Electronic Notes in Theoretical Computer Science 66 No. 1 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume66.html 17 pages

Hypergeometric Functions in Exact Geometric Computation

Zilin Du [a,1], Maria Eleftheriou [b], Jose Moreira [b], Chee Yap [a,1]

[a] *Department of Computer Science, Courant Institute, New York University, 251 Mercer Street, New York, NY 10012.*

[b] *IBM T.J. Watson Research Center, Yorktown Heights, New York.*

Abstract

Most problems in computational geometry are algebraic. A general approach to address nonrobustness in such problems is Exact Geometric Computation (EGC). There are now general libraries that support EGC for the general programmer (e.g., `Core Library`, `LEDA Real`). But many applications require non-algebraic functions as well. In this paper, we wish to provide non-algebraic functions in the context of other EGC capabilities. We implemented a multiprecision hypergeometric series package which can be used to evaluate the elementary functions. This can be achieved relatively easily using the `Core Library` which supports a guaranteed precision level of accuracy. We address several issues of efficiency in such a hypergeometric package: automatic error analysis, argument reduction, preprocessing of hypergeometric parameters, and precomputed constants. Some preliminary experimental results are reported.

1 Introduction

There are two basic modes of numerical computation: those that assume fixed precision and those that require arbitrarily high precision. The former is by far the dominant mode, as it includes most of scientific computation. A second dichotomy in numerical computation is based on classifying computations as either algebraic or those that goes beyond algebraic. By definition, an **algebraic computation** may only use algebraic functions as primitives (for instance, $+, -, \times, \div, \sqrt{}$); a non-algebraic computation would invoke non-algebraic functions such as sin or log. A **algebraic problem** is one that can be solved using algebraic computations [22]. Most problems arising in computational geometry [7,1], and a large part of the geometric modeling are algebraic. To treat non-algebraic computation, we could go into the general setting of

[1] Supported by NSF/ITR Grant #CCR-0082056.

mathematical analysis, an area whose computability aspects is studied under computable analysis [19]. But numerical *algebraic* computation, being more specialized, admits much stronger techniques than are generally available.

In terms of the two dichotomies above, this paper addresses numerical computation that is non-algebraic and which requires arbitrarily high precision. More precisely, we want to evaluate multiprecision hypergeometric functions. What is interesting is that this capability is placed in a setting where a very strong assumption is generally made. This is the premise of **guaranteed precision**, first proposed in [20] and embodied in the `Core Library` accuracy API [10]. This assumption can be satisfied in an algebraic setting. The idea of guaranteed precision is a natural one: a user might want a particular computation to be "guaranteed up to 100 bits". Here, the 100 bits can be taken in a relative or in an absolute sense. It means the error (relative or absolute) of each computed numerical value is at most 2^{-100}. Such guarantees can be achieved if the computation uses our `Core Library` for its numerical engine. The problem is that, when we incorporate non-algebraic functions into our system, it is a major open problem whether a similar guarantee can be made.

The concept of guaranteed accuracy is rooted in a general solution to the widespread problem of numerical non-robustness in geometric algorithms [18,21]. This well-known problem plagues many scientific and engineering computations. For instance, no computer-aided design (CAD) software used in geometric modeling has any robustness guarantees – they can be made to crash or give qualitatively wrong results with a suitable choice of inputs. Practitioners have generally accepted this as the inevitable cost of approximate numerical computation. However, major economic costs can be attached to such unreliability. For instance, scientific productivity suffers when programmers spent large amount of time debugging numerical programs. In principle, such problems can be eliminated for a large class of problems using an approach we call **exact geometric computation** (EGC) [22]. Basically, EGC amounts to computing with guaranteed accuracy. It is important to realize that guaranteed accuracy does not require "exact arithmetic" but arithmetic that has "sufficient accuracy". This simple remark is actually a key to making EGC a practical form of computing. Techniques such as floating point filters [2,6,12], constructive root bounds [11,5,4,12], low-degree predicates [13], are among the techniques driven by the EGC mode of computation. There are currently two general numerical libraries that support the EGC mode: `LEDA Real` [3] and the `Core Library` [10]. Using them, the general programmer can easily construct fully robust geometric algorithms. Such robust algorithms are what distinguishes the major software packages such as `LEDA` [15] and `CGAL` [8,17].

The point of departure of this paper is the observation that many numerical computations require both algebraic and non-algebraic computations. Thus, in physical simulation applications, we may need to construct a mesh as well as solve some numerical differential equations over this domain; mesh construction is an algebraic process while the numerical solving is non-algebraic. Currently, `Core Library` and `LEDA Real` do not support non-algebraic functions. This paper takes a first step at filling this gap. We describe the design and implementation of a hypergeometric package in the `Core Library`. Most common non-algebraic functions such as $\exp(x), \log(x), \mathrm{erf}(x)$ and the trigonometric functions are hypergeometric. Indeed, all the non-algebraic mathematical functions in a standard library such as `math.h` (for `C/C++`) are hypergeometric. Jeandel [9] describes a recent effort to provide hypergeometric functions in a general multiprecision number package (in this case, *gmp*). While multiprecision arithmetic is a pre-requisite for EGC, it still lacks the critical EGC capability of exact comparison.

Is EGC possible for non-algebraic computation?

The design of the `Core Library` aims to make robust programs easily constructed by any programmer. Towards this end, we define in [20] 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 10 bits)
- Level IV: Mixed Accuracy (i.e., combinations of 3 previous levels)

The goal is to allow a single program to be run in any of these levels, just by calling the library. This facilitates debugging, experimentation and flexible choice of accuracy targets to suit the many applications that a single program might have. Core is designed so that most "ordinary `C++` programs" (written with no knowledge of `Core Library` in mind) can be adapted to use the `Core Library` with minimal modifications.

Level II has no guarantees: computing to 1000 bits do not guarantee that any fraction of 1000 bits are correct because of error propagation. Level III is a refinement of the **basic EGC requirement**, namely, the ability to make exact comparisons $x : y$. Thus, if we compute $x - y$ to *one bit* of relative accuracy, we can make the comparison $x : y$ without error. There is a fundamental gap between Levels II and III that may not be apparent: Level III is more than simply iterating a Level II computation with increasing precision. While we know how to provide Level III capability for all algebraic computation [21], it is an open question whether Level III is possible in the non-algebraic case. We call this the **fundamental problem** of EGC. Let us isolate this critical issue precisely: let Ω be a set of real or complex functions or constants. In practice, Ω contains $+, -, \times, n \in \mathbb{Z}$, among other things. The set of constant expressions over Ω is denoted $E(\Omega)$. The **value** of an expression $e \in E(\Omega)$ is a real (or complex) number, but it may also be undefined. The **Constant Zero Problem** for Ω, denoted $\mathrm{CZP}(\Omega)$, is to decide for a given $e \in E(\Omega)$, whether the value of e is defined and equal to 0. When Ω contains at least one non-algebraic function such as $\sin x$ or $\log x$, the Constant Zero Problem for Ω is not known to be decidable, and closely related to undecidable ones [14]. Yet, this is precisely what we need for guaranteed accuracy. So the fundamental problem of EGC is really a family of problems, CZP(Ω) for each Ω.

Contributions of This Paper.

- We describe the design and implementation of a multiprecision hypergeometric library. This is easily and naturally implemented using the Level III accuracy of the `Core Library`.
- This paper introduces the problem of processing of hypergeometric parameters for efficient evaluation of hypergeometric functions.
- We describe techniques for the automatic error analysis of hypergeometric functions.
- We address the problem of argument reduction. It also requires a form of automatic error analysis.
- A natural application of the library is in the computation of mathematical constants to arbitrary accuracy. Here we have both an application *of* the hypergeometric package, as well as an application *to* the hypergeometric package. We define file formats for storing these constants as well as access methods for these files. This has applications beyond the current library.

2 Hypergeometric Series

We review some basic facts of the hypergeometric series. A hypergeometric series $\sum_{k=0}^{\infty} t_k$ is one in which $t_0 = 1$ and the ratio of two consecutive terms is a rational function of the summation index k

$$\frac{t_{k+1}}{t_k} = \frac{P(k)}{Q(k)} x, \tag{1}$$

where $P(k)$ and $Q(k)$ are monic polynomials in k, and x is a constant called the **argument**. By factoring the polynomials $P(k)$ and $Q(k)$ we can write

$$\frac{t_{k+1}}{t_k} = \frac{P(k)}{Q(k)} x = \frac{(a_1 + k)(a_2 + k)\cdots(a_p + k)}{(b_1 + k)(b_2 + k)\cdots(b_q + k)(k+1)} x. \tag{2}$$

Note the presence of the factor $(k+1)$ in the denominator; this is a standard convention. Many common hypergeometric series have this factor; but in case $Q(k)$ does not naturally contain such a factor, we artificially attach this factor to $P(k)$ and $Q(k)$. 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 $(a)_0 = 1$. Using this symbol, the general expression for term t_k becomes

$$t_k = \frac{(a_1)_k (a_2)_k \cdots (a_p)_k}{(b_1)_k (b_2)_k \cdots (b_q)_k} \frac{x^k}{k!}. \tag{3}$$

Note that the factor $k! = (1)_k$ in the denominator comes from the factor $(k+1)$ in $Q(k)$. Thus hypergeometric series is completely defined by the sequences

$a = (a_1, a_2, \ldots, a_p)$ and $b = (b_1, b_2, \ldots, b_q)$. These are called the **upper** and **lower parameters** of the hypergeometric series. The b_i's may not be zero or negative integers (otherwise we will have a division by 0). A negative a_i, on the other hand, turns $\sum_{k\geq 0} t_k$ into a finite series and hence a polynomial in x. Note that the conventional factor of $k!$ in the denominator of t_k in (3) amounts to an *implicit* lower parameter of $b_0 = 1$. The hypergeometric series corresponding to these parameters is denoted

$$ {}_pF_q(a_1, a_2, ..., a_p; b_1, b_1, b_2, ..., b_q; x) = \sum_{k=0}^{\infty} \frac{(a_1)_k (a_2)_k \cdots (a_p)_k}{(b_1)_k (b_2)_k \cdots (b_q)_k} \frac{x^k}{k!}, \tag{4}$$

The above series is easily seen to converge for any complex x when $p \leq q$, and for $|x| < 1$ when $p = q+1$. The corresponding complex function (extended by continuation to the rest of the complex plane) is a **hypergeometric functions**. In this paper, we are only interested in the case where x as well as the parameters a_i, b_j are real.

Elementary Functions.

Most common elementary functions [16] are hypergeometric functions. For example, the usual series for $\exp(x)$ is $\sum_{k=0}^{\infty} \frac{x^k}{k!}$ with the initial term $t_0 = 1$. The ratio between two consecutive terms is $t_{k+1}/t_k = x/(k+1)$. From Equation (2), we see that $p = q = 0$ and hence

$$\exp(x) = {}_0F_0(;;x).$$

Table 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., $\sin x, \arcsin x$).

The standard series for $\log(1+x)$ in Table 1 has poor convergence properties. By subtracting the standard series for $\log(1+x)$ from $\log(1-x)$, we obtain

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

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 1, which are mostly computed through direct evaluation of listed hypergeometric series, we also compute $\tan(x)$, $\cot(x)$, and $\arccos(x)$ as

Table 1
The representation of some elementary functions in terms of hypergeometric series.

Elementary functions	Power series	Ratio t_{k+1}/t_k	Hypergeometric series
$\exp(x)$	$\sum_{k=0}^{\infty} \frac{x^k}{k!}$	$\frac{1}{(k+1)} x$	${}_0F_0(;;x)$
$\mathrm{erf}(x)$	$\frac{2x}{\sqrt{\pi}} \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)k!} x^{2k}$	$-x^2 \frac{(k+\frac{1}{2})}{(k+\frac{3}{2})(k+1)}$	$\left(\frac{2x}{\sqrt{\pi}}\right) {}_1F_1(\frac{1}{2}; \frac{3}{2}; -x^2)$
$\sin(x)$	$x \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k}}{(2k+1)!}$	$(-\frac{1}{4}x^2) \frac{1}{(k+\frac{3}{2})(k+1)}$	$x\ {}_0F_1(; \frac{3}{2}; \frac{-x^2}{4})$
$\cos(x)$	$\sum_{k=0}^{\infty} (-1)^k \frac{x^{2k}}{(2k)!}$	$-\frac{1}{4}x^2 \frac{1}{(k+\frac{1}{2})(k+1)}$	${}_0F_1(; \frac{1}{2}; \frac{-x^2}{4})$
$\arcsin(x)$	$x \sum_{k=0}^{\infty} \frac{(2k)!}{2^{2k}(2k+1)(k!)^2} x^{2k}$	$x^2 \frac{(k+\frac{1}{2})^2}{(k+\frac{3}{2})(k+1)}$	$x\ {}_2F_1(\frac{1}{2}, \frac{1}{2}; \frac{3}{2}; x^2)$
$\arctan(x)$	$x \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k}}{2k+1}$	$-x^2 \frac{(k+\frac{1}{2})}{(k+\frac{3}{2})}$	$x\ {}_2F_1(\frac{1}{2}, 1; \frac{3}{2}; -x^2)$
$\log(1+x)$	$x \sum_{k=0}^{\infty} \frac{(-x)^k}{k+1}$	$(-x) \frac{(k+1)(k+1)}{(k+2)(k+1)}$	$x\ {}_2F_1(1, 1; 2; -x)$

$$\tan(x) = \frac{\sin(x)}{\cos(x)}, \tag{5}$$

$$\cot(x) = \frac{\cos(x)}{\sin(x)}, \tag{6}$$

$$\arccos(x) = \arcsin\left(\sqrt{1-x^2}\right),\ 0 \le x \le 1. \tag{7}$$

3 Automatic Error Analysis

To achieve guaranteed precision automatically, we need to analyze the hypergeometric parameters $a_1, \ldots, a_p$, $b_1, \ldots, b_q$. We may as well assume that $p \le q+1$, since otherwise the series diverge. Moreover, we assume the upper and lower parameters to be positive. Several cases have to be distinguished in this analysis.

To fix the terminology, let $S = \sum_{i=0}^{\infty} t_i$, $S_n = \sum_{i=0}^{n-1} t_i$ and $R_n = S - S_n$. Our goal is to compute a value $\widetilde{S}$ such that

$$|\widetilde{S} - S| \le \varepsilon$$

where $\varepsilon \ge 0$ is given (absolute) error bound. If we evaluate the series to n terms only, we obtain an approximate value $\widetilde{S}_n$, and so $|S_n - \widetilde{S}_n|$ is the **evaluation error**. The term $|R_n|$ is the **truncation error**. In general, these quantities are a function of an argument x, so $S = S(x), t_i = t_i(x)$, etc.

Sometimes, we have a third source of error: the argument x is only an approximated of some true value x^* where

$$|x - x^*| \le \varepsilon'.$$

where ε' is the **argument error bound**. For instance, when we perform argument reduction, such errors will be introduced. In the present section, we assume no argument error, *i.e.*, $\varepsilon' = 0$. The case $\varepsilon' > 0$ will be treated under argument reduction below.

When t_n are hypergeometric terms, write

$$(8) \qquad f(n) = P(n)/Q(n), \qquad P(n) = \prod_{i=1}^{p}(n + a_i), \quad Q(n) = \prod_{j=0}^{q}(n + b_j).$$

Using the `Core Library`, it turns out that the approximation error $|S_n - \widetilde{S}_n|$ is a non-issue because we simply rely on the `Core Library` facility to compute an expression to any desired error bounds. We can simply set the approximation error to $\varepsilon/2$. It remains to bound the truncation error by $\varepsilon/2$. Our goal is to determine the n such that $|R_n| \le \varepsilon/2$.

(A) Alternating case.

The simplest case is when the t_n's are alternating in sign, *i.e.*, $t_n t_{n+1} \le 0$ for all $n \in \mathbb{N}$. In this case, we have the well-known fact: *if $|t_i| \ge |t_{i+1}|$ for all $i \ge n$ then $|R_n| \le |t_n|$ and $R_n t_n \ge 0$.* To apply this result to hypergeometric series, let $\overline{a} = (a_1 + \cdots + a_p)/p$, $\overline{b} = (1 + b_1 + \cdots + b_q)/(q+1)$, and $a^* = \max_{i=1,\ldots,p} a_i$. We have the following sufficiency criteria:

Lemma 3.1 *Let $R_n = \sum_{i \ge n} t_i$ be alternating. Then $|R_n| \le |t_n|$ in the following two situations:*
(i) Case $p = q + 1$: $\overline{b} > \overline{a}$ and

$$n \ge \frac{2^p (a^*)^2}{(\overline{b} - \overline{a})p}.$$

(ii) Case $p \le q$: $n \ge \max\{2, 2^p a^*\}$.

Proof. (i) We have $P(n) \le n^p + p\overline{a}n^{p-1} + 2^p (a^*)^2 n^{p-2}$ and $Q(n) \ge n^{q+1} + q\overline{b}n^q$. Then if $n \ge \frac{2^p (a^*)^2}{(\overline{b} - \overline{a})p}$ we conclude that $P(n)/Q(n) \le 1$, i.e., $|t_{n+1}| \le |t_n|$.
(ii) If $p \le q$, it is sufficient to show that

$$1 + (p\overline{a})/n + 2^p (a^*)^2/n^2 \le n.$$

This is clear. **Q.E.D.**

(B) The Geometric Case.

When the series R_n is not alternating, we can sometimes can use the following somewhat weaker bound:

Lemma 3.2 *If $xf(n) < 1$ and for all $i \ge n$, $f(i) \ge f(i+1)$, then $|R_n| \le \frac{t_n}{1 - xf(n)}$.*

Proof. Note that for $i \geq 0$,

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

Summing,

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

Q.E.D.

In particular, the geometric case would apply whenever $p \leq q$. For instance, among the non-alternating series in Table 1, the series for $\exp(x)$ falls under this case.

(C) The Hard Case.

Among the functions in Table 1, the arcsin series is covered by neither case (A) nor case (B). It falls under what we call the "hard case". This means that $p = q + 1$ and the series is not alternating. In this case, we provide an á priori bound on $|R_n|$ assuming that $|x| < 1$ and certain conditions hold.

Lemma 3.3 *If $|x| < 1$ and $f(i) < 1$ for all $i \geq n$, then $|R_n| \leq \frac{t_n}{1-x}$.*

In proof, we have

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

For instance, assume

$$\sum_{i=1}^{p} a_i < \sum_{j=0}^{q} b_j \tag{9}$$

then for the lemma will hold for sufficiently large n. Moreover, we see that $t_n \to 0$ as $n \to \infty$. Hence, we simply choose n such that $t_n \leq \varepsilon(1-x)/2$.

We may verify that the arcsin series satisfies the inequality (9), and moreover, x is replaced by x^2. The analysis for (9) amounts to looking at the coefficient of n^q in $P(n)$ and in $Q(n)$. So in case

$$\sum_{i=1}^{p} a_i = \sum_{j=0}^{q} b_j,$$

we can looking at the coefficient of n^{q-1} in the polynomials $P(n)$ and $Q(n)$, etc.

Implementation.

The above bounds are applied directly to the elementary functions in Table 1. However, when the user requests the evaluation of a hypergeometric

function with some general parameters, here is the proposed algorithm:
(1) First we check if $x < 0$. If so, we can apply case (A) for alternating series.
(2) Else, we check if $p \leq q$. If so, we apply case (B) for the geometric case.
(3) Else, we check if $x < 1$ and $p = q + 1$. If so, we apply case (C).
(4) Else, we report an error.
But more generally, our implementation will accept a user-defined error bound function $B(x, \varepsilon)$ such that if $n = B(x, \varepsilon)$ then $|R_n| \leq \varepsilon$ is guaranteed. If such a function is provided, it is used instead of the above analysis.

In general, to determine the n such that $|R_n| \leq \varepsilon/2$, we need to upper bound the value of t_n. A simple method is to accumulate the terms in S_n while at the same time check the next term t_n. On the face of it, computing t_n requires an iteration, as the number of factors in t_n grows with n. But in many situations, the number of factors in t_n is independent of n because of cancellation. E.g., in $\lg(1 + x)$. In this case, we can determine n in constant time by a direct evaluation. This yields a much faster implementation. This situation is exploited under parameter processing (see below).

4 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. 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 [16, 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 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 4.1 *For $\varepsilon > 0$, we have the following bounds:*

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

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

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

Then

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

Q.E.D.

Natural Log function.

We implement a standard argument reduction: 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 $k = \lfloor \lg x \rfloor$.
2. Compute $\widetilde{\log(2)}$ as an approximation of $\log(2)$ to absolute error $\le \varepsilon 2^{-k}$.
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 $\le \varepsilon/4$.

5. Return $z = k\widetilde{\log(2)} + \widetilde{\log(\widetilde{r})}$.
It is easy to show, using lemma 4.1 that this procedure is correct, *i.e.*, $|z - \log(x)| \leq \varepsilon$. Moreover, each of the steps is effectively computable (in fact, they are straightforward using the `Core Library`). Note that step 2 requires an approximation to the constant $\log(2)$, and in our system, this is precomputed (see Section 6).

Exponential function.

Again, the reduction is standard. Let $k = \lfloor x/\log(2) \rfloor$ and $r = x - k\log(2)$. Then

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

1. First, we compute k (the details of this computation is 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 $\varepsilon 2^{-k} e^{-2}$.
3. Compute $\widetilde{\exp(\widetilde{r})}$ as an approximation to $\exp(\widetilde{r})$ to absolute error $\varepsilon 2^{-k}$.
4. Return $z = 2^k \widetilde{\exp(\widetilde{r})}$.

Trigonometric functions.

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 4.1, 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.

5 Hypergeometric Parameter Pre-processing

Recall that an extra hypergeometric parameters are sometimes artificially introduced in order to achieve the standard form of these series. For instance, one of the upper parameters in $x\ {}_2F_1(1, 1; 2; -x)$ $(= \log(1 + x))$ amounts to cancelling the implicit lower parameter of $b_0 = 1$. 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 differs by an integer, cancellations occur and one can gain improvements in efficiency by recognizing this.

We outline a general algorithm for 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 pairs 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 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	1.	5.01	6.99	8.89	10.46
Parameter Preprocessing	0.22	0.52	0.88	1.35	1.83
Speedup:	4.5	9.6	7.9	6.8	5.7

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 $(B_1 < B_2 < \cdots < B_n)$. 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

$$S = \sum_{i=1}^{m} |A_i - B_{\alpha(i)}| \tag{10}$$

is minimized. 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 (10) 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 $O(mn)$.

6 Mathematical Constants: Evaluation, File Formats and Access

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 $\mathrm{erf}(x)$, we need $1/\sqrt{\pi}$. While it is possible to compute these constants on the fly, we should improve performance by precomputing 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

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

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	0.04	0.50	2.49	5.88	10.66	17.19	21.38	107.61
Precomputed	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. 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. Ths 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 allows comments, which also facilitates memoization and collaboration. The formal specification is[3] distributed with `Core Library` version 1.3 or higher.

7 Design and Implementation of Hypergeometric Class

We designed and implemented in `C++` a hypergeometric class, based on the `Real` and `Expr` classes of `Core Library`. The main function is

```
static Real MultiPrecision::HyperGeometric::
F(vector <Expr*> &A, vector <Expr*> &B, Expr &fx, Expr &fz);.
```

This is used to evaluating 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 class `ElementaryFunction` as an interface for the common elementary functions: log, exp, erf, sin, cos, tan, arccos, arcsin, arctan. This is basically a registry service, to associate with each function a name (i.e., a string key), its hypergeometric parameters, and other evaluation arguments. Automatic error analysis is also performed

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

during the preregistration. The registered functions can then by invoked by name. Users can register additional functions.

8 Conclusions

This paper describes an effort to incorporate non-algebraic functions into a framework that normally supports guaranteed precision. The implementation of such a package is naturally achieved using the basic capabilities of the `Core Library`. There are several issues of efficiency and automation which we have addressed: automatic error bound computation, hypergeometric parameter processing, argument reduction, and finally constant precomputation, storage and retrieval. Using our package it is easy for a user to add other hypergeometric functions, replace the hypergeometric series by more efficient ones, etc. Other topics for future work include improved automatic error bounds and acceleration of series.

The hypergeometric package is distributed with the `Core Library`, at our website `http://cs.nyu.edu/exact/`.

References

[1] J.-D. Boissonnat and M. Yvinec. *Algorithmic Geometry.* Cambridge University Press, 1997. Translated by Hervé Brönnimann.

[2] H. Brönnimann, C. Burnikel, and S. Pion. Interval arithmetic yields efficient dynamic filters for Computation Geometry. *ACM Symp. on Computational Geometry*, 14:165–174, 1998.

[3] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. Exact geometric computation made easy. In *Proc. 15th ACM Symp. Comp. Geom.*, pages 341–450, 1999.

[4] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. A strong and easily computable separation bound for arithmetic expressi ons involving radicals. *Algorithmica*, 27:87–99, 2000.

[5] C. Burnikel, S. Funke, K. Mehlhorn, and S. Schirra. A separation bound for real algebraic expressions, 2001. preprint (submitted to journal).

[6] C. Burnikel, S. Funke, and M. Seel. Exact geometric comptuation using cascading. *Int'l. J. Computational Geometry and Applications*, 11(3):245–266, 2001. Special Issue.

[7] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.

[8] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schoenherr. The CGAL kernel: a basis for geometric computation. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*,

pages 191–202, Berlin, 1996. Springer. Lecture Notes in Computer Science No. 1148; Proc. 1st ACM Workshop on Applied Computational Geometry (WACG), Federated Computing Research Conference 1996, Philadelphia, USA.

[9] E. Jeandel. Évaluation rapide de fonctions hypergéométriques. Rapport Technique 242, INRIA, 2000. 17 pages.

[10] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A Core library for robust numerical and geometric libraries. In *15th ACM Symp. Computational Geometry*, pages 351–359, 1999.

[11] C. Li and C. Yap. A new constructive root bound for algebraic expressions. In *Proc. 12th ACM-SIAM Symposium on Discrete Algorithms*, pages 496–505. ACM and SIAM, Jan. 2001.

[12] C. Li and C. Yap. Recent progress in exact geometric computation. In S. Basu and L. Gonzalez-Vega, editors, *Proc. DIMACS Workshop on Algorithmic and Quantitative Aspects of Real Algebraic Geometry in Mathematics and Computer Science, March 12 - 16, 2001.*, DIMACS Books Series. American Math. Society, 2001. Submitted. Paper download `http://cs.nyu.edu/exact/doc/`.

[13] G. Liotta, F. Preparata, and R. Tamassia. Robust proximity queries: an illustration of degree-driven algorithm design. *SIAM J. Computing*, 2001. to apear.

[14] Y. V. Matiyasevich. *Hilbert's Tenth Problem.* The MIT Press, Cambridge, Massachusetts, 1994.

[15] K. Mehlhorn and S. Näher. LEDA: a platform for combinatorial and geometric computing. *CACM*, 38:96–102, 1995.

[16] J.-M. Muller. *Elementary Functions: Algorithms and Implementation.* Birkhäuser, Boston, 1997.

[17] M. Overmars. Designing the computational geometry algorithms library CGAL. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, pages 53–58, Berlin, 1996. Springer. Lecture Notes in Computer Science No. 1148; Proc. 1st ACM Workshop on Applied Computational Geometry (WACG).

[18] S. Schirra. Precision and robustness in geometric computations. In M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, editors, *Algorithmic Foundations of Geographic Information Systems*, volume 1340 of *Lecture Notes Comp. Science*, pages 255–287. Springer, 1997. Chapter 9.

[19] K. Weihrauch. *Computable Analysis.* Springer, Berlin, 2000.

[20] C. Yap. A new number core for robust numerical and geometric libraries. In *3rd CGC Workshop on Geometric Computing*, 1998. Invited Talk. Brown University, Oct 11–12, 1998. For abstracts, see `http://www.cs.brown.edu/cgc/cgc98/home.html`.

[21] C. K. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 35, pages 653–668. CRC Press LLC, Boca Raton, FL, 1997.

[22] C. K. Yap. Towards exact geometric computation. *Computational Geometry: Theory and Applications*, 7:3–23, 1997. Invited talk, Proceed. 5th Canadian Conference on Comp. Geometry, Waterloo, Aug 5–9, 1993.