.18}$$

Now we analyze the complexity of computing the approximation $\widetilde{S}_N$ such that $|\widetilde{S}_N - S_N| \leq 2^{-\ell-1}$ where $S_N = \sum_{k=0}^{N} t_k$. It is sufficient that for each k, $0 \leq k \leq N$, we compute an approximation $\widetilde{t}_k$ such that

$$|\widetilde{t}_k - t_k| \leq 2^{-\ell-1-\lg N}.$$

Noting that $t_k = u_k x^k$, $\lg |u_k| \leq 2kq' \lg(2^n + k)$ and $\lg |x^k| \leq km$, it is sufficient to compute u_k and x_k to relative $r + 2$ bits where

$$r = (\ell + 1 + \lg N + 2kq' \lg(2^n + k) + km).$$

LEMMA 40. *We can compute x^k to $r + 2$ relative precision in time $O(kM(r))$.*

Proof. To compute x^k to relative $r + 2$ bits, we can use two steps:

1. compute x to relative $(r + 2 + \lg(k + 1))$ bits and truncate,

2. multiply them (in linear order or binary tree order).

Note that x is a rational number $\frac{N}{D}$ with bit length m, so the first step can be done in time $O(M(r + \lg k) + \lg m)$. For the second step, if we do the multiplication in linear order, $x^2 = x \times x$ can be done in time $O(M(r + \lg k) + \lg m)$, $x^3 = x^2 \times x$ can be done in time $O(M(r+\lg k)+\lg(2m)), \ldots, x^k = x^{k-1} \times x$ can be done in time $O(M(r + \lg k) + \lg((k - 1)m))$, so the total running time is $O(kM(r + \lg k) + k \lg(km)) = O(kM(r))$. **Q.E.D.**

LEMMA 41. *We can compute u_k to $r + 2$ relative precision in time $O((k + q')M(r))$.*

Proof. Computing u_k to relative $r + 2$ bits can be done the following steps:

1. compute $(a_i)_k, (b_j)_k$ to relative $(r + 4 + \lg q')$ bits and truncate,

2. compute $P(k) = (a_1)_k(a_2)_k \cdots (a_p)_k$ to relative $(r+4)$ bits and truncate,

3. compute $Q(k) = (b_1)_k(b_2)_k \cdots (b_q)_k(1)_k$ to relative $(r+4)$ bits and truncate,

4. compute $\frac{P(k)}{Q(k)}$ to relative $r+2$ bits.

The first step can be done in time $O(kM(r+\lg q+\lg k)+k\lg(kn)) = O(kM(r))$. The second and third step can be done in time $O(q'M(r+\lg q')+kq'\lg(2^n+k)) = O(q'M(r))$, and the final step can be done in time $O(M(r)+kq'\lg(2^n+k)) = O(M(r))$. Therefore, computing u_k takes $O((k+q')M(r))$ time. **Q.E.D.**

COROLLARY 42. *The summation S_N takes $O(N^2M(\ell+q'N\lg N+Nm))$ time where N is defined in (3.18).*

Proof. Note that $r = O(\ell+q'N\lg N+Nm)$, so computing t_k takes at most $O(NM(\ell+q'N\lg N+Nm))$ time, hence the total running time for computing S_N is $O(N^2M(\ell+q'N\lg N+Nm))$. **Q.E.D.**

The upshot of these calculations yields:

THEOREM 43. *The general hypergeometric function H can be approximated in time that is singly exponential in q', n, m, and polynomial in ℓ.*

3.5 Argument Reduction, Parameter Pre-processing and Constants

3.5.1 Argument Reduction

An issue in the efficient evaluation of hypergeometric functions is the well-known problem of argument reduction. Thus the evaluation of $\sin(10^{22}) =$

$-0.8522008497671888017727...$ might well arise because its argument is automatically generated in some sequence of evaluations [70] [39] [40] [41].

Each hypergeometric series is generally valid within a bounded range, and the problem is to reduce a general argument to this range. Even when an argument is in the valid range, argument reduction can still be applied to achieve faster convergence. As noted in [64, p.145–147], argument reduction in trigonometric functions (the "additive type" of reductions) are prone to catastrophic errors, with the result that evaluating $\sin(10^{22})$ on many computers have widely divergent answers (some outright wrong).

Whenever we perform argument reductions, an error is introduced into the modified arguments. We need to bound the effects of this error. For instance, argument reduction for the trigonometric functions uses the fact that they have period 2π. By exploiting other properties, the arguments can be reduced to a range of size $\pi/2$. If r is the reduced argument corresponding to an original argument of x, we have

$$r = x - \frac{\pi}{2}\left\lfloor \frac{2}{\pi} x \right\rfloor .$$

But we can only compute an approximation $\widetilde{r}$ to r. Using a sufficiently accurate approximation to π, we can bound $|r - \widetilde{r}|$ by any desired error bound ε'. The choice of ε' can be deduced using the following lemma:

LEMMA 44. *For $\varepsilon > 0$, we have the following bounds:*

$$\begin{aligned}
|\sin(x+\varepsilon) - \sin x| &\leq \varepsilon \\
|\cos(x+\varepsilon) - \cos x| &\leq \varepsilon \\
|\tan(x+\varepsilon) - \tan x| &\leq 4\varepsilon, \qquad 0 \leq x \leq \pi/4, \varepsilon < \pi/12 \\
|\cot(x+\varepsilon) - \cot x| &\leq 2\varepsilon, \qquad \pi/4 \leq x \leq \pi/2, \varepsilon \leq \pi/4 \\
|\arcsin(x+\varepsilon) - \arcsin x| &\leq 2\varepsilon, \qquad |x| < 0.5, \varepsilon \leq 1/4 \\
|\arccos(x+\varepsilon) - \arccos x| &\leq 2\varepsilon, \qquad |x| < 0.5, \varepsilon \leq 1/4 \\
|\arctan(x+\varepsilon) - \arctan x| &\leq \varepsilon, \qquad |x| < 1 \\
|\log(x+\varepsilon) - \log x| &\leq \varepsilon/x, \qquad x > 0. \\
|\exp(x+\varepsilon) - \exp x| &\leq 2\varepsilon \exp(x), \qquad \varepsilon \leq \log(2).
\end{aligned}$$

Proof. We use the remainder form of the Taylor expansion,

$$f(x+h) = f(x) + hf'(x+\theta), \qquad 0 \leq \theta \leq h.$$

Then

$$\begin{aligned}
|\sin(x+\varepsilon)-\sin x| &= \varepsilon|\cos(x+\theta)| \le \varepsilon \\
|\cos(x+\varepsilon)-\cos x| &= \varepsilon|\sin(x+\theta)| \le \varepsilon \\
|\tan(x+\varepsilon)-\tan x| &= \varepsilon|\sec^2(x+\theta)| \le 4\varepsilon \\
|\cot(x+\varepsilon)-\cot x| &= \varepsilon|\csc^2(x+\theta)| \le 2\varepsilon \\
|\arcsin(x+\varepsilon)-\arcsin x| &= \varepsilon|\frac{1}{\sqrt{1-(x+\theta)^2}}| \le 2\varepsilon \\
|\arccos(x+\varepsilon)-\arccos x| &= \varepsilon|\frac{1}{\sqrt{1-(x+\theta)^2}}| \le 2\varepsilon \\
|\arctan(x+\varepsilon)-\arctan x| &= \varepsilon|\frac{1}{1+(x+\theta)^2}| \le \varepsilon \\
|\log(x+\varepsilon)-\log x| &= \varepsilon|1/(x+\theta)| \le \varepsilon/x \\
|\exp(x+\varepsilon)-\exp x| &= \varepsilon|\exp(x+\theta)| \le 2\varepsilon\exp(x)
\end{aligned}$$

Q.E.D.

Now we give the detailed reduction algorithms for most common elementary functions.

§63. Natural Log function If $x > 2$, we let $x = 2^k r$ where $k \in \mathbb{N}$ and $1 < r \le 2$. Then

$$\log(x) = k\log(2) + \log(r).$$

Here are the steps to approximate this expression:

1. First compute $k = \lfloor \log_2 x \rfloor$.
2. Compute $\widetilde{\log(2)}$ as an approximation of $\log(2)$ to absolute error $\le \varepsilon/(2k)$.
3. Compute $\widetilde{r}$ such that $|r - \widetilde{r}| \le \varepsilon/4$ where $r = x2^{-k}$.

4. Compute $\widetilde{\log(\widetilde{r})}$ as an approximation of $\log(\widetilde{r})$ to absolute error $\leq \varepsilon/4$.

5. Return $z = k\widetilde{\log(2)} + \widetilde{\log(\widetilde{r})}$.

It is easy to show, using Lemma 44, that this procedure is correct. That is, $|z - \log(x)| \leq \varepsilon$. Moreover, each of the steps is easily computed using the `Core Library`. Step 2 requires an approximation to the constant $\log(2)$, which we pre-compute (see Section 3.5.3).

§64. Exponential function Let $k = \lfloor x/\log(2) \rfloor$ and $r = x - k\log(2)$. Then

$$\exp(x) = 2^k \exp(r).$$

Note that $1 \leq r < 2$.

1. First, we compute k (the details of this computation are omitted, but will require a suitable approximation to $\log 2$).

2. Compute $\widetilde{r}$ as an approximation to $r = x - k\log 2$, to absolute error $\varepsilon 2^{-k-2}e^{-2}$.

3. Compute $\widetilde{\exp(\widetilde{r})}$ as an approximation to $\exp(\widetilde{r})$ to absolute error $\varepsilon 2^{-k-1}$.

4. Return $z = 2^k\widetilde{\exp(\widetilde{r})}$.

§65. Trigonometric functions reduction-trig To compute $\arcsin(x)$ when $0.5 < x \leq 1$, we use

$$\arcsin(x) = \frac{\pi}{2} - 2\arcsin\left(\sqrt{\frac{1-x}{2}}\right).$$

From Lemma 44, we see that it is sufficient to compute π to absolute error bound of $\varepsilon/2$ and compute $\sqrt{(1-x)/2}$ to absolute error bound of $\varepsilon/8$. A

similar reduction applies for $\arccos(x)$. For $\arctan(x)$ when $|x| > 1$, we use

$$\arctan(x) = \frac{\pi}{2} - \arctan\left(\frac{1}{x}\right).$$

Again, we need to compute $1/x$ to absolute error bound of $\varepsilon/2$. The cases for $\sin, \cos, \tan, \cot$ are even simpler.

3.5.2 Parameter Pre-processing

Hypergeometric parameters are sometimes artificially introduced in order to achieve the standard form of these series. For instance, one of the upper parameters in $x \cdot {}_2F_1(1, 1; 2; -x)$ $(= \log(1 + x))$ amounts to canceling the implicit lower parameter $b_0 = 1$. This leads to a factor $k!/k!$ in the kth term t_k. While mathematically harmless, this has major performance impact in the `Core Library` evaluation mechanism. The example of $\log(1 + x)$ also illustrates another improvement possible: the upper parameter of 1 with a lower parameter of 2 amounts to the factor $1/(k + 1)$ in the ratio t_{k+1}/t_k. Again, it is important not to evaluate this factor as $(1)_k/(2)_k$. More generally, whenever an upper and a lower parameter differ by an integer, cancellations occur and one can gain improvements in efficiency by recognizing this.

§66. Parameter Pre-processing Algorithm We outline a general algorithm for pre-processing the hypergeometric parameters. Let $a_1, a_2, , \ldots, a_p$ and $b_0, b_1, , \ldots, b_q$ be the upper and lower parameters of ${}_pF_q$. Note that we have added $b_0 = 1$ to the standard list of lower parameters.

(1) We first sort the a's and then the b's. Let $a_1 \leq a_2 \leq \cdots \leq a_p$ and $b_0 \leq b_1 \leq \cdots \leq b_q$ be the sorted result. In implementation, we use a simple insertion sort. The advantage of this is that the insertion sort of a sorted list of

length n only requires $n-1$ comparisons. Conventionally, parameter lists are given in sorted order.

(2) By a merge-like algorithm we eliminate common terms from both lists. Note that we still maintain the separate lists.

(3) Next we form the maximum number of (a_i, b_j) of an upper and a lower parameter where $a - i - b_i$ is an integer. Let us call two real numbers x, y **equivalent** if $x - y \in \mathbb{Z}$. Let (A_i, B_i) $(i = 1, \ldots, r)$ be the set of such equivalent pairs; these are called ab-pairs since A_i is an upper parameter and B_i a lower parameter. Their corresponding values A_i, B_i are deleted from the original parameter lists. It is easy to see that the maximum number r of ab-pairs is unique. However, the set of these pairs are not unique. To ensure the most efficient code, we must minimize the sum $\sum_{i=1}^{r} |A_i - B_i|$ because this is the number of linear factors that the pairs contribute to the term t_n. Below, we present a quadratic-time algorithm to solve this matching problem.

(4) We compute the successive terms t_n as follows: Let s_n be the term that is computed from the upper and lower parameter list in the usual way:

$$s_n = s_{n-1} \times f_n$$

where $f_n = (a_1 + n)(a_2 + n) \cdots (a_p + n)/(b_0 + n) \cdots (b_q + n)$. We then initialize t_n to s_n. Then for each pair (A, B) where $B - A = k \geq 1$, we update

$$t_n := t_n * \frac{A(A+1)\cdots(A+k-1)}{(A+n)\cdots(A+k+n-1)}.$$

If $A - B \geq 1$, there is an analogous factor. There is a special type of pair that can be further exploited: when A, B are multiples of halves (this can be generalized too). In case $A = \alpha/2$ and $B - A = k \geq 1$, then (A, B) contributes

the following factor to t_n:

$$\frac{\alpha(\alpha+2)\cdots(\alpha+2(k-1))}{(\alpha+2n)\cdots(\alpha+2(k+n-1))}.$$

In the `Core Library`, this formulation will again lead to expressions of smaller depth, and to a more efficient evaluation. The following table shows the speedup as a result of exploiting parameter reduction (using the standard series for $\log(1+x)$).

Number of digits	100	200	300	400	500
No preprocessing (sec)	1.	5.01	6.99	8.89	10.46
Parameter preprocessing (sec)	0.22	0.52	0.88	1.35	1.83
Speedup:	4.5	9.6	7.9	6.8	5.7

We give a solution to the following problem: given two lists $a_1, \ldots, a_p$ and $b_1, \ldots, b_q$ of p+q distinct numbers, compute a maximal set $\{(A_1, B_1), \ldots, (A_r, B_r)\}$ of ab-pairs such that $\sum_{i=1}^{r} |A_i - B_i|$ is minimized. We present an $O(n \log n)$ solution, by a reduction to sorting. Put the a_i's and b_j's into a common list and sort them. The sort key for any number x is the pair $key(x) = (x \mod 1, x)$. The comparison of two sort keys $key(x) : key(y)$ is lexicographic (first compare $x \mod 1 : y \mod 1$, and if there is a tie, we then compare $x; y$). This puts the a's and b's into groups based on their equivalence classes. Then, we march through the sorted lists of each equivalent classes:

§67. Minimum Matching Problem Consider the following problem[2] where we are given two sorted lists of real numbers, $(A_1 < A_2 < \cdots < A_m)$ and

[2] Although this might appear to be a known problem in the literature, we have not been able to find a reference.

$(B_1 < B_2 < \cdots < B_n)$. Assume $m \leq n$. We want to compute a set of m pairs $(A_1, B_{\alpha(1)}), \ldots, (A_m, B_{\alpha(m)})$ such that the sum

$$S = \sum_{i=1}^{m} |A_i - B_{\alpha(i)}| \tag{3.19}$$

is minimized where $(\alpha(1), \cdots, \alpha(n))$ is a permutation of $(1, \cdots, m)$. We give an $O(mn)$ time algorithm. Two pairs $(A_i, B_{\alpha(i)})$ and $(A_j, B_{\alpha(j)})$ are said to **cross** if $i < j$ and $\alpha(i) > \alpha(j)$. It is easy to see that if we "uncross" such a pair, we obtain a solution whose sum S in (3.19) is not more than the original. Consider the subproblems $P(i, j)$ comprising the input lists $(A_1, \ldots, A_i)$ and $(B_1, \ldots, B_j)$. Let $S(i, j)$ be the minimum value for subproblem $P(i, j)$. When $i = j$, the solution is unique in the obvious way. Otherwise, for $i < j$,

$$S(i, j) = \min\{S(i, j-1), S(i-1, j-1) + |A_i - B_j|\}.$$

Then, using standard dynamic programming, we can solve this problem in time $O(mn)$.

3.5.3 Mathematical Constants: Evaluation, File Formats

The hypergeometric evaluation algorithm requires arbitrarily precise constants. For instance, when doing argument reduction for trigonometric functions, we need π. For argument reduction for $\exp(x)$ and for $\log(1 + x)$ we need $\log 2$. For the error function $\operatorname{erf}(x)$, we need $1/\sqrt{\pi}$. While it is possible to compute these constants on the fly, we can improve performance by pre-computing these constants, storing them in files, and accessing them as needed. The following raw timing results give some idea about possible gains: we compare the number of seconds to compute π to a certain number of bits (using Machin's formula) versus the time it takes to read the same number of bits from a text file.

Bits	100	1000	3000	5000	7000	9000	10000	20000
On the fly (sec)	0.04	0.50	2.49	5.88	10.66	17.19	21.38	107.61
Pre-computed (sec)	0.01	0.01	0.01	0.01	0.02	0.02	0.03	0.11
Speedup	4	50	249	588	533	859	713	978

Thus, for 5000 bits or more, we can expect gains of up to three orders of magnitude. Hence, we now describe facilities to compute, to store and to read constants in file formats. A fundamental decision was to use text files rather than binary files as the former is human readable. The reason for this choice is that files admits better human interfaces, as an ordinary text editor can be used to enter and modify values. The main drawback is a constant factor overhead in storage as well as in speed. Storage is not an issue, considering that storage is practically free within the modest space requirements of our applications. The format supports both integer, floating point and rational number representations. Next, the base of the numbers can be binary, hexadecimal or decimal. The advantage of binary/hexadecimal is that conversion into the internal format of the `Core Library` takes linear time. The advantage of the decimal format is that they are directly comprehensible by humans (a useful fact for experimentation). Our files also allow comments, which also facilitates memorization and collaboration. The formal specification is distributed with the `Core Library` version 1.3 or higher. [3]

[3]Look under the directory `progs/fileIO`.

3.6 Integration of Hypergeometric Functions into the Core Library

Now we are ready to integrate hypergeometric functions into the Core Library using the new extension framework we developed in Section 2.4. The main function we provided for hypergeometric functions is

```
Expr hyper(const std::vector<BigRat>& A, const std::vector<BigRat>& B,
           const Expr& fx, const Expr& fz);
```

It is used to evaluate the hypergeometric series

$$ {}_pF_q = f_z F(a_1, a_2, ..., a_p, b_1, b_1, b_2, ..., b_p; f_x). $$

The upper and lower parameters of the series are stored in the vectors A and B respectively; fz and fx are scalars. We also defined the interfaces for the following common elementary functions:

$$ \log, \exp, \mathrm{erf}, \sin, \cos, \tan, \arccos, \arcsin, \arctan $$

which are just inline functions to call hyper().

Like other functions, sum(), rootOf(), in the implementation of hyper() a new type of node HyperRep is constructed. Inside HyperRep, we implemented most functionalities. HyperRep is designed as a unary node. With the new extension mechanism in the Core Library, we need to overload the following functions: compute_filter(), compute_sgn(), compute_uMSB(), compute_lMSB(), comput_r_approx() and compute_a_approx(). Actually, we only need implement compute_a_approx() which basically uses the algorithms we developed in Section 3.2. While we can easily compute the approximation of the summation

using `Expr` class, a more efficient method is to use the newly developed `sum()` function to avoid build large expression DAGs.

Argument reduction and parameter pre-processing are done in the constructor of `HyperRep`. The constants used in the argument reduction are read from pre-computed files when it is available, otherwise computed on the fly.

To obtain more efficiency, we design some leaf nodes to compute those mathematical constants using fast algorithms. For example, we use AGM to compute the approximations of π and $\log 2$.

Since we have no root bound for hypergeometric functions, the escape bound inside the `Rootbd` class is used when it is needed. Hence users should realize that in the presence of hypergeometric functions, the sign computation is conditional, i.e., it is only guaranteed to some finite precision.

3.7 Final Remarks and Open Problems

In work closely related to the new results of this chapter, van der Hoeven [90, 91] presented fast algorithms for evaluating holonomic functions $f(x)$. Such an $f(x)$ satisfies a linear differential equation $\sum_{i=0}^{k} P_i(x) f^{(i)}(x) = 0$ where $P_i(x) \in \mathbb{Z}[x]$. For instance, if $f(x) = {}_2F_1(a, b; c; x)$ then it satisfies the equation

$$x(1-x)f''(x) + (c - x(1+a+b))f'(x) - abf(x) = 0.$$

Van der Hoeven's setting is more general than ours in two ways: first, hypergeometric functions are holonomic and second, he treats complex functions which live on Riemann surfaces. He shows that $f(x)$ can be approximated to absolute n-bits in time $O(M(n \log^2 n))$. But this complexity bound is a "local result" in the sense that f is fixed and x restricted to a local neighborhood. In contrast,

our complexity bounds are global results: x is unbounded and our functions f are specified by input parameters $\mathbf{a}, \mathbf{b}$. As seen in Section 3.4, global bounds can be nontrivial with subtle difficulties. In any case, it remains a challenge to give a global complexity bound in van der Hoeven's setting.

We have shown for the first time that the general hypergeometric function is absolutely approximable. Our complexity bound for H is the first step towards gauging the inherent complexity of H or ${}_pF_q$ (for fixed p, q), which is useful for many applications (see, e.g., [48]). In particular, we are interested in lower bounds for H and its relation to other important problems in algebraic complexity.

Another natural question is the relative approximation of H. The **zero problem for** H is this:

(ZD): Given rational $\mathbf{a}, \mathbf{b}, x$, is $H(\mathbf{a}; \mathbf{b}; x) = 0$?

This seems to be a difficult problem and is currently wide open. For instance, Beukers has shown that

$$ {}_2F_1(1 - 3a, 3a; a; 1/2) = 2^{2-3a} \cos(\pi a). $$

By setting $a = 1/2$, we obtain ${}_2F_1(-\frac{1}{2}, \frac{3}{2}; \frac{1}{2}; \frac{1}{2}) = 0$.

So the issue is to detect identities of this sort automatically. It is shown in [94] that a real function f is relatively approximable iff it is absolutely approximable and the zero problem for f is decidable. Since H is absolutely approximable, we conclude:

LEMMA 45. *The general hypergeometric problem is relatively approximable iff the zero problem for H is decidable.*

We can consider related problem:

(RD): Given rational $\mathbf{a}, \mathbf{b}, x$, is $H(\mathbf{a}; \mathbf{b}; x)$ rational?

(RE): Given rational $\mathbf{a}, \mathbf{b}, x, r$, is $H(\mathbf{a}; \mathbf{b}; x) = r$?

LEMMA 46. *The problem (ZD) can be reduced to the special case of (RE) in which the parameters* $\mathbf{a}, \mathbf{b}$ *are positive.*

In fact, if $F = {}_pF_q(\mathbf{a}; \mathbf{b}; x) = S_n + R_n$ where $R_n = \sum_{k \geq n} t_k$, then

$$R_n = t_{np+1} F_{q+1}(\mathbf{a} + n, 1; \mathbf{b} + n, 1 + n; x)$$

For instance, Beuker's example above can be transformed in this way to

$${}_2F_1(-\frac{1}{2}, \frac{3}{2}; \frac{1}{2}; \frac{1}{2}) = 1 + t_1 \cdot {}_3F_2(\frac{1}{2}, \frac{5}{2}, 1; \frac{3}{2}, 2; \frac{1}{2})$$

where $t_1 = -3/4$. Hence,

$${}_3F_2(\frac{1}{2}, \frac{5}{2}, 1; \frac{3}{2}, 2; \frac{1}{2}) = 3/4.$$

Chapter 4

Non-asymptotic Error Analysis of AGM Algorithm

Brent showed that an n-bit approximation of an elementary function f may be evaluated in $O(M(n)\lg^2(n))$ operations. His algorithms depend on the theory of elliptic integrals, and uses the arithmetic-geometric mean (AGM) iteration and ascending Landen transformations. However, he only gave the asymptotic error analysis and the precision for each operations in the algorithms are not given explicitly. Thus, they cannot be used for direct implementation. In this chapter, we present a non-asymptotic error analysis for these AGM-based algorithms. With this analysis and the new mechanism we introduced in Chapter 2, we can implement and incorporate many elementary functions into the `Core Library`.

Overview of this chapter In Section 4.1, we describe the AGM algorithm and some of its properties. We give an error analysis for the AGM algorithm in Section 4.2. In Section 4.3, we first present Brent's AGM-based algorithm for computing π, then we modify it with explicit precision given for each operation. In Section 4.4, we describe the Brent's method for computing the exponen-

tial and logarithm functions first, then with non-asymptotic error analysis we present our modified algorithms which can be directly implemented.

4.1 Arithmetic-geometric Mean Iteration

Arithmetic-geometric mean (AGM) iteration was known to Gauss [33]. Let a_0, b_0 be two positive numbers. We iterate as follows:

$$a_{i+1} = \frac{a_i + b_i}{2} \qquad \text{(arithmetic mean)}$$

and

$$b_{i+1} = \sqrt{a_i b_i} \qquad \text{(geometric mean)}$$

for $i = 0, 1, \ldots$.

§68. Second-order convergence AGM converges very fast. If $b_i \ll a_i$, then

$$\frac{b_{i+1}}{a_{i+1}} = \frac{2\sqrt{b_i/a_i}}{1 + b_i/a_i} \simeq 2\sqrt{\frac{b_i}{a_i}},$$

so only about $|\lg(a_0/b_0)|$ iterations are required before b_i/a_i is of order 1. Once a_i and b_i get close, the convergence is second order, since for $b_i/a_i = 1 - \epsilon_i$, then

$$\epsilon_{i+1} = 1 - b_{i+1}/a_{i+1} = 1 - 2(1-\epsilon_i)^{\frac{1}{2}}/(2-\epsilon_i) = \epsilon_i^2/8 + O(\epsilon_i^3).$$

§69. Monotonicity of AGM From the definitions of a_{i+1} and b_{i+1}, we have

$$a_{i+1} = \frac{a_i + b_i}{2} \geq \sqrt{a_i b_i} = b_{i+1}$$

for $i = 0, 1, \ldots$. Hence, if $i \geq 1$, we have

$$a_{i+1} = \frac{a_i + b_i}{2} \leq \frac{a_i + a_i}{2} = a_i$$

and

$$b_{i+1} = \sqrt{a_i b_i} \geq \sqrt{b_i b_i} = b_i,$$

i.e., $\{a_i\}$ is monotone decreasing and $\{b_i\}$ is monotone increasing. Thus, for $i \geq 1$, we have

$$a_1 \geq \cdots \geq a_i \geq a_{i+1} \geq b_{i+1} \geq b_i \geq \cdots \geq b_1. \tag{4.1}$$

Without loss of generality, we assume $a_0 \geq b_0$. Then we have

$$a_0 \geq a_1 \geq \cdots a_i \geq a_{i+1} \geq b_{i+1} \geq b_i \geq \cdots \geq b_1 \geq b_0.$$

for $i = 1, 2, \ldots$.

§70. Limit of AGM By monotone convergence principle, $\{a_i\}$ is a convergent sequence. Let A be the limit of $\{a_i\}$, i.e.,

$$A = \lim_{i \to \infty} a_i.$$

Note that $\lim_{i\to\infty} b_i = \lim_{i\to\infty} a_i$ and

$$b_0 \leq A \leq a_0. \tag{4.2}$$

4.2 Error Analysis of AGM

The algorithms that we are going to present in the rest of this chapter depend on computing A, i.e., the limit of $\{a_i\}$. Now we present the error analysis of computing $\lim_{i\to\infty} a_i$ using the AGM iteration.

From (4.1), we have

$$a_i \geq \lim_{i \to \infty} a_i \geq b_i$$

for $i = 1, 2, \ldots$. Hence,

$$|a_i - \lim_{i\to\infty} a_i| \leq |a_i - b_i|.$$

If $|a_i - b_i| \leq 2^{-n}$, then $|a_i - \lim_{i\to\infty} a_i| \leq 2^{-n}$, i.e., we can use a_i as an approximation of $\lim_{i\to\infty} a_i$ with absolute precision n. Since the AGM iteration converges with order 2 (§68), we need $\lg(n) + O(1)$ iterations. The following lemma gives us the exact number of iterations we need in order to guarantee $|a_i - b_i| \leq 2^{-n}$:

LEMMA 47. *Let $\delta_i = a_i - b_i$, then*

$$\delta_{i+1} \leq \frac{1}{8b_0}\delta_i^2, \tag{4.3}$$

for $i = 0, 1, \ldots$. Moreover, let

$$C_1(n) = \log_2(n + 3 + \log_2 b_0) - \log_2\left(3 + \log_2 \frac{b_0}{\delta_0}\right),$$

If $\frac{b_0}{a_0} \geq \frac{1}{9}$, then after $i \geq C_1(n)$ iterations, $\delta_i \leq 2^{-n}$. If $\frac{b_0}{a_0} \geq \frac{1}{5}$, then after $i \geq \log_2(n + 3 + \log_2 b_0)$ iterations, $\delta_i \leq 2^{-n}$.

Proof. Note that

$$\begin{aligned}
\delta_i^2 = (a_i - b_i)^2 &= (a_i + b_i)^2 - 4a_i b_i \\
&= 4a_{i+1}^2 - 4b_{i+1}^2 \\
&= 4(a_{i+1} + b_{i+1})(a_{i+1} - b_{i+1}) \\
&\geq 8b_0\delta_{i+1}.
\end{aligned}$$

Hence,

$$\delta_{i+1} \leq \frac{1}{8b_0}\delta_i^2.$$

Moreover,

$$\begin{aligned}
\delta_i &\leq \frac{1}{8b_0}\delta_{i-1}^2 \\
&\leq \frac{1}{8b_0}\left(\frac{1}{8b_0}\delta_{i-2}^2\right)^2 \\
&= \left(\frac{1}{8b_0}\right)^3 \delta_{i-2}^4 \\
&\leq \cdots \\
&\leq \left(\frac{1}{8b_0}\right)^{2^i-1} \delta_0^{2^i}.
\end{aligned}$$

In order to compute $\delta_i \leq 2^{-n}$, it suffices to have

$$\left(\frac{1}{8b_0}\right)^{2^i-1} \delta_0^{2^i} \leq 2^{-n}.$$

If $\frac{\delta_0}{8b_0} < 1$, i.e., $b_0 > \frac{1}{9}a_0$, then

$$i \geq \log_2\left(n + 3 + \log_2 b_0\right) - \log_2\left(3 + \log_2 \frac{b_0}{\delta_0}\right).$$

If $b_0 \geq \frac{1}{5}a_0$, then we can choose

$$i \geq \log_2\left(n + 3 + \log_2 b_0\right)$$

since $\log_2\left(3 + \log_2 \frac{b_0}{\delta_0}\right) \geq 0$.

Q.E.D.

However, the above analysis assumes that the initial values are exact and arithmetic operations in each iteration can be performed exact. In our setting, we use bigfloat computation. We would like to compute an approximation $\widetilde{a_i}$ of a_i. To obtain n bit accuracy in a_i, we need work out the precision that we need for the initial values and each arithmetic operations. The following lemma gives the error propagation rules for the AGM iterations:

LEMMA 48. *Let $a > 0$ and $b > 0$.*

(i) If $c = (a+b)/2$, to guarantee p relative bits in c, it suffices to guarantee $p+1$ relative bits in a and b and perform the addition in $p+2$ bits relative precision.

(ii) If $c = \sqrt{ab}$, to guarantee p relative bits in c, it suffices to guarantee $p+1$ relative bits in a and b and $p+2$ relative bits in the performing multiplication and square root extraction.

Proof. Assume $\widetilde{a} = a(1+\delta)$, $\widetilde{b} = b(1+\delta')$.

(i) If $\delta \leq 2^{-(p+1)}$, $\delta' \leq 2^{-(p+1)}$, $\rho_+ \leq 2^{-(p+2)}$, then we have

$$\begin{aligned}
\widetilde{c} &= \widetilde{\frac{\widetilde{a}+\widetilde{b}}{2}} \\
&= \frac{a(1+\delta)+b(1+\delta')}{2}(1+\rho_+) \\
&\leq \frac{a+b}{2}(1+2^{-(p+1)})(1+2^{-(p+2)}) \\
&\leq c(1+2^{-p}).
\end{aligned}$$

(ii) If $\delta \leq 2^{-(p+1)}$, $\delta' \leq 2^{-(p+1)}$, $\rho_\times \leq 2^{-(p+2)}$ and $\rho_+ \leq 2^{-(p+2)}$,

$$\begin{aligned}
\widetilde{c} &= \widetilde{\sqrt{\widetilde{a}\widetilde{b}}} \\
&= \sqrt{ab(1+\delta)(1+\delta')(1+\rho_\times)}(1+\rho_{\sqrt{}}) \\
&\leq \sqrt{ab}\sqrt{(1+\delta)(1+\delta')}(1+\frac{\rho_\times}{2})(1+\rho_{\sqrt{}}) \\
&\leq \sqrt{ab}(1+2^{-(p+1)})(1+2^{-(p+3)})(1+2^{-(p+2)}) \\
&\leq c(1+2^{-p}).
\end{aligned}$$

Q.E.D.

From the above lemma, we can see that in the AGM iteration, if a_i, b_i have $p+1$ bit relative precision and we perform the addition, multiplication and square root in $p+2$ relative precision (divide by 2 can be done exactly), then

a_{i+1} and b_{i+1} have p bits relative precision, i.e., 1 bit precision is lost in each iteration. Thus, we have the following lemma:

LEMMA 49. *If* $0 \leq b_0 \leq a_0 \leq 1$ *and* $\frac{b_0}{a_0} \geq \frac{1}{9}$, *then it suffices to approximate* a_0, b_0 *within relative* $n+1+C_1(n+1)$ *bits and guarantee* $n+2+C_1(n+1)-i$ *bits relative precision in the addition, multiplication and square root operations in* i*th iteration in order to compute* $\lim_{n\to\infty} a_n$ *to absolute* n *bits using the AGM iteration, where* $C_1(n)$ *is given in Lemma 47.*

Proof. By Lemma 47, if $\frac{b_0}{a_0} \geq \frac{1}{9}$, in $C_1(n+1)$ iterations, we can guarantee $|a_i - b_i| \leq 2^{-(n+1)}$, hence

$$|a_i - \lim_{i\to\infty} a_i| \leq 2^{-(n+1)}.$$

By Lemma 48, if we approximate a_0, b_0 within $n+1+C_1(n+1)$ bits relative precision and perform the addition, multiplication and square root operations in $n+2+C_1(n+1)-i$ in i-th iteration, then after $C_1(n+1)$ iteration, the computed a_i and b_i have $n+1$ bits relative precision, i.e.,

$$|\widetilde{a_i} - a_i| \leq |a_i| \cdot 2^{-(n+1)} \leq 2^{-(n+1)}.$$

Hence,

$$|\widetilde{a_i} - \lim_{i\to\infty} a_i| \leq |\widetilde{a_i} - a_i| + |a_i - \lim_{i\to\infty} a_i| \leq 2^{-n}.$$

Q.E.D.

4.3 Fast Multiple-precision Evaluation of π

The classical methods for evaluating π to precision n take time $O(n^2)$. Brent described a $O(M(n)\log^2(n))$ method using Machin's formula in [9]. Asymptotically the fastest known methods require time $O(M(n)\log(n))$. Brent presented

such a method in [10] which is based on the AGM iteration and Legendre's Identity. In this section, we present this algorithm with non-asymptotic analysis.

Assume $a_0 = 1$ and $b_0 = \cos\alpha$ for some α. If $A(\alpha) = \lim_{i\to\infty} a_i = \lim_{i\to\infty} b_i$, then

$$A(\alpha) = \frac{\pi}{2F(\alpha)}, \tag{4.4}$$

where $F(\alpha)$ is the complete elliptic integral of the first kind, i.e.,

$$F(\alpha) = \int_0^{\pi/2} \frac{d\theta}{\sqrt{1 - \sin^2\alpha \sin^2\theta}}.$$

(A simple proof of (4.4) is given in [8, 60].) Also if $c_0 = \sin\alpha$, $c_{i+1} = a_i - a_{i+1}$, $i = 0, 1, \ldots,$ then

$$\sum_{i=0}^{\infty} 2^{i-1} c_i^2 = 1 - \frac{E(\alpha)}{F(\alpha)} \tag{4.5}$$

where $E(\alpha)$ is the complete elliptic integral of the second kind, i.e.,

$$E(\alpha) = \int_0^{\pi/2} \sqrt{1 - \sin^2\alpha \sin^2\theta}\, d\theta$$

Both (4.4) and (4.5) were known by Gauss. Let $t_i = \frac{1}{2} - \sum_{j=0}^{i} 2^{j-1} c_j^2$ for $i \geq 0$, and $T(\alpha) = \lim_{i\to\infty} t_i$. Then

$$T(\alpha) = \frac{E(\alpha)}{F(\alpha)} - \frac{1}{2}. \tag{4.6}$$

§71. Legendre's Identity and Gauss-Legendre Method The following Legendre's Identity is used in Brent's method:

$$F(\alpha)E(\alpha') + F(\alpha')E(\alpha) - F(\alpha)F(\alpha') = \frac{\pi}{2}, \tag{4.7}$$

where $\alpha + \alpha' = \frac{\pi}{2}$. Taking $\alpha = \alpha' = \pi/4$, we obtain

$$2E(\pi/4)F(\pi/4) - F^2(\pi/4) = \pi/2. \tag{4.8}$$

Thus, from (4.4), (4.6) and (4.8), we have

$$\pi = A^2(\alpha)/T(\alpha). \tag{4.9}$$

Let $a_0 = 1$, $b_0 = c_0 = \frac{\sqrt{2}}{2}$. Then $t_0 = \frac{1}{4}$. Using (4.9) and the AGM iteration, we obtain the following algorithm evaluating π:

```
BRENT'S ALGORITHM FOR COMPUTING π
  // A = a_i, B = b_i, T = t_i, X = 2^i
  A ← 1; B ← 2^(-1/2); T ← 1/4; X ← 1;
  while (A − B > 2^(−n)) {
      Y ← A; A ← (A + B)/2; B ← √(BY);
      T ← T − X(A − Y)^2; X ← 2X;
  }
  return A^2/T;
```

Note that $T(\alpha)$ is a summation. Hence this AGM-based algorithm for computing π is not self-correcting, i.e., we cannot start with low precision and increase it. Brent showed that to obtain precision n, it is necessary to work with precision $n+O(\lg\lg(n))$. But no constant was given explicitly in his asymptotic analysis.

We propose the following modified algorithm using the `Kernel` class that we described in Chapter 2.

MODIFIED ALGORITHM $Pi()$ FOR COMPUTING π TO RELATIVE PRECISION n

$c \leftarrow C_1(n+6)$; $p \leftarrow n+6+c$; $pp \leftarrow p+1$;

$A \leftarrow 1$; $B \leftarrow sqrt(2,p)/2$; $T \leftarrow \frac{1}{4}$;

`for (i=0; i<c; ++i) {`

$Y \leftarrow A$; $A \leftarrow add(A,B,pp)$; $A \leftarrow div\_2exp(A,1)$;

$B \leftarrow mul(B,Y,pp)$; $B \leftarrow sqrt(B,pp)$;

$P \leftarrow sub(A,Y,pp)$; $P \leftarrow mul(P,P,pp)$; $P \leftarrow mul\_2exp(P,i)$;

$T \leftarrow sub(T,P,pp)$; $pp \leftarrow pp-1$;

}

$A \leftarrow mul(A,A,n+4)$;

`return` $div(A,T,n+2)$;

All these operations (`add(A, B, p)`, `sub(A, B, p)`, `mul(A, B, p)`, `sqrt(B, p)`) in above algorithm assume that p is a specified relative precision.

From Lemma 4 and Lemma 7, we know that to guarantee n relative bits in computing π, it is sufficient to guarantee $n+4$ relative bits in $A(\alpha)$ and $n+2$ relative bits in $T(\alpha)$ and to perform the multiplication in $n+4$ and division in $n+2$ relative precision. Since $A(\alpha) \geq b_0 = \frac{\sqrt{2}}{2} \geq \frac{1}{2}$, $T(\alpha) = A(\alpha)^2/\pi \geq \frac{1}{8}$, the lower bound of $\lg|A(\alpha)|$ is -1 and the lower bound of $\lg|T(\alpha)|$ is -3. By Lemma 1 (ii), it is enough to guarantee $n+5$ absolute precision $A(\alpha)$ and $n+5$ absolute bits in $T(\alpha)$. While Lemma 49 can show the above modified algorithm can approximate $A(\alpha)$ to $n+5$ absolute bits, it remains to show that it compute $T(\alpha)$ within $n+5$ absolute bits:

LEMMA 50. *The modified Algorithm $Pi()$ computes $T(\alpha)$ within $n+5$ absolute precision.*

Proof. We can first estimate the upper bound of c_i using Lemma 47:

$$\begin{aligned} c_{i+1} &= a_i - a_{i+1} \\ &= \frac{a_i - b_i}{2} \\ &\leq \left(\frac{1}{8}\right)^{2^i-1} \cdot (a_0 - b_0)^{2^i} \\ &\leq \left(\frac{1}{8}\right)^{2^i-1} \cdot \left(\frac{1}{2}\right)^{2^i} \quad (\text{since } a_0 - b_0 \leq 1/2) \\ &= 2^{-(2^{i+2}-4)} \end{aligned}$$

After $c = C_1(n+6)$ iterations, we have

$$|t_c - \lim_{i\to\infty} t_i| = |\sum_{j=c+1}^{\infty} 2^{j-1} c_j^2| \leq 2^{-2^{c+1}} \leq 2^{-(2n)}.$$

Let $\widetilde{a_i}$ be the approximation of a_i in i-th iteration, then from Lemma 49, we have $\widetilde{a_i} \leq a_i(1 + 2^{-(p-i)})$ where $p = n + 6 + c$. Hence

$$\begin{aligned} \widetilde{c_i} &= (\widetilde{a_{i-1}} - \widetilde{a_i})(1 + \rho_-) \\ &\leq (a_{i-1} - a_i)(1 + 2^{-(p-i)+1})(1 + 2^{-(p+1-i)}) \\ &\leq c_i(1 + 2^{-(p-i)+2}). \end{aligned}$$

Thus, $\widetilde{c_i}^2 \leq c_i^2(1 + 2^{-(p-i)+2})^2 \leq c_i^2(1 + 2^{-(p-i)+4})$. After c iterations, we have

$$|\widetilde{t_c} - t_c| = \sum_{j=0}^{c} 2^{j-1}(\widetilde{c_j} - c_j)^2 \leq \sum_{j=0}^{c} 2^{j-1} c_j^2 2^{-2(p-i)+8} \leq 2^{-2p} \leq 2^{-2n}$$

Therefore

$$|\widetilde{t_c} - \lim_{i\to\infty} t_i| \leq 2^{-2n)} + 2^{-2n)} \leq 2^{-(n+5)}.$$

Q.E.D.

4.4 Fast Evaluation of Exponential and Logarithm Functions

Brent's fast evaluation algorithms of elementary functions are based on elliptic integral theory.

4.4.1 Elliptic Integrals

Elliptic integrals of the first and second kind are defined by

$$F(\psi, \alpha) = \int_0^{\psi} \frac{d\theta}{\sqrt{1 - \sin^2 \alpha \sin^2 \theta}} \tag{4.10}$$

and

$$E(\psi, \alpha) = \int_0^{\psi} \sqrt{1 - \sin^2 \alpha \sin^2 \theta} d\theta \tag{4.11}$$

The complete elliptic integrals, $E(\pi/2, \alpha)$ and $F(\pi/2, \alpha)$, are simply written as $E(\alpha)$ and $F(\alpha)$, respectively, as seen in the previous section.

§72. Small Angle Approximation From (4.10) it is clear that

$$F(\psi, \alpha) \geq \int_0^{\psi} d\theta = \psi$$

and

$$F(\psi, \alpha) \leq \int_0^{\psi} \frac{d\theta}{\sqrt{1 - \sin^2 \alpha}} = \frac{\psi}{\cos \alpha} \leq \frac{\psi}{1 - \alpha^2/2} \leq \psi(1 + \alpha^2) \tag{4.12}$$

as $|\alpha| \to 0$.

§73. Large Angle Approximation From (4.10),

$$F(\psi, \alpha) \leq \int_0^{\psi} \frac{d\theta}{\sqrt{1 - \sin^2 \theta}} = F(\psi, \pi/2) \tag{4.13}$$

If $0 \le \psi \le \psi_0 < \pi/2$ and $|\pi/2 - \alpha| \to 0$, then

$$\begin{aligned}
\frac{\sqrt{1-\sin^2\theta}}{\sqrt{1-\sin^2\alpha\sin^2\theta}} &= \frac{\sqrt{1-\sin^2\theta}}{\sqrt{\cos^2\theta+\sin^2\theta(1-\sin^2\alpha)}} \\
&= \frac{1}{\sqrt{1+\tan^2\theta\cos^2\alpha}} \\
&\ge 1-\tan^2\theta\cos^2\alpha \\
&\ge 1-C(\pi/2-\alpha)^2
\end{aligned}$$

for $0 \le \theta \le \psi$ and $C = \tan^2\psi_0$. Hence,

$$F(\psi,\alpha) \ge F(\psi,\pi/2)\left(1-C(\pi/2-\alpha)^2\right) \tag{4.14}$$

Also, we note that

$$F(\psi,\pi/2) = \log\tan(\pi/4+\psi/2). \tag{4.15}$$

§74. Ascending Landen Transformation Let α_i, ψ_i be two sequences satisfying

$$\begin{aligned}
&0 < \alpha_i < \alpha_{i+1} < \pi/2, \\
&0 < \psi_{i+1} < \psi_i < \pi/2, \\
&\sin\alpha_i = \tan^2(\alpha_{i+1}/2),
\end{aligned} \tag{4.16}$$

and

$$\sin(2\psi_{i+1}-\psi_i) = \sin\alpha_i\sin\psi_i. \tag{4.17}$$

Then

$$F(\psi_{i+1},\alpha_{i+1}) = \frac{1+\sin\alpha_i}{2}F(\psi_i,\alpha_i). \tag{4.18}$$

If $s_i = \sin\alpha_i$ and $v_i = \tan(\psi_i/2)$, then (4.16) gives

$$s_{i+1} = \frac{2\sqrt{s_i}}{1+s_i}, \tag{4.19}$$

and (4.17) gives

$$v_{i+1} = \frac{W_i}{1 + \sqrt{1 + {W_i}^2}}, \tag{4.20}$$

where

$$W_i = \tan \psi_{i+1} = \frac{v_i + W_i'}{1 - v_i W_i'}, \tag{4.21}$$

$$W_i' = \tan(\psi_{i+1} - \psi_i/2) = \frac{W_i''}{1 + \sqrt{1 - {W_i''}^2}}, \tag{4.22}$$

and

$$W_i'' = \sin(2\psi_{i+1} - \psi_i) = \frac{2 s_i v_i}{1 + v_i^2}. \tag{4.23}$$

(4.18) becomes

$$F(\psi_{i+1}, \alpha_{i+1}) = \frac{1 + s_i}{2} F(\psi_i, \alpha_i). \tag{4.24}$$

It is clear that s_i can be computed using (4.19) and v_i can be computed using (4.20)-(4.23) recursively.

§75. Monotonicity of Ascending Landen Transformation It is interesting to see how these transformation changed when i is increasing. Since $0 < \psi_{i+1} < \psi_i < \pi/2$,

$$v_{i+1} < v_i < 1.$$

From (4.23), we have

$$\frac{\partial W_i''}{\partial v_i} = \frac{2(1 - v_i^2)}{1 + v_i^2} > 0,$$

hence $W_{i+1}'' < W_i''$. From (4.22), we have

$$\frac{\partial W_i'}{\partial W_i''} = \frac{1}{1 + \sqrt{1 - {W_i''}^2}} - \frac{W_i'' / \left(2\sqrt{1 - {W_i''}^2}\right)}{\left(1 + \sqrt{1 - {W_i''}^2}\right)^2} > 0,$$

hence $W'_{i+1} < W'_i$. From (4.21), we have

$$\frac{\partial W_i}{\partial W'_i} = \frac{1 + {W'_i}^2}{(1 - v_i W'_i)^2} > 0,$$

and

$$\frac{\partial W_i}{\partial v_i} = \frac{1 + {v_i}^2}{(1 - v_i W'_i)^2} > 0,$$

hence $W_{i+1} < W_i$.

4.4.2 Brent's method

§76. $U_k(m)$ **and** $T_k(m)$ We can apply the ascending Landen transformation (4.24) with $i = 0, 1, \cdots, k-1$ and $\psi_0 = \pi/2$ which gives

$$F(\psi_k, \alpha_k) = F(\alpha_0) \prod_{i=0}^{k-1} \frac{1 + s_i}{2} \tag{4.25}$$

where $F(\alpha_0)$ may be evaluated using (4.4) and the AGM iteration. On the other hand, from (4.13), (4.14) and (4.15), we have

$$F(\psi_k, \alpha_k) \le \log \tan(\pi/4 + \psi_k/2) \tag{4.26}$$

and

$$F(\psi_k, \alpha_k) \ge \log \tan(\pi/4 + \psi_k/2) \left(1 - C(\pi/2 - \alpha_k)^2\right). \tag{4.27}$$

Define the functions

$$U_k(m) = F(\alpha_0) \prod_{i=0}^{k-1} \frac{1 + s_i}{2} \tag{4.28}$$

and

$$T_k(m) = \tan(\pi/4 + \psi_k/2) = \frac{1 + v_k}{1 - v_k}. \tag{4.29}$$

Then, we have

$$\log T_k(m) \ge U_k \ge \log T_k(m) \left(1 - C(\pi/2 - \alpha_k)^2\right).$$

Taking the limit as k tends to ∞, we have

$$\lim_{k\to\infty} U_k(m) = \lim_{k\to\infty} \log T_k(m) = \log \lim_{k\to\infty} T_k(m).$$

Define

$$U(m) = \lim_{k\to\infty} U_k(m) \tag{4.30}$$

and

$$T(m) = \lim_{k\to\infty} T_k(m), \tag{4.31}$$

we obtain the following fundamental identity:

$$U(m) = \log T(m). \tag{4.32}$$

This identity is used to evaluate the exponential and logarithm functions.

Using (4.19) - (4.23), we can evaluate $U_k(m)$ and $T_k(m)$ as follows:

ALGORITHM FOR COMPUTING $U_k(m)$

$A \leftarrow 1;\ B \leftarrow \sqrt{1-m};$

`while` $(A - B > 2^{-n/2})$ {

$\quad C \leftarrow (A+B)/2;\ B \leftarrow \sqrt{AB};\ A \leftarrow C;$

}

$A \leftarrow \pi/(A+B);\ S \leftarrow \sqrt{m};$

`while` $(1 - S > 2^{-n/2})$ {

$\quad A \leftarrow A(1+S)/2;\ S \leftarrow 2\sqrt{S}/(1+S);$

}

`return` $A(1+S)/2;$

ALGORITHM FOR COMPUTING $T_k(m)$

$V \leftarrow 1;\ S \leftarrow \sqrt{m};$

`while` $(1 - S > 2^{-n})$ {

$W \leftarrow 2SV/(1 + V * V);\ W \leftarrow W/(1 + \sqrt{1 - W * W});$

$W \leftarrow (V + W)/(1 - V * W);\ V \leftarrow W/(1 + \sqrt{1 + W * W});$

$S \leftarrow 2\sqrt{S}/(1 + S);$

}

`return` $(1 + V)/(1 - V);$

Some values of $U(m)$ and $T(m)$ for $m \in (0, 1)$ are shown in Table 4.1 and Figure 4.1.

m	$U(m)$	$T(m)$	m	$U(m)$	$T(m)$
0.10	0.982438	2.670961	0.60	1.722836	5.600387
0.20	1.154937	3.173824	0.70	1.902090	6.699886
0.30	1.297205	3.659055	0.75	2.009459	7.459284
0.36	1.378276	3.968054	0.78	2.082689	8.026022
0.40	1.432174	4.187795	0.80	2.136394	8.468844
0.50	1.570796	4.810477	0.90	2.511507	12.323487

Table 4.1: The Functions $U(m)$ and $T(m)$

Brent showed that in $O(M(n)\log(n))$ operations, the above algorithms can compute $U_k(m)$ and $T_k(m)$ to precision. However, the precision in each operations are unclear.

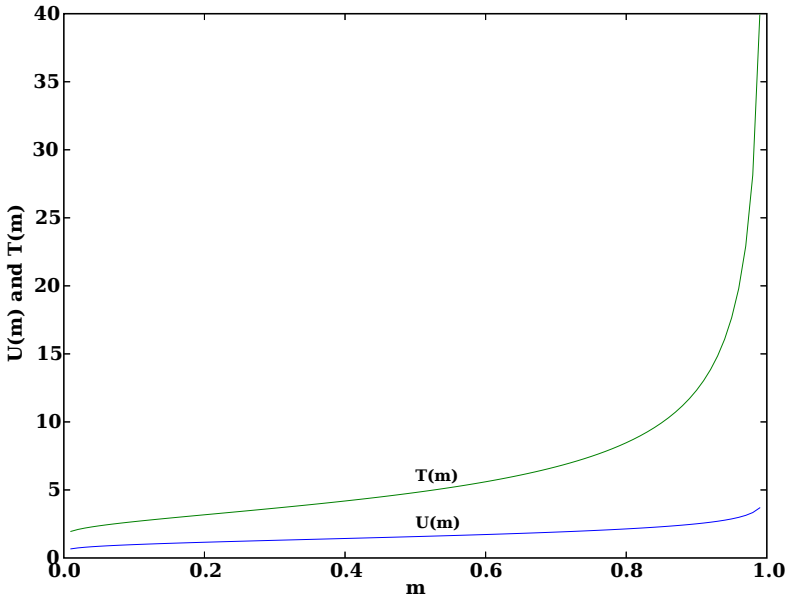

Figure 4.1: The Functions $U(m)$ and $T(m)$ for $m \in (0, 1)$

4.4.3 Convergence of $U_k(m)$ and $T_k(m)$

Suppose $\delta > 0$ fixed, and $m \in [\delta, 1-\delta]$. If $s_0 = \sin\alpha_0 = \sqrt{m}$, it follows from (4.19) that $s_i \to 1$ as $i \to \infty$. In fact, we have following lemma:

LEMMA 51. *Let* $\delta_i = 1 - s_i$. *If* $0 < \delta_i < 1/2$, *then*

$$\delta_{i+1} \leq \frac{3}{8}\delta_i^2, \tag{4.33}$$

i.e., s_i *converges to* 1 *with order* 2.

Proof. From (4.19), we have

$$\delta_{i+1} = 1 - s_{i+1} = 1 - \frac{\sqrt{1-\delta_i}}{1-\delta_i/2} \leq 1 - \frac{1 - \frac{\delta_i}{2} - \frac{\delta_i^2}{4}}{1 - \frac{\delta_i}{2}} \leq \frac{\delta_i^2}{4}(1+\delta_i) \leq \frac{3}{8}\delta_i^2$$

since $0 < \delta_i < 1/2$. So $s_i \to 1$ with order 2. **Q.E.D.**

Moreover, we can compute the number of iterations k needed to guarantee $1 - s_k \leq 2^{-n}$ using the Lemma below:

LEMMA 52. *Let*

$$C_2(n) = \log_2\left(n + \log_2 \frac{8}{3}\right) - \log_2 \log_2\left(\frac{8}{3\delta_0}\right) \tag{4.34}$$

where $\delta_0 = 1 - s_0$*, then after* $k \geq C_2(n)$ *iteration* $1 - s_k \leq 2^{-n}$*. Moreover, if* $\delta_0 \leq 1$*, then we have* $k \geq \log_2(n+2)$*.*

Proof. From Lemma 51, we see that

$$1 - s_k = \delta_k \leq \frac{3}{8}\delta_{k-1}^2 \leq \frac{3}{8}\left(\frac{3}{8}\delta_{k-2}^2\right)^2 = \left(\frac{3}{8}\right)^3 \delta_{k-2}^4 \leq \cdots \leq \left(\frac{3}{8}\right)^{2^k-1} \delta_0^{2^k}.$$

It suffices to let

$$\left(\frac{3}{8}\right)^{2^k-1} \delta_0^{2^k} \leq 2^{-n},$$

i.e.,

$$k \geq \log_2\left(n + \log_2 \frac{8}{3}\right) - \log_2 \log_2\left(\frac{8}{3\delta_0}\right).$$

Moreover, if $\delta_0 \leq 1$, then $\log_2 \frac{8}{3} \leq 2$ and $\log_2 \log_2\left(\frac{8}{3\delta_0}\right) > 0$, we can choose

$$k \geq \log_2(n+2).$$

Q.E.D.

4.4.4 Approximation of $U_k(m)$ and $T_k(m)$

Since $U(m)$ and $T(m)$ are both limits which we cannot compute directly, we will compute $U_k(m)$ and $T_k(m)$ instead for some k (we will describe how to compute k in next section). In this section, we present the algorithms for approximating $U_k(m)$ and $T_k(m)$ to n bits of absolute precision for fixed k.

LEMMA 53. *If $\widetilde{s_i} = s_i(1+\epsilon_i)$ and $|\epsilon_i| \le 2^{-n}$, then to guarantee relative n bits in s_{i+1}, it suffices to perform division with relative precision $n+4$ and square root extraction with relative precision $n+4$ bits when computing s_{i+1} using (4.19).*

Proof. From (4.19), we have

$$\begin{aligned}\widetilde{s_{i+1}} &= \widetilde{\left(\frac{2\widetilde{\sqrt{\widetilde{s_i}}}}{1+\widetilde{s_i}}\right)} \\ &= \frac{2\sqrt{s_i}\sqrt{1+\epsilon_i}(1+\rho_i)}{1+s_i(1+\epsilon_i)}(1+\rho_i')\end{aligned}$$

If $0 < \epsilon_i \le 1/2$, we have

$$\widetilde{s_{i+1}} \le \frac{2\sqrt{s_i}\sqrt{1+\epsilon_i}(1+\rho_i)}{1+s_i}(1+\rho_i') \le s_{i+1}(1+\epsilon_i/2)(1+\rho_i)(1+\rho_i').$$

If $-1/2 < \epsilon_i \le 0$, we have

$$\widetilde{s_{i+1}} \le \frac{2\sqrt{s_i}\sqrt{1+\epsilon_i}(1+\rho_i)}{(1+s_i)(1+\epsilon_i)}(1+\rho_i') \le s_{i+1}(1-3\epsilon_i/4)(1+\rho_i)(1+\rho_i').$$

i.e.,

$$\widetilde{s_{i+1}} \le s_{i+1}(1+3|\epsilon_i|/4)(1+\rho_i)(1+\rho_i').$$

If $|\rho_i| \le 2^{-(n+4)}$ and $|\rho_i'| \le 2^{-(n+4)}$, then

$$(1+3|\epsilon_i|/4)(1+\rho_i)(1+\rho_i') \le (1+3\cdot 2^{-n}/4)(1+2^{-(n+4)})^2 \le 1+2^{-n},$$

i.e.,

$$\widetilde{s_{i+1}} \le s_{i+1}(1+2^{-n}).$$

Q.E.D.

The above lemma shows that if we can perform division and square root extraction with relative $n+4$ bits, then we can compute s_i with relative n bits for $i = 0, 1, 2, \ldots$ if we approximate s_0 to n relative bits.

LEMMA 54. *We can approximate $U_k(m)$ to absolute n bits if we compute $s_0 = \sqrt{m}$ to relative $n + \lceil \log_2 F(\alpha_0) \rceil + \lceil \log_2 k \rceil + 2$ bits and approximate $F(\alpha_0)$ to relative $n + \lceil \log_2 F(\alpha_0) \rceil + 2$ bits. If $b_0 = \sqrt{1-m} \geq \frac{1}{4}$, then we just need compute s_0 to relative $n + \lceil \log_2 k \rceil + 5$ bits and approximate $F(\alpha_0)$ to relative $n+5$ bits.*

Proof. Note that

$$\begin{aligned}
\widetilde{U_k(m)} &= \widetilde{F(\alpha_0)} \prod_{i=0}^{k-1} \frac{1+\widetilde{s_i}}{2} \\
&\leq F(\alpha_0)(1+\rho) \prod_{i=0}^{k-1} \frac{1+s_i}{2} \prod_{i=0}^{k-1} (1+|\epsilon_i|) \\
&= U_k(m)(1+\rho) \prod_{i=0}^{k-1} (1+|\epsilon_i|)
\end{aligned}$$

Let $\epsilon = 2^{-(n+\log_2 F(\alpha_0)+\lceil \log_2 k \rceil + 2)}$, from Lemma 53, if $0 < |\epsilon_0| \leq \epsilon \leq \frac{1}{2}$, then $|\epsilon_i| \leq \epsilon$, we see that

$$\prod_{i=0}^{k-1} (1+|\epsilon_i|) \leq (1+\epsilon)^k \leq 1 + 2k \cdot \epsilon \leq 1 + \frac{1}{F(\alpha_0)} \cdot 2^{-(n+1)}.$$

Thus, if $|\rho| \leq 2^{-(n+\log_2 F(\alpha_0)+2)}$, then

$$\begin{aligned}
\left| \widetilde{U_k(m)} - U_k(m) \right| &\leq U_k(m) \frac{1}{F(\alpha_0)} \cdot 2^{-n} \\
&\leq 2^{-n}
\end{aligned}$$

Since $U_k(m) \leq F(\alpha_0)$. If $b_0 \geq \frac{1}{4}$, then

$$F(\alpha_0) = \frac{\pi}{2A} \leq \frac{\pi}{2b_0} \leq 8.$$

Hence, $\lceil \log_2 F(\alpha_0) \rceil \leq 3$. **Q.E.D.**

Now we assume $b_0 \geq \frac{1}{4}$. From (4.4), to obtain $n+5$ relative bits in $F(\alpha_0)$, we need compute π and A to $n+7$ relative bits and perform the division in relative

precision $n+7$. Since $A \geq b_0$ and $\pi > 2$, it requires to compute A to $n+5$ absolute bits and compute π to $n+8$ absolute bits using precision conversion. Both can be done using the algorithm we present in Section 4.3. Hence, we present the complete algorithm to compute $U_k(m)$ to absolute precision n:

ALGORITHM $U\_k()$ FOR COMPUTING $U_k(m)$ TO ABSOLUTE PRECISION n

Input: `BigFloat` m, integer k and precision n.

Output: `BigFloat` A such that $|A - U_k(m)| \leq 2^{-n}$.

$c \leftarrow C_1(n+6)$; $p \leftarrow n+6+c$; $pp \leftarrow p+1$;

$A \leftarrow 1$; $B \leftarrow sqrt(1-m, p)$;

`for (i=0; i<c; ++i) {`

    $Y \leftarrow A$; $A \leftarrow add(A, B, pp)$; $A \leftarrow div\_2exp(A, 1)$;

    $B \leftarrow mul(B, Y, pp)$; $B \leftarrow sqrt(B, pp)$;

    $pp \leftarrow pp - 1$;

}

$A \leftarrow div(Pi(n+8), A, n+7)$; $A \leftarrow div\_2exp(A, 1)$;

$pp \leftarrow n + 5 + ceillg(k)$;

$S \leftarrow sqrt(m, pp)$; $S_0 \leftarrow 0$;

`for (i=0; i<k-1; ++i) {`

    $S_0 \leftarrow add(1, S, pp+4)$; $S_0 \leftarrow div\_2exp(S_0, 1)$;

    $A \leftarrow mul(A, S_0, pp+4)$;

    $S_1 \leftarrow sqrt(S, pp+4)$; $S \leftarrow div(S_1, S_0, pp+4)$;

}

$A \leftarrow mul(A, S_0, pp+4)$;

`return` A;

For approximating $T_k(m)$ to n absolute bits, we need the following lemmas:

LEMMA 55. *If we have n bits relative precision in s_i and v_i and we perform all operations within $(n+1)$ bits relative precision in (4.23), then W_i'' has $n-3$ relative bits precision.*

Proof. Let ϵ_{s_i}, ϵ_{s_i} be the relative errors in s_i and v_i and ρ_1, ρ_2, ρ_3 and ρ_4 be the relative errors in the corresponding operations. From (4.23), we have

$$\begin{aligned}\widetilde{W_i''} &= \frac{2s_iv_i(1+\epsilon_{s_i})(1+\epsilon_{v_i})(1+\rho_1)}{(1+v_i^2(1+\epsilon_{v_i})^2(1+\rho_2))(1+\rho_3)}(1+\rho_4)\\ &\leq \frac{2s_iv_i}{1+v_i^2}(1+\epsilon_{s_i})(1+\epsilon_{v_i})(1+\rho_1)(1+3|\epsilon_{v_i}|)(1+2|\rho_2|)(1+2|\rho_3|)(1+\rho_4)\end{aligned}$$

If $\epsilon = 2^{-n}$, then $|\epsilon_{s_i}| \leq \epsilon$, $|\epsilon_{v_i}| \leq \epsilon$, $\rho_1 \leq \frac{\epsilon}{2}$, $\rho_2 \leq \frac{\epsilon}{2}$, $\rho_3 \leq \frac{\epsilon}{2}$ and $\rho_4 \leq \frac{\epsilon}{2}$. Hence

$$|\widetilde{W_i''} - W_i''| \leq |W_i''| \cdot (8\epsilon) = |W_i''| \cdot 2^{-(n-3)}.$$

Q.E.D.

LEMMA 56. *If we have n bits relative precision in W_i'' and we perform all operations within $(n+4)$ bits relative precision in (4.22), then W_i' has $n-1$ relative bits precision.*

Proof. Let $\epsilon_{W_i''}$ be the relative error in W_i'' and ρ_1, ρ_2, ρ_3 and ρ_4 be the relative errors in the corresponding operations. From (4.22), we have

$$\begin{aligned}\widetilde{W_i'} &= \frac{W_i''(1+\epsilon_{W_i''})}{(1+\sqrt{(1-W_i''^2(1+\epsilon_{W_i''})^2)(1+\rho_1)}(1+\rho_2))(1+\rho_3)}(1+\rho_4)\\ &\leq \frac{W_i''}{1+\sqrt{1-W_i''^2}}(1+\epsilon_{W_i''})(1-|\epsilon_{W_i''}|/2)(1+2|\rho_1|)(1+2|\rho_2|)(1+2|\rho_3|)(1+\rho_4)\end{aligned}$$

If $\epsilon = 2^{-n}$, then $|\epsilon_{W_i''}| \leq \epsilon$, $\rho_1 \leq \frac{\epsilon}{16}$, $\rho_2 \leq \frac{\epsilon}{16}$, $\rho_3 \leq \frac{\epsilon}{16}$ and $\rho_4 \leq \frac{\epsilon}{16}$. Hence

$$|\widetilde{W_i'} - W_i'| \leq |W_i'| \cdot (2\epsilon) = |W_i'| \cdot 2^{-(n-1)}.$$

Q.E.D.

LEMMA 57. *Let* $d = \left\lceil \log_2 \frac{1}{1-v_0 W_0'} \right\rceil$. *If we have* n *bits relative precision in* W_i' *and* v_i *and we perform all operations within* $(n+1)$ *bits relative precision in (4.21), then* W_i *has* $n-2-d$ *relative bits precision*

Proof. Let $\epsilon_{W_i'}$ be the relative error in W_i', ϵ_{v_i} be the relative error in v_i and ρ_1, ρ_2, ρ_3 and ρ_4 be the relative errors in the corresponding operations. From (4.21), we have

$$\begin{aligned}
\widetilde{W_i} &= \frac{(v_i(1+\epsilon_{v_i}) + W_i'(1+\epsilon_{W_i'}))(1+\rho_1)}{(1 - v_i W_i'(1+\epsilon_{v_i})(1+\epsilon_{W_i'})(1+\rho_2))(1+\rho_3)}(1+\rho_4) \\
&\leq \frac{v_i + W_i'}{1 - v_i W_i'}(1+|\epsilon_{v_i}|)(1+|\epsilon_{W_i'}|)(1+\rho_1)(1+2|\rho_3|)(1+\rho_4) \\
&\quad \cdot \left(1 + \frac{2}{1 - v_i W_i'} \left|(1+\epsilon_{v_i})(1+\epsilon_{W_i'})(1+\rho_2) - 1\right|\right)
\end{aligned}$$

If $\epsilon = 2^{-n}$, then $|\epsilon_{v_i}| \leq \epsilon$, $|\epsilon_{W_i'}| \leq \epsilon$, $\rho_1 \leq \frac{\epsilon}{2}$, $\rho_2 \leq \frac{\epsilon}{2}$, $\rho_3 \leq \frac{\epsilon}{2}$ and $\rho_4 \leq \frac{\epsilon}{2}$. Hence

$$|\widetilde{W_i} - W_i| \leq |W_i| \cdot \frac{4\epsilon}{1 - v_i W_i'} \leq |W_i| \cdot \frac{4\epsilon}{1 - v_0 W_0'} \leq |W_i| \cdot 2^{-(n-2-d)}.$$

Since v_i, W_i' are decreasing (§75). **Q.E.D.**

Since $v_0 = 1$, $s_0 = \sqrt{m}$, $W_0'' = \sqrt{m}$, $W_0' = \frac{\sqrt{m}}{1-\sqrt{1-m}}$. Thus,

$$\frac{1}{1 - v_0 W_0'} = \frac{1 - \sqrt{1-m}}{1 - \sqrt{1-m} - \sqrt{m}}.$$

We can compute d using IEEE double.

LEMMA 58. *If we have* n *bits relative precision in* W_i *and we perform all operations within* $(n+4)$ *bits relative precision in (4.20), then* v_{i+1} *has* $n-1$ *relative bits precision.*

Proof. Let ϵ_{W_i} be the relative error in W_i and ρ_1, ρ_2, ρ_3 and ρ_4 be the relative errors in the corresponding operations. From (4.20), we have

$$\begin{aligned}\widetilde{v_{i+1}} &= \frac{W_i(1+\epsilon_{W_i})}{(1+\sqrt{(1+{W_i}^2(1+\epsilon_{W_i})^2)(1+\rho_1)}(1+\rho_2))(1+\rho_3)}(1+\rho_4) \\ &\leq \frac{W_i}{1+\sqrt{1+{W_i}^2}}(1+\epsilon_{W_i})(1-|\epsilon_{W_i}|/2)(1+2|\rho_1|)(1+2|\rho_2|)(1+2|\rho_3|)(1+\rho_4)\end{aligned}$$

If $\epsilon = 2^{-n}$, then $|\epsilon_{W_i}| \leq \epsilon$, $\rho_1 \leq \frac{\epsilon}{16}$, $\rho_2 \leq \frac{\epsilon}{16}$, $\rho_3 \leq \frac{\epsilon}{16}$ and $\rho_4 \leq \frac{\epsilon}{16}$. Hence

$$|\widetilde{v_{i+1}} - v_{i+1}| \leq |v_{i+1}| \cdot (2\epsilon) = |v_{i+1}| \cdot 2^{-(n-1)}.$$

Q.E.D.

Therefore, we can see that if v_i, s_i has precision n, then after one iteration, v_{i+1} has only $(((n-3)-1)-2-d)-1 = n-7-d$ bits because of accumulation of errors. Now we describe the complete algorithm to compute $T_k(m)$ to absolute precision n:

ALGORITHM $T\_k()$ FOR COMPUTING $T_k(m)$ TO PRECISION n

Input: `BigFloat` m, integer k and precision n.

Output: `BigFloat` T such that $|T - T_k(m)| \leq 2^{-n}$.

$m_1 \leftarrow sqrt(m)$; $m_2 \leftarrow sqrt(1-m)$;

$d \leftarrow ilogb((1-m_2)/(1-m_2-m_1))$;

$p \leftarrow n + k*(7+d) + 2$; $V \leftarrow 1$; $S \leftarrow sqrt(m, p)$;

`for (i=0; i<k; ++i) {`

$pp \leftarrow p+1$; $W_0 \leftarrow mul(S, V, pp)$;

$W_0 \leftarrow mul\_2exp(W_0, 1)$; $W_1 \leftarrow mul(V, V, pp)$;

$W_1 \leftarrow add(1, W_1, pp)$; $W \leftarrow div(W_0, W_1, pp)$;

$pp \leftarrow p+1$; $W_1 \leftarrow mul(W, W, pp)$;

$W_1 \leftarrow sub(1, W_1, pp)$; $W_1 \leftarrow sqrt(W_1, pp)$;

$W_1 \leftarrow add(1, W_1, pp)$; $W \leftarrow div(W, W_1, pp)$;

$pp \leftarrow p-3$; $W_0 \leftarrow add(V, W, pp)$;

$W_1 \leftarrow mul(V, W, pp)$; $W_1 \leftarrow sub(1, W_1, pp)$;

$W \leftarrow div(W_0, W_1, pp)$; $pp \leftarrow p-2-d$;

$W_1 \leftarrow mul(W, W, pp)$; $W_1 \leftarrow add(1, W_1, pp)$;

$W_1 \leftarrow sqrt(W_1, pp)$; $W_1 \leftarrow add(1, W_1, pp)$;

$V \leftarrow div(W, W_1, pp)$;

$S_0 \leftarrow add(1, S, p+4)$; $S_0 \leftarrow div\_2exp(S_0, 1)$;

$S \leftarrow div(S_1, S_0, p+4)$; $p \leftarrow p-7-d$;

}

$T_1 \leftarrow add(1, V, p+2)$; $T_2 \leftarrow sub(1, V, p+2)$;

$T \leftarrow div(V_1, V_2, p+2)$;

`return` T;

4.4.5 Approximation of $U(m)$ and $T(m)$

LEMMA 59. *For $k \geq C_2(n + \lceil \log_2 F(\alpha_0) \rceil + 1)$, we have*

$$U_k(m) - U(m) \leq 2^{-n}.$$

Proof. Note that

$$\begin{aligned} |U_k(m) - U(m)| &= U_k(m) - U_k(m) \prod_{i=k}^{\infty} \frac{1+s_i}{2} \\ &= U_k(m) \left(1 - \prod_{i=k}^{\infty} \frac{1+s_i}{2}\right) \quad (s_i < 1) \\ &\leq F(\alpha_0) \left(1 - \prod_{i=k}^{\infty} \frac{1+s_i}{2}\right). \end{aligned}$$

Let $\delta_i = \frac{1-s_i}{2}$. If $0 < \delta_i \leq 1/2$, then

$$\prod_{i=k}^{\infty} \frac{1+s_i}{2} = \prod_{i=k}^{\infty} (1-\delta_i) \geq \prod_{i=k}^{\infty} e^{-\frac{4}{3}\delta_i} = e^{-\frac{4}{3}\sum_{i=k}^{\infty}\delta_i} \geq e^{-\frac{8}{3}\delta_k} \geq 1 - \frac{8}{3}\delta_k.$$

From Lemma 52, if $k \geq C_2(n + \lceil \log_2 F(\alpha_0) \rceil + 1)$ iterations, then

$$2\delta_k = (1 - s_k) \leq 2^{-(n+\log_2 F(\alpha_0)+1)}.$$

Thus,

$$U_k(m) - U(m) \leq F(\alpha_0)\frac{8}{3}\delta_k \leq 2F(\alpha_0)(1 - s_k) \leq 2^{-n}.$$

Q.E.D.

LEMMA 60. *To approximate $U(m)$ to n absolute bits, it suffices to evaluate $U_k(m)$ to $(n+1)$ absolute bits with $k \geq C_2(n + \lceil \log_2 F(\alpha_0) \rceil + 2)$.*

Proof. If $k \geq C_2(n + \lceil \log_2 F(\alpha_0) \rceil + 2)$, then from Lemma 59 we have

$$|U(m) - U_k(m)| \leq 2^{-(n+1)}.$$

Hence, if $\widetilde{U_k(m)}$ is $(n+1)$ bits absolute approximation of $U_k(m)$, we see that

$$\begin{aligned} |\widetilde{U_k(m)} - U(m)| &\leq |\widetilde{U_k(m)} - U_k(m)| + |U(m) - U_k(m)| \\ &\leq 2^{-(n+1)} + 2^{-(n+1)} \\ &= 2^{-n}. \end{aligned}$$

Q.E.D.

LEMMA 61. *To approximate $T(m)$ to n absolute bits, it suffices to evaluate $T_k(m)$ to $(n+1)$ absolute bits with $k = C_2(n + 2\lceil F(\alpha_0)\rceil + \lceil \log_2 F(\alpha_0)\rceil + 2)$.*

Proof. Note that

$$|T(m) - T_k(m)| \leq e^{\theta} |\log T(m) - \log T_k(m)|$$

where $\log T(m) \leq \theta \leq \log T_k(m)$. Since $\log T_k(m) \leq U_k(m) \leq F(\alpha_0)$,

$$\begin{aligned} |T(m) - T_k(m)| &\leq e^{F(\alpha_0)} (\log T_k(m) - \log T(m)) \\ &\leq e^{F(\alpha_0)} (U_k(m) - U(m)) . \end{aligned}$$

From Lemma 59, after $k = C_2(n + 2\lceil F(\alpha_0)\rceil + \lceil \log_2 F(\alpha_0)\rceil + 2)$ iterations,

$$|U(m) - U_k(m)| \leq 2^{-(n+2\lceil F(\alpha_0)\rceil+1)}.$$

Hence, if $\widetilde{T_k(m)}$ is $(n+1)$ bits absolute approximation of $T_k(m)$, we see that

$$\begin{aligned} |\widetilde{T_k(m)} - T(m)| &\leq |\widetilde{T_k(m)} - T_k(m)| + |T(m) - T_k(m)| \\ &\leq 2^{-(n+1)} + e^{F(\alpha_0)} \cdot 2^{-(n+2\lceil F(\alpha_0)\rceil+1)} \\ &\leq 2^{-n}. \end{aligned}$$

Q.E.D.

4.4.6 Discrete Newton Iteration

Our algorithms for evaluating exponential and logarithm functions also involve solving nonlinear equations, which we use discrete Newton iteration.

Consider the classical Newton iteration of the form:

$$x_{i+1} = x_i - \frac{f(x_i) - c}{f'(x_i)}. \tag{4.35}$$

for solving the equation $f(x) = c$. By Taylor's expansion, we have

$$0 = f(x) - c = f(x_i) - c + f'(x_i)(x - x_i) + f''(\zeta)(x - x_i)^2/2$$

where ζ is between x and x_i. Hence

$$x = x_i - \frac{f(x_i) - c}{f'(x_i)} - \frac{f''(\zeta)}{2f'(x_i)}(x - x_i)^2.$$

If $\epsilon_i = |x_i - x|$ is sufficiently small, then

$$|x_{i+1} - x| = \left|\frac{f''(\zeta)}{2f'(x_i)}\right| (x - x_i)^2 = O(\epsilon_i^2). \tag{4.36}$$

so the convergence of Newton iteration is quadratic. The classical Newton iteration assumed that (4.35) is satisfied exactly, but a results like (4.36) holds if we evaluated $f(x)$ with absolute error $O(\epsilon_i^2)$ and $f'(x)$ with absolute error $O(\epsilon_i)$ and perform the division with relative error $O(\epsilon_i)$ [9, 88].

When $f'(x)$ is not available, we can approximate $f'(x_i)$ using the one-sided difference

$$f'(x_i) \approx \frac{f(x_i + h_i) - f(x_i)}{h_i}.$$

To obtain $O(\epsilon_i)$ bits of absolute precision in $f'(x)$, it requires h_i is of order $O(\epsilon_i)$ and the evaluation of $f(x_i + h_i)$ and $f(x_i)$ are performed with an absolute error $O(\epsilon_i^2)$. This is called **discrete Newton Iteration**. It has the following form:

$$x_{i+1} = x_i + \frac{(f(x_i) - c)h_i}{f(x_i + h_i) - f(x_i)} \tag{4.37}$$

where we can choose $h = 2^{-i}$. A pseudo code is shown below:

ALGORITHM *Newton*() FOR DISCRETE NEWTON ITERATION:

Input: `BigFloat` x_0, callback function f and precision n.

Output: `BigFloat` x such that $|x - \zeta| \leq 2^{-n}$ where ζ is the root of f.

$x \leftarrow x_0$; $del \leftarrow 1$;

$f \leftarrow 0$; $f' \leftarrow 0$; $p \leftarrow 2$;

`do` {

$pp \leftarrow 2 * p$; $h \leftarrow div\_2exp(1, p)$;

$f \leftarrow f(x, pp)$; $f' \leftarrow f(x + h, pp)$; $f' \leftarrow sub(f', f, pp)$;

$f \leftarrow sub(f, c, pp)$; $f \leftarrow div\_2exp(f, p)$;

$del \leftarrow div(f, f', p)$; $x \leftarrow sub(x, del, pp)$;

$p \leftarrow pp$;

} `while` $(-del.uMSB() < n)$;

`return` x;

4.4.7 Evaluation $\exp(x)$ and $\log(x)$

We present our final algorithms for evaluating $\exp(x)$ and $\log(x)$:

§77. Evaluation of $\exp(x)$ To evaluate $\exp(x)$ to absolute precision n, we first reduce the argument to a suitable domain (see below), then we solve the equation:

$$U(m) = x, \tag{4.38}$$

obtaining m to absolute precision $n + 1 + \lceil \log M_{exp} \rceil$ using discrete Newton iteration where M_{exp} is the Lipschitz constant of $U(m)$ on this domain. Finally

we evaluate $T(m)$ to absolute precision $n+1$. From (4.32) and (4.38), $T(m) = \exp(x)$, so we have computed $\exp(x)$ to absolute precision n. The complete algorithm is as follows:

> ALGORITHM FOR COMPUTING $\exp(x)$ TO ABSOLUTE PRECION n:
>
> Input: `BigFloat` x, integer M_{exp} and precision n.
>
> Output: `BigFloat` y such that $|y - e^x| \leq 2^{-n}$.
>
> 1. find an initial value m_0 for solving $U(m) = x$ using Table 4.1.
> 2. solve $U(m) = x$ using discrete Newton iteration with the initial value m_0 and precision $n + 1 + M_{exp}$.
> 3. compute $T(m)$ with precision $n + 1$.

The correctness of this algorithm is shown by the following lemma:

LEMMA 62. *If $\infty < a < b < +\infty$, M_{exp} be the Lipschitz constants of $U(m)$ for the domain $[a, b]$, then the above algorithm can compute* $\exp(x)$ *to n absolute bits for $x \in [a, b]$.*

Proof. Let $\widetilde{T(\widetilde{m})}$ be the $(n+1)$ absolute bits approximation of $T(\widetilde{m})$, then

$$\begin{aligned} |\widetilde{T(\widetilde{m})} - e^x| &\leq |\widetilde{T(\widetilde{m})} - T(\widetilde{m})| + |T(\widetilde{m}) - e^x| \\ &\leq |\widetilde{T(\widetilde{m})} - T(\widetilde{m})| + M_{exp} \cdot |\widetilde{m} - m| \\ &\leq 2^{-(n+1)} + M_{exp} \cdot 2^{-(n+1+\lceil \log M_{exp} \rceil)} \\ &\leq 2^{-n}. \end{aligned}$$

Q.E.D.

§78. Evaluation of $\log(x)$ Similarly, to evaluate $\log(x)$ to absolute precision n, we first reduce the argument to some domain (see below), then we solve the

equation:

$$T(m) = x, \tag{4.39}$$

obtaining m to absolute precision $n + 1 + \lceil \log M_{log} \rceil$ using discrete Newton iteration where M_{log} is the Lipschitz constant of $T(m)$ on this domain Finally we evaluate $U(m)$ to absolute precision n. From (4.32) and (4.39), $U(m) = \log(x)$, we obtained $\log(x)$ within absolute precision n. The complete algorithm is as follows:

> Algorithm for Computing $\log(x)$ to absolute precion n:
>
> Input: `BigFloat` x, integer M_{log} and precision n.
>
> Output: `BigFloat` y such that $|y - \log x| \leq 2^{-n}$.
>
> 1. find an initial value m_0 for solving $T(m) = x$ using Table 4.1.
> 2. solve $T(m) = x$ using discrete Newton iteration with the initial value m_0 and precision $n + 1 + M_{log}$.
> 3. compute $U(m)$ with precision $n + 1$.

Lemma 63. *If $0 < a < b < +\infty$, M_{log} be the Lipschitz constants of $T(m)$ for the domain $[a, b]$, then the above algorithm can compute $\log(x)$ to n absolute bits for $x \in [a, b]$.*

Proof. Let $\widetilde{U(\widetilde{m})}$ be the $(n+1)$ absolute bits approximation of $U(\widetilde{m})$, then

$$\begin{aligned}
|\widetilde{U(\widetilde{m})} - \log(x)| &\leq |\widetilde{U(\widetilde{m})} - U(\widetilde{m})| + |U(\widetilde{m}) - \log(x)| \\
&\leq |\widetilde{U(\widetilde{m})} - U(\widetilde{m})| + M_{log} \cdot |\widetilde{m} - m| \\
&\leq 2^{-(n+1)} + M_{log} \cdot 2^{-(n+1+\lceil \log M_{log} \rceil)} \\
&\leq 2^{-n}.
\end{aligned}$$

Q.E.D.

§79. Lipschitz Constants of $U(m)$ and $T(m)$ Our algorithms require to find Lipschitz constants of $U(m)$ and $T(m)$. So we need to find a more explicit formula for $U(m)$ with respect to m.

Let $a_0 = 1$ and $b_0 = \sin\alpha_0 = \cos(\frac{\pi}{2} - \alpha_0)$ where α_0 is defined in §74. Then $s_0 = \sin\alpha_0 = b_0/a_0$. From (4.16) and the AGM iteration, it follows that $s_i = a_i/b_i$ for all $i \geq 0$. Thus, $(1 + s_i)/2 = a_{i+1}/a_i$, and

$$\prod_{i=0}^{\infty} \frac{1+s_i}{2} = \lim_{i\to\infty} a_i = A(\frac{\pi}{2} - \alpha_0) = \frac{\pi}{2F(\frac{\pi}{2} - \alpha_0)}.$$

Substituting it into (4.28), we obtain another formula for $U(m)$:

$$U(m) = \frac{\pi}{2} \cdot \frac{F(\alpha_0)}{F(\frac{\pi}{2} - \alpha_0)}.$$

From [8, page 9], we can derive the following two derivatives (with respect to m):

$$F'(\alpha_0) = \frac{E(\alpha_0) - (1-m)F(\alpha_0)}{2m(1-m)}$$

and

$$F'(\frac{\pi}{2} - \alpha_0) = -\frac{E(\frac{\pi}{2} - \alpha_0) - mF(\frac{\pi}{2} - \alpha_0)}{2m(1-m)}.$$

Hence,

$$\begin{aligned} U'(m) &= \frac{\pi}{2} \cdot \frac{F'(\alpha_0)F(\frac{\pi}{2} - \alpha_0) - F(\alpha_0)F'(\frac{\pi}{2} - \alpha_0)}{F^2(\frac{\pi}{2} - \alpha_0)} \\ &= \frac{\pi}{2} \cdot \frac{E(\alpha_0)F(\frac{\pi}{2} - \alpha_0) - F(\alpha_0)F(\frac{\pi}{2} - \alpha_0) + F(\alpha_0)E(\frac{\pi}{2} - \alpha_0)}{2m(1-m)F^2(\frac{\pi}{2} - \alpha_0)} \\ &= \frac{1}{2m(1-m)} \cdot \frac{(\frac{\pi}{2})^2}{F^2(\frac{\pi}{2} - \alpha_0)} \\ &= \frac{1}{2m(1-m)} A^2(\frac{\pi}{2} - \alpha_0). \end{aligned}$$

Here we use Legendre's Identity (4.7). From (4.2), we have $A(\frac{\pi}{2} - \alpha_0) \leq a_0 = 1$, therefore, we obtain the following upper bound function for $U'(m)$:

$$U'(m) \leq \frac{1}{2m(1-m)}.$$

From the identity (4.32), we have

$$U'(m) = \frac{T'(m)}{T(m)}.$$

Hence, we have the following upper bound function for $T'(m)$:

$$T'(m) = T(m)U'(m) \leq \frac{T(m)}{2m(1-m)}.$$

The graphs for these upper bound functions are shown in Figure 4.2.

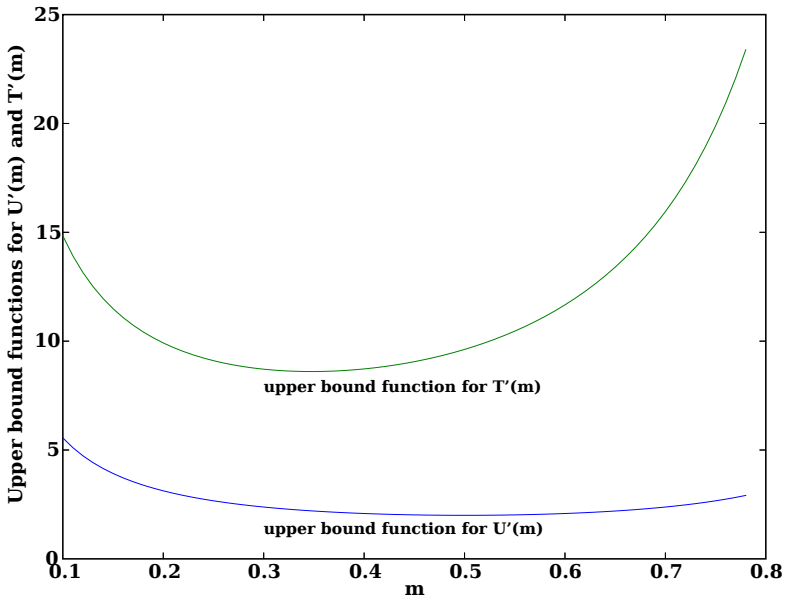

Figure 4.2: Upper Bound Functions of $U'(m)$ and $T'(m)$ for $m \in (0, 1)$

We can easily see that the function

$$G(m) := \frac{1}{2m(1-m)}$$

decreases for $m < 0.5$ and increases for $m > 0.5$. So $G(m)$ achieves its maximal value on the endpoints of the specified domain.

For the $\exp(x)$ evaluation algorithm, we can reduce the argument x to $[1, 2]$ (see §64 for detailed algorithm). From Table 4.1, we can see that solution m of (4.38) lies in $(0.10, 0.75)$. Hence,

$$\frac{1}{2m(1-m)} < G(0.10) < 6.$$

We can choose the Lipschitz constant M_{exp} of $U(m)$ to be 6.

For the $\log(x)$ evaluation algorithm, we reduce x to $[4, 8]$ (using a similar algorithm in §63). We can see from Table 4.1 that the solution m of (4.39) lies in $(0.36, 0.78)$. Hence

$$\frac{1}{2m(1-m)} < G(0.78) < 3.$$

Thus,

$$T'(m) \leq 3 \cdot T(m) < 3 \cdot 8.03 < 25,$$

i.e, we can choose $M_{log} = 25$.

For any continuous function $f(x)$, we can also compute an approximation $\widetilde{f'(x)}$ of $f'(x)$ using $f'(x) \approx \frac{f(x+h)-f(x)}{h}$. We compare the approximations of $U'(m)$ and $T'(m)$ with the computed values of their upper bound functions in Table 4.2.

m	$\widetilde{U'(m)}$	$G(m)$	$\widetilde{T'(m)}$	$T(m)G(m)$	m	$\widetilde{U'(m)}$	$G(m)$	$\widetilde{T'(m)}$	$T(m)G(m)$
0.10	2.057144	5.555556	5.500076	14.838675	0.60	1.628218	2.083333	9.125906	11.667473
0.20	1.512100	3.125000	4.802686	9.918201	0.70	2.002509	2.380952	13.429708	15.952109
0.30	1.363641	2.380952	4.992963	8.712036	0.75	2.319130	2.666667	17.318653	19.948216
0.36	1.344962	2.170139	5.340387	8.611229	0.78	2.319130	2.913753	17.318653	23.385845
0.40	1.352635	2.083333	5.668301	8.724573	0.80	2.805413	3.248863	23.791176	26.465136
0.50	1.436182	2.000000	6.913569	9.620955	0.90	5.296746	5.555556	65.443490	68.463815

Table 4.2: Approximations and Upper Bounds of $U'(m)$ and $T'(m)$

4.5 Summary

We presented a non-asymptotic error analysis of AGM algorithms in this chapter to further investigate the evaluation problems of transcendental functions. Our analysis gave the explicit precision required in each step of AGM based evaluation algorithms. It will help users implement and verify their applications. Moreover, this analysis make it possible for us to incorporate elementary functions into the `Core Library` using AGM algorithms. An implementation of computing π, exponential and logarithm functions have been incorporated into the `Core Library`.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

This dissertation investigates the first effort at bringing transcendental functions into Exact Geometric Computation (EGC).

The key technique in EGC computation is the use of "constructive root bounds". It makes it possible to compute the exact sign of expressions. However, existing constructive root bounds apply only available to algebraic expressions. In this thesis, we incorporate transcendental functions into expressions by introducing an "escape bound" mechanism in lieu of root bounds. This is not going to be the ultimate solution but it makes it possible for EGC to handle transcendental applications.

Approximation algorithms for transcendental functions are essential for our implementation. We present a complete algorithm for absolute approximation of the general hypergeometric function and various AGM based algorithms for the elementary functions. Although the hypergeometric functions include the elementary functions, the performance of the general algorithm is inferior to the

AGM-based algorithms.

General EGC software libraries provide a basis for robust geometric computing. While theoretically the performance of EGC is constrained by root bounds in the worst case, the design and implementation of a EGC library can impact the complexity and the performance of the whole system. We review the design and performance issues existing in our current `Core Library`. A new design that aims to increase the modularity, extensibility and efficiency is presented. The structure of the redesigned library is much cleaner and the code is easier to maintain and debug. The most important improvement is the performance. Our benchmark shows that our system can obtain 5-10 times speedup.

For improving usability of the `Core Library`, a new mechanism is provided to help users develop their own operations. Several new operations: `sum()`, `product()`, `pi()` are presented to demonstrate the use of this new mechanism. Such operations can lead to greater efficiency for specific applications.

In conclusion, our redesigned `Core Library` is more efficient and easier for both transcendental and algebraic computation. We have reasons to be optimistic that the EGC approach will become more and more effective for many problems.

5.2 Future Work

Root Bounds for Transcendental Functions While the zero problem for transcendental functions is still an open problem, even partial results for this problem would be useful. For example, in [69] several lower bounds are provided for non-algebraic expressions

$$|e^{\beta} - \alpha| \quad \text{and} \quad |\beta - \log(\alpha)|$$

where α, β are algebraic numbers. How to extend these results to arbitrary non-algebraic expressions such as $\Omega_3 \cup \{\exp(\cdot), \log(\cdot)\}$, $\Omega_3 \cup \{\exp(\cdot), \pi\}$ and compute them efficiently will be a big challenge.

On Development of the `Core Library` Further effort is needed to improve the efficiency of our `Core Library` at both the algorithmic level and the system level.

At the algorithmic level, we need to continue to improve our key algorithms and develop new algorithms for root bound, filter and bigfloat computation. For instance, our expression evaluation algorithm propagates precision to the operands symmetrically for binary operations. The optimal allocation of bits is an interesting problem for future work.

In the current `Core Library`, only constants are allowed at the leaves of the expression DAGs. An interesting topic is to allow leaf nodes in expression DAG to be variables. This could speed up the whole computation for certain applications since we can avoid re-construction of the DAG.

At the system level, we currently implemented our exact ring operations for the `BigFloat` class by estimating the precision for results first and then calling `MPFR` functions. A more efficient solution would be to implement those operations directly using `mpn` functions in `GMP`.

We plan to support more operations in the `Core Library`, for example, the diamond operator. Another important future work is to apply our EGC approach to more areas of computation where guaranteed sign computation is critical.

On Hypergeometric Functions Our approximation algorithm gives an upper bound on the number of terms needed to guarantee a given absolute precision. Such a general bound could be over-estimated. Further work should

compare the performance of this algorithm with the progressive evaluation algorithm.

Many hypergeometric series have poor convergence properties. The rate of convergence of these series can be accelerated using some techniques. For example, Gosper's algorithm [37] and WZ method [2] proved to be very efficient in series acceleration. Adding automatic transformations for series acceleration into our hypergeometric package will further improve its performance.

On AGM algorithm While we analyzed and implemented the evaluation algorithms for exponential and logarithm function in this thesis, similar work for AGM-based algorithms of other elementary functions such as trigonometric functions remains to be done.

In §79, we derive an explicit form of $U'(m)$ (and $T'(m)$). So we can use classical Newton iteration instead of discrete Newton iteration for solving (4.38) and (4.39). Future work should compare these two methods.

AGM iteration is simple and efficient and it connects with elliptic integrals, hypergeometric functions. We should further explore how we develop other AGM-based algorithms.

Appendix A

BFS Filter

In the section we use `BfsFilter` as an example to show how users can write their own `Filter` class. The `Bfsfilter` is based on BFS filter in [17].

To design a `Filter` class, a user needs to do the following steps:

1. Define internal data fields. BFS filter maintains 3 parameters: `fpVal`, `maxAbs` and `ind`.

```
1 class BfsFilter {
2   double fpVal;
3   double maxAbs;
4   int ind;
5   ...
6 };
```

2. Implement checking function `is_ok()`. Given an expression E, the criteria for certifying the sign of E for BFS filter is

$$|\texttt{fpVal(E)}| > \texttt{maxAbs(E)} \cdot \texttt{ind(E)} \cdot 2^{-p}$$

where p is 52 for the standard C/C++ data type `double`. This can be translated into the following code in a straightforward way:

```
bool is_ok() const {
 return (get_filter_flag()&&//check if filter is enabled
         finite(fpVal)&&      //test for infinite and NaNs
         (fabs(fpVal)>maxAbs*ind*CORE_EPS));
}
```

Here, `CORE_EPS` is pre-defined to be 2^{-52}. Note that we need to call the `get_filter_flag()` function.

3. Implement `sgn()`, `uMSB()` and `lMSB()`. When the filter is certified, the sign of E is as same as `fpVal(E)`:

```
int sign() const
{ return (fpVal == 0.0) ? 0 : (fpVal > 0.0 ? 1: -1); }
```

and the upper and lower bound of $\lg|E|$ can be computed using the formula below:

$$E^{+} = \left\lceil \log_2 \left(|\texttt{fpVal(E)}| + \texttt{maxAbs(E)} \cdot \texttt{ind(E)} \cdot 2^{-p} \right) \right\rceil$$

and

$$E^{-} = \left\lfloor \log_2 \left(|\texttt{fpVal(E)}| - \texttt{maxAbs(E)} \cdot \texttt{ind(E)} \cdot 2^{-p} \right) \right\rfloor$$

```
long uMSB() const
{ return long(ilogb(fabs(fpVal)+maxAbs*ind*CORE_EPS))+1; }
long lMSB() const
{ return long(ilogb(fabs(fpVal)-maxAbs*ind*CORE_EPS)); }
```

4. Implement assignment function and arithmetic functions. The recursive computation rules for BFS filter are given in Table A.1, where $\oplus, \odot, \oslash$ are used to represent the corresponding floating-point arithmetic operations and $\tilde{E}$ is used to represent the `fpVal(E)`. $sqrt(), cbrt(), root()$ are the functions which can computing the corresponding approximated results

in relative 52 bits precision ($sqrt()$ is standard ANSI C function and we implemented $cbrt()$ and $root()$ using `MPFR`).

Expr E	`maxAbs(E)`	`ind(E)`
exact E	$\|E\|$	0
approx E	$\|round(E)\|$	1
$E = F \pm G$	$\texttt{maxAbs(F)} \oplus \texttt{maxAbs(G)}$	$1 + \max\{\texttt{ind(F)}, \texttt{ind(G)}\}$
$E = F \times G$	$\texttt{maxAbs(F)} \odot \texttt{maxAbs(G)}$	$1 + \texttt{ind(F)} + \texttt{ind(G)}$
$E = F/G$	$\frac{(\|\tilde{F}\| \oslash \|\tilde{G}\|) \oplus (\texttt{maxAbs(F)} \oslash \texttt{maxAbs(G)})}{(\|\tilde{G}\| \oslash \texttt{maxAbs(G)}) \ominus (\texttt{ind(G)}+1) \cdot 2^{-p}}$	$1 + \max\{\texttt{ind(F)}, \texttt{ind(G)} + 1\}$
$E = \sqrt[2]{F}$	$\begin{cases} (\texttt{maxAbs(F)} \oslash \tilde{F}) \odot sqrt(\tilde{F}) & \text{if } \tilde{F} > 0 \\ sqrt(\texttt{maxAbs(F)}) \odot 2^{p/2} & \text{if } \tilde{F} = 0 \end{cases}$	$1 + \texttt{ind(F)}$
$E = \sqrt[3]{F}$	$\begin{cases} (\texttt{maxAbs(F)} \oslash \tilde{F}) \odot cbrt(\tilde{F}) & \text{if } \tilde{F} > 0 \\ cbrt(\texttt{maxAbs(F)}) \odot 2^{p/3} & \text{if } \tilde{F} = 0 \end{cases}$	$1 + \texttt{ind(F)}$
$E = \sqrt[k]{F}$	$\begin{cases} (\texttt{maxAbs(F)} \oslash \tilde{F}) \odot root(\tilde{F}, k) & \text{if } \tilde{F} > 0 \\ root(\texttt{maxAbs(F)}, k) \odot 2^{p/k} & \text{if } \tilde{F} = 0 \end{cases}$	$1 + \texttt{ind(F)}$

Table A.1: Rules for BFS filter, p=52.

For implementing `set()`, we can apply the rules in the first and second row of above table. For example, we can implement it as follow when the input parameter `value` is `long` type:

```
1 void set(long value) {
2   fpVal = value; maxAbs = value > 0 ? value : (-value);
3   ind = (sizeof(long) > 4 && ceillg(value) >= 53) ? 1 : 0;
4 }
```

Here we set `ind(E)` to be 1 if the input integer has more than 53 bits since it cannot be convert to `double` exactly. Otherwise we set it to be 0. The arithmetic functions `add()` and `sub()` can be implemented using the rule in the third row:

```
1 void add(const thisClass& f, const thisClass& s) {
2     fpVal = f.fpVal + s.fpVal;
3     maxAbs = f.maxAbs + s.maxAbs;
4     ind = 1 + (f.ind > s.ind ? f.ind : s.ind);
```

```
5  }
6  void sub(const thisClass& f, const thisClass& s) {
7      fpVal = f.fpVal - s.fpVal;
8      maxAbs = f.maxAbs + s.maxAbs;
9      ind = 1 + (f.ind > s.ind ? f.ind : s.ind);
10 }
```

Similarly, one can implement functions `mul()`, `div()`, `sqrt()`, `cbrt()`, `root()` using the rules in rows 4-6.

Remarks:

(1) We did not use IEEE double interval as our floating-point filter [12] since the mechanism for setting IEEE rounding modes is not very portable and switching between different rounding modes is costly.

(2) When underflow in those machine floating-point arithmetic operations happens, we can ignore it for $\oplus$, $\ominus$ and $\sqrt[k]{\ }$, and add a small constant `MIN_DBL` $= 2.2250738585072014e^{-308}$ to `maxAbs(E)` for $\odot$ and $\oslash$. For overflow, we can detect it using the function `finite()` from `<cmath>` (see the implementation of `is_ok()`).

Appendix B

BFMSS Root Bound

Here we give an example of how we write a `Rootbd` class from a set of constructive root bound rules. We use BFMSS bound [15, 16] as an example. Conceptually, BFMSS bound first transforms a radical expression E to a quotient of two division-free expression $U(E)$ and $L(E)$. But we do not have to compute $U(E)$ or $L(E)$. Instead we maintain two parameters $u(E)$ and $l(E)$, the upper bounds on the conjugates of $U(E)$ and $L(E)$, respectively by the recursive rules in Table B.1:

Expr E	$u(E)$	$l(E)$
Rational $\frac{a}{b}$	$\|a\|$	$\|b\|$
$E = F \pm G$	$u(F)l(G) + l(F)u(G)$	$l(F)l(G)$
$E = F \times G$	$u(F)u(G)$	$l(F)l(G)$
$E = F \div G$	$u(F)l(G)$	$l(F)u(G)$
$E = \sqrt[k]{F}$	$\min\{\sqrt[k]{u(F)l(F)^{k-1}}, u(F)\}$	$\min\{\sqrt[k]{u(F)^{k-1}l(F)}, l(F)\}$

Table B.1: Rules for BFMSS bound.

The BFMSS root bound function is

$$B(E) = \frac{1}{u(E)^{d(E)-1}l(E)}$$

where $d(E)$ is the degree bound that we mentioned in §19.

Now we are going to design our `BfmssRootbd` class:

1. Define internal data fields. Bfmss Rootbd maintains 3 data fields: `u_e`, `l_e` and `d_e` which store upper bounds on $\lg u(E)$, $\lg l(E)$ and $d(E)$ respectively:

```
1 class BfmssRootbd {
2   unsigned long u_e;
3   unsigned long l_e;
4   unsigned long d_e;
5   ...
6 };
```

2. Implement `get_bound()`. We just return $-\lg(E)$, i.e.,

$$\lg u(E) \cdot (d(E) - 1) + \lg l(E) = \texttt{u\_e} * (\texttt{d\_e} - 1) + \texttt{l\_e}.$$

```
1 unsigned long get_bound() const
2 { return u_e*(d_e-1)+l_e; }
```

3. Implement `is_degree_based()`, `set_degree_bound()` and `get_degree_bound()`.

```
1 static bool is_degree_based()
2 { return true; }
3 void set_degree_bound(unsigned long d)
4 { d_e = d; }
5 unsigned long get_degree_bound() const
6 { return d_e; }
```

4. Implement assignment functions `set()` for different types:

```
1 void set(long value)
2 { u_e = ceillg(value); l_e = 0; }
3 void set(unsigned long value)
4 { u_e = ceillg(value); l_e = 0; }
```

```
5  void set(double value)
6  { set(BigFloat(value)); }
7  void set(const BigInt& value)
8  { u_e = value.ceillg(); l_e = 0; }
9  void set(const BigRat& value)
10 { u_e = value.num().ceillg(); l_e = value.den().ceillg(); }
11 void set(const BigFloat& value) {
12   BigInt x; exp_t e = value.get_z_exp(x);
13   if (e >= 0) { // convert to integer
14     x.mul_2exp(x, e); set(x);
15   } else { // convert to rational
16     BigRat q; q.div_2exp(x, -e); set(q);
17   }
18 }
```

5. Implement arithmetic functions. Since we now maintain `u_e` and `l_e`, we derive logarithm form rules from above Table B.1:

$E = F \pm G$	$\max\{F.\texttt{u\_e} + G.\texttt{l\_e}, F.\texttt{l\_e} + G.\texttt{u\_e}\} + 1$	$F.\texttt{l\_e} + G.\texttt{l\_e}$
$E = F \times G$	$F.\texttt{u\_e} + G.\texttt{u\_e}$	$F.\texttt{l\_e} + G.\texttt{l\_e}$
$E = F \div G$	$F.\texttt{u\_e} + G.\texttt{l\_e}$	$F.\texttt{l\_e} + G.\texttt{u\_e}$
$E = \sqrt[k]{F}$	$\begin{cases} \frac{F.\texttt{u\_e}+(k-1)*F.\texttt{l\_e}}{k} & \text{if } F.\texttt{u\_e} \geq F.\texttt{l\_e} \\ F.\texttt{u\_e} & \text{if } F.\texttt{u\_e} < F.\texttt{l\_e} \end{cases}$	$\begin{cases} F.\texttt{l\_e} & \text{if } F.\texttt{u\_e} \geq F.\texttt{l\_e} \\ \frac{(k-1)*F.\texttt{u\_e}+F.\texttt{l\_e}}{k} & \text{if } F.\texttt{u\_e} < F.\texttt{l\_e} \end{cases}$

Table B.2: Rules for BFMSS bound in logarithm form.

```
1  void neg(const thisClass& child)
2  { u_e = child.u_e; l_e = child.l_e; }
3  void root(const thisClass& child, unsigned long k) {
4    if (child.u_e >= child.l_e) {
5      u_e = (child.u_e + (k-1)*child.l_e + (k-1)) / k;
6      l_e = child.l_e;
7    } else {
8      u_e = child.u_e;
9      l_e = ((k-1)*child.u_e + child.l_e + (k-1)) / k;
10   }
11 }
```

```
12 void add(const thisClass& f, const thisClass& s) {
13   u_e = std::max(f.u_e + s.l_e, f.l_e + s.u_e) + 1;
14   l_e = f.l_e + s.l_e;
15 }
16 void sub(const thisClass& f, const thisClass& s) {
17   u_e = std::max(f.u_e + s.l_e, f.l_e + s.u_e) + 1;
18   l_e = f.l_e + s.l_e;
19 }
20 void mul(const thisClass& f, const thisClass& s)
21 { u_e = f.u_e + s.u_e; l_e = f.l_e + s.l_e; }
22 void div(const thisClass& f, const thisClass& s)
23 { u_e = f.u_e + s.l_e; l_e = f.l_e + s.u_e; }
```

It is clear that once we have root bound functions and recursive rules, writing a `Rootbd` is straightforward. We give this information for other two root bound algorithms that we provided in the `Core Library`:

Degree-Measure bound: The root bound function is

$$B(E) = \frac{1}{m(E)}.$$

The recursive rules are shown in Table B.3:

Expr E	$d(E)$	$m(E)$
Rational $\frac{a}{b}$	1	$\max\{\|a\|, \|b\|\}$
$E = F \pm G$	$d(F)d(G)$	$m(F)^{d(G)}m(G)^{d(F)}2^{d(E)}$
$E = F \times G$	$d(F)d(G)$	$m(F)^{d(G)}m(G)^{d(F)}$
$E = F \div G$	$d(F)d(G)$	$m(F)^{d(G)}m(G)^{d(F)}$
$E = \sqrt[k]{F}$	$kd(F)$	$m(F)$

Table B.3: Rules for Degree-Measure bound.

Li-Yap bound: The root bound function for Li-Yap bound [54] is

$$B(E) = \frac{1}{\mu(E)^{d(E)-1}lc(E)}.$$

Expr E	$lc(E)$	$tc(E)$	$\mu(E)$	$\nu(E)$
Rational $\frac{a}{b}$	$\|b\|$	$\|a\|$	$\|\frac{a}{b}\|$	$\|\frac{a}{b}\|$
$E = F \pm G$	$lc(F)^{d(G)} lc(G)^{d(F)}$	$m(F)^{d(G)} m(G)^{d(F)} 2^{d(E)}$	$\mu(F) + \mu(G)$	$(*)$
$E = F \times G$	$lc(F)^{d(G)} lc(G)^{d(F)}$	$tc(F)^{d(G)} tc(G)^{d(F)}$	$\mu(F)\mu(G)$	$\nu(F)\nu(G)$
$E = F \div G$	$lc(F)^{d(G)} tc(G)^{d(F)}$	$tc(F)^{d(G)} lc(G)^{d(F)}$	$\mu(F)/\nu(G)$	$\nu(F)/\mu(G)$
$E = \sqrt[k]{F}$	$lc(F)$	$tc(F)$	$\sqrt[k]{\mu(F)}$	$\sqrt[k]{\nu(F)}$

Table B.4: Rules for Li-Yap bound.

The recursive rules are shown in Table B.4:

The last entry in Line 3 is missing: this special entry is

$$\max\{M(E)^{-1}, (\mu(E)^{d(E)-1} lc(E))^{-1}\}.$$

Bibliography

[1] G. Alefeld and J. Herzberger. *Introduction to Interval Computations*. Academic Press, New York, 1983. Translated from German by Jon Rokne.

[2] T. Amdeberhan and D. Zeilberger. Hypergeometric series acceleration via the WZ method, 1996.

[3] W. N. Bailey. *Generalized Hypergeometric Series*. Cambridge University Press, 1935.

[4] Bernard Chazelle, et al. The computational geometry impact task force report: an executive summary. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, pages 59–66. Springer, 1996. Lecture Notes in Computer Science No. 1148.

[5] Bernard Chazelle, et al. Application challenges to computational geometry: CG impact task force report. Tr-521-96, Princeton Univ., April, 1996. See also URL www.cs.princeton.edu/~chazelle/.

[6] J.-D. Boissonnat and M. Yvinec. *Algorithmic Geometry*. Cambridge University Press, 1997. Translated by Hervé Brönnimann.

[7] D. Booth. Integrating Python, C and C++. `http://www.suttoncourtenay.org.uk/duncan/accu/integratingpython.html`. Python Integration.

[8] J. M. Borwein and P. B. Borwein. *Pi and the AGM*. John Wiley and Sons, New York, 1987.

[9] R. P. Brent. The complexity of multiple-precision arithmetic. In R. S. Anderssen and R. P. Brent, editors, *The Complexity of Computational Problem Solving*, pages 126–165. University of Queensland Press, Brisbane, 1976. Retyped and postscript added 1999.

[10] R. P. Brent. Fast multiple-precision evaluation of elementary functions. *J. of the ACM*, 23:242–251, 1976.

[11] R. P. Brent. Multiple-precision zero-finding methods and the complexity of elementary function evaluation. In J. F. Traub, editor, *Proc. Symp. on Analytic Computational Complexity*, pages 151–176. Academic Press, 1976.

[12] H. Brönnimann, C. Burnikel, and S. Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. *Discrete Applied Mathematics*, 109(1-2):25–47, 2001.

[13] C. Burnikel. *Exact Computation of Voronoi Diagrams and Line Segment Intersections*. Ph.D thesis, Universität des Saarlandes, Mar. 1996.

[14] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. Exact efficient geometric computation made easy. In *Proc. 15th ACM Symp. Comp. Geom.*, pages 341–450, New York, 1999. ACM Press.

[15] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. A strong and easily computable separation bound for arithmetic expressions involving radicals. *Algorithmica*, 27:87–99, 2000.

[16] C. Burnikel, S. Funke, K. Mehlhorn, S. Schirra, and S. Schmitt. A separation bound for real algebraic expressions. In *Lecture Notes in Computer Science*, pages 254–265. Springer, 2001. to appear, Algorithmica.

[17] C. Burnikel, S. Funke, and M. Seel. Exact geometric predicates using cascaded computation. In *Symposium on Computational Geometry*, pages 175–183, 1998.

[18] C. Burnikel, S. Funke, and M. Seel. Exact geometric computation using cascading. *Int'l. J. Comput. Geometry and Appl.*, 11(3):245–266, 2001. Special Issue.

[19] CGAL Homepage, 1998. URL `http://www.cgal.org/`. Computational Geometry Algorithms Library (CGAL) Project. A 7-institution European Community effort.

[20] J. Choi, J. Sellen, and C. Yap. Approximate Euclidean shortest paths in 3-space. *Int'l. J. Comput. Geometry and Appl.*, 7(4):271–295, 1997. Also: 10th ACM Symp. on Comp. Geom. (1994), pp.41–48.

[21] Core Library homepage, since 1999. Software downloads, documentation and links: `http://cs.nyu.edu/exact/core/`.

[22] B. P. Demidovich and I. A. Maron. *Computational Mathematics*. MIR Publishers, Moscow, 1976. Translated from Russian by G.Yankovsky.

[23] Z. Du, M. Eleftheriou, J. Moreira, and C. Yap. Hypergeometric functions in exact geometric computation. In V. Brattka, M. Schoeder, and K. Weihrauch, editors, *Proc. 5th Workshop on Computability and Complexity in Analysis*, pages 55–66, 2002. Malaga, Spain, July 12-13, 2002. In Electronic Notes in Theoretical Computer Science, 66:1 (2002), `http://www.elsevier.nl/locate/entcs/volume66.html`.

[24] T. Dubé and C. K. Yap. A basis for implementing exact geometric algorithms (extended abstract), September, 1993. Paper from `ftp://cs.nyu.edu/pub/local/yap/exact/basis.ps.gz`.

[25] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer-Verlag, 1987.

[26] A. R. Forrest. Computational geometry and software engineering: Towards a geometric computing environment. In D. F. Rogers and R. A. Earnshaw, editors, *Techniques for Computer Graphics*, pages 23–37. Springer-Verlag, 1987.

[27] S. Fortune. Editorial: Special issue on implementation of geometric algorithms. *Algorithmica*, 27(1), 2000.

[28] S. J. Fortune and C. J. van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Transactions on Graphics*, 15(3):223–248, 1996.

[29] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. Mpfr: A multiple-precision binary floating-point library with correct round-

ing. Research Report 5753, INRIA, 2005. available electronically at http://hal.inria.fr/inria-000000818.

[30] S. Funke. Exact arithmetic using cascaded computation. Master's thesis, Max Planck Institute for Computer Science, Saarbrücken, Germany, 1997.

[31] S. Funke, K. Mehlhorn, and S. Näher. Structural filtering: A paradigm for efficient and exact geometric programs. In *Proc. 11th Canadian Conference on Computational Geometry*, 1999.

[32] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[33] C. F. Gauss. *Carl Friedrich Gauss Werke*. Göttingen, 1876.

[34] R. Geus, O. Skavhaug, and H. P. Langtangen. Python wrapper tools; a performance study. Talk at the EuroPython 2004 Conference, Gothenburg, Sweden, 2004. Presented by R. Geus.

[35] GNU MP Homepage, 2000. GNU MP (=GMP) is a free library for arbitrary precision arithmetic on integers, rational numbers and floating point numbers. URL `http://www.swox.com/gmp/`. GMP 3.1 released Aug 2000.

[36] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[37] R. W. Gosper. Acceleration of series. Artificial Intelligence Lab. Memo 340, M.I.T., 1974.

[38] G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. *MPFR*. LORIA, INRIA and INRIA Lorraine, 2.2.0 edition, 2005. available electronically at http://www.mpfr.org/.

[39] J. Harrison. Formal verification of floating point trigonometric functions. In *Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design*, pages 217–233, 2000.

[40] J. Harrison. Formal verification of ia-64 division algorithms. In *Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics*, pages 234–251, 2000.

[41] J. Harrison, T. Kubaska, S. Story, and P. T. P. Tang. The computation of transcendental functions on the ia-64 architecture. pages 234–251, 1999.

[42] S. Hert, M. Hoffmann, L. Kettner, S. Pion, and M. Seel. An adaptable and extensible geometry Kernel. In *Proc. 5th Int'l Workshop on Algorithm Engineering (WAE-01)*, pages 79–90, Berlin, 2001. Springer. Aarhus, Denmark, August 28 - 30, 2001.

[43] C. M. Hoffmann. The problems of accuracy and robustness in geometric computation. *IEEE Computer*, 22(3):31–42, March 1989.

[44] IEEE. IEEE Standard 754-1985 for binary floating-point arithmetic, 1985. ANSI/IEEE Std 754-1985. From The Institute of Electrical and Electronic Engineers, Inc.

[45] E. Jeandel. Évaluation rapide de fonctions hypergéométriques. Rapport Technique 242, INRIA, 2000. 17 pages.

[46] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A Core library for robust numerical and geometric computation. In *15th ACM Symp. Computational Geometry*, pages 351–359, 1999.

[47] M. Karasick, D. Lieber, and L. R. Nackman. Efficient Delaunay triangulation using rational arithmetic. *ACM Trans. on Graphics*, 10:71–91, 1991.

[48] D.-S. Kim, K. Yu, Y. Cho, D. Kim, and C. Yap. Shortest paths for disc obstacles. In A. L. et al., editor, *Proc. Computational Sci. and Its Applic. (ICCSA 2004)*, number 3045 in Lecture Notes in Computer Science, pages 62–70, 2004. Intl. Workshop on Comp. Geom. and Applic., at ICCSA 2004, S. Maria degli Angeli, Assisi (Perugia, Italy) May 14–17, 2004.

[49] LEDA Homepage, Since 1995. URL `http://www.mpi-sb.mpg.de/LEDA/`. Library of Efficient Data Structures and Algorithms (LEDA) Project. From the Max Planck Institute of Computer Science.

[50] V. Lefèvre. The generic multiple-precision floating-point addition with exact rounding (as in the mpfr library). In *Proceedings of the 6th Conference on Real Numbers and Computers*, pages 135–145, 2004.

[51] V. Lefèvre, J.-M. Muller, and A. Tisserand. Towards correctly rounded transcendentals. *IEEE Trans. Computers*, 47(11):1235–1243, 1998.

[52] C. Li. *Exact Geometric Computation: Theory and Applications*. Ph.d. thesis, New York University, Department of Computer Science, Courant Institute, Jan. 2001. Download from `http://cs.nyu.edu/exact/doc/`.

[53] C. Li, S. Pion, and C. Yap. Recent progress in exact geometric computation. *J. of Logic and Algebraic Programming*, 64(1):85–111, 2004. Special issue on "Practical Development of Exact Real Number Computation".

[54] C. Li and C. Yap. A new constructive root bound for algebraic expressions. In *12th ACM-SIAM Symp. on Discrete Algorithms*, pages 496–505, Jan. 2001.

[55] D. Lozier and F. Olver. Numerical evaluation of special functions. In W. Gautschi, editor, *Mathematics of Computation 1943–1993: A Half-Century of Computational Mathematics*, volume 48, pages 79–125. American Math. Soc., Providence, Rhode Island, 1994 and 2000. Proceedings of Symposia in Applied Mathematics. Updated version (2000) available from http://math.nist.gov/nesf/.

[56] Y. L. Luke. *Algorithms for the Computation of Mathematical Functions*. Academic Press, 1977.

[57] K. Mahler. Zur approximation der exponentialfunktionen und des logarithmus. *J. Reine Angew. Math. 166*, pages 118–136, 1932.

[58] M. Marden. *The Geometry of Zeros of a Polynomial in a Complex Variable*. Math. Surveys. American Math. Soc., New York, 1949.

[59] K. Mehlhorn and S. Schirra. Exact computation with `leda_real` – theory and geometric applications. In G. Alefeld, J. Rohn, S. Rump, and T. Yamamoto, editors, *Symbolic Algebraic Methods and Verification Methods*, pages 163–172, Vienna, 2001. Springer-Verlag.

[60] Z. A. Melzak. *Companion to Concrete Mathematics*. Wiley, New York, NY, 1973.

[61] M. Mignotte and D. Ştefănescu. *Polynomials: An Algorithmic Approach*. Springer, 1999.

[62] R. E. Moore. *Interval Analysis*. Series in Automatic Computation. Prentice Hall, Englewood Cliffs, NJ, 1966.

[63] MPFR Homepage, Since 2000. URL `http://www.mpfr.org/`. The MPFR Library.

[64] J.-M. Muller. *Elementary Functions: Algorithms and Implementation*. Birkhäuser, Boston, 1997.

[65] N. T. Müller. The iRRAM: Exact arithmetic in C++. In J. Blank, V. Brattka, and P. Hertling, editors, *Computability and Complexity in Analysis*, pages 222–252. Springer, 2000. 4th International Workshop, CCA 2000, Swansea, UK, September 17-19, 2000, Selected Papers, Lecture Notes in Computer Science, No. 2064.

[66] K. Mulmuley. *Computational Geometry: an Introduction through Randomized Algorithms*. Prentice-Hall, Inc, Englewood Cliffs, New Jersey, 1994.

[67] M. Nardin, W. Perger, and A. Bhalla. Algorithm 707, CONHYP: A numerical evaluator of the confluent hypergeometric function for complex arguments of large magnitudes. *ACM Trans. Math. Softw.*, 18(3):345–349, 1992.

[68] M. Nardin, W. Perger, and A. Bhalla. Numerical evaluation of the confluent hypergeometric function for complex arguments of large magnitudes. *J. Computational and Applied Math.*, 39:193–200, 1992.

[69] Y. Nesterenko and M. Waldschmidt. On the approximation of the values of exponential function and logarithm by algebraic numbers. *Mat. Zapiski*, 2:23–42, 196. Diophantine Approximations, Proc. of papers dedicated to the memory of Prof. N.I. Feldman, Moscow. Available from http://arxiv.org/abs/math.NT/0002047.

[70] K. C. Ng. Argument reduction for huge arguments: Good to last bits. Technical report, SunPro, 1992. available electronically at http://www.validgh.com/.

[71] J. O'Rourke. *Computational Geometry in C.* Cambridge University Press, second edition edition, 1998.

[72] K. Ouchi. Real/Expr: Implementation of an exact computation package. Master's thesis, New York University, Department of Computer Science, Courant Institute, Jan. 1997. Download from `http://cs.nyu.edu/exact/doc/`.

[73] N. M. Patrikalakis, W. Cho, C.-Y. Hu, T. Maekawa, E. C. Sherbrooke, and J. Zhou. Towards robust geometric modelers, 1994 progress report. In *Proc. 1995 NSF Design and Manufacturing Grantees Conference*, pages 139–140, 1995.

[74] S. Pion. Interval arithmetic: An efficient implementation and an application to computational geometry. In *Workshop on Applications of Interval Analysis to Systems and Control*, pages 99–110, 1999.

[75] S. Pion and C. Yap. Constructive root bound method for k-ary rational input numbers. In *19th ACM Symp. on Comp. Geometry*, pages 256–263, San Diego, California., 2003.

[76] J. Popken. Zur transzendenz von e. *Math. Z. 29*, pages 525–541, 1929.

[77] F. P. Preparata and M. I. Shamos. *Computational Geometry*. Springer-Verlag, 1985.

[78] Python Homepage, 1990. `http://www.python.org/`. Python Programming Language.

[79] Topical Python Software. `http://www.scipy.org/Topical_Software`. Topical Python Software.

[80] D. Richardson. Some undecidable problems involving elementary functions of a real variable. *The Journal of Symbolic Logic*, 33(4):511–520, 1968.

[81] D. Richardson. How to recognize zero. *J. of Symbolic Computation*, 24:627–645, 1997.

[82] D. Richardson. The uniformity conjecture. In J. Blank, V. Brattka, and P. Hertling, editors, *Computability and Complexity in Analysis*, pages 253–272. Springer, 2000. 4th International Workshop, CCA 2000, Swansea, UK, September 17-19, 2000, Selected Papers, Lecture Notes in Computer Science, No. 2064.

[83] Roberto Tamassia, et al. Strategic directions in computational geometry: Working group report. *Computing Surveys*, 28(4):591–606, Dec. 1996.

[84] S. Schirra. Robustness and precision issues in geometric computation. In J. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*. Elsevier Science Publishers, B.V. North-Holland, Amsterdam, 1999.

[85] H. Sekigawa. Using interval computation with the Mahler measure for zero determination of algebraic numbers. *Josai Information Sciences Researches*, 9(1):83–99, 1998.

[86] J. Sellen, J. Choi, and C. Yap. Precision-sensitive Euclidean shortest path in 3-Space. *SIAM J. Computing*, 29(5):1577–1595, 2000. Also: 11th ACM Symp. on Comp. Geom., (1995)350–359.

[87] D. Shanks. *Solved and Unsolved Problems in Number Theory*. Dover Publications, Inc, New York, NY, 1993.

[88] V. Sharma, Z. Du, and C. Yap. Robust Approximate Zeros. In G. S. Brodal and S. Leonardi, editors, *Proc. 13th European Symp. on Algorithms (ESA)*, volume 3669 of *Lecture Notes in Computer Science*, pages 874–887. Springer-Verlag, Apr. 2005. Palma de Mallorca, Spain, Oct 3-6, 2005.

[89] B. Stroustup. *The Design and Evolution of C++*. Addison Wesley, New York, NY, April 1994.

[90] J. van der Hoeven. Fast evaluation of holonomic functions. *Theor. Comput. Sci.*, 210(1):199–215, 1999.

[91] J. van der Hoeven. Fast evaluation of holonomic functions near and in regular singularities. *J. Symb. Comput.*, 31(6):717–743, 2001.

[92] C. Yap and K. Mehlhorn. Towards robust geometric computation, 2001. Invited White Paper. CSTB-NSF Conference on Fundamentals of Computer Science, Washington DC, July 25-26, 2001. See Appendix, **Computer Science: Reflections on/from the Field**, The National Academies Press, Washington DC, 2004.

[93] C. K. Yap. A new number core for robust numerical and geometric libraries. In *3rd CGC Workshop on Computational Geometry*, 1998. Invited Talk. Brown University, Oct 11–12, 1998.
Abstracts, `http://www.cs.brown.edu/cgc/cgc98/home.html`.

[94] C. K. Yap. On guaranteed accuracy computation. In F. Chen and D. Wang, editors, *Geometric Computation*, chapter 12, pages 322–373. World Scientific Publishing Co., Singapore, 2004.

[95] C. K. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 41, pages 927–952. Chapmen & Hall/CRC, Boca Raton, FL, 2nd edition, 2004. Expanded from 1997 version.

[96] C. K. Yap and T. Dubé. The exact computation paradigm. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, pages 452–492. World Scientific Press, Singapore, 2nd edition, 1995.

[97] J. Yu. *Exact arithmetic solid modeling*. Ph.D. dissertation, Department of Computer Science, Purdue University, West Lafayette, IN 47907, June 1992. Technical Report No. CSD-TR-92-037.