Validated Computation of Certain Hypergeometric Functions

MICHEL COLMAN, ANNIE CUYT, and JORIS VAN DEUN, Universiteit Antwerpen

We present an efficient algorithm for the validated high-precision computation of real continued fractions, accurate to the last digit. The algorithm proceeds in two stages. In the first stage, computations are done in double precision. A forward error analysis and some heuristics are used to obtain an a priori error estimate. This estimate is used in the second stage to compute the fraction to the requested accuracy in high precision (adaptively incrementing the precision for reasons of efficiency). A running error analysis and techniques from interval arithmetic are used to validate the result.

As an application, we use this algorithm to compute Gauss and confluent hypergeometric functions when one of the numerator parameters is a positive integer.

Categories and Subject Descriptors: G.1.2 [**Numerical Analysis**]: Approximation—*Rational approximation*; G.4 [**Mathematical Software**]: Reliability and robustness

General Terms: Algorithms, Reliability

Additional Key Words and Phrases: Validated software, continued fractions, hypergeometric functions

ACM Reference Format:
Colman, M., Cuyt, A., and Van Deun, J. 2011. Validated computation of certain hypergeometric functions. ACM Trans. Math. Softw. 38, 2, Article 11 (December 2011), 20 pages.
DOI = 10.1145/2049673.2049675 http://doi.acm.org/10.1145/2049673.2049675

1. INTRODUCTION

The hypergeometric function of Gauss [Abramowitz and Stegun 1964, Chap. 15] is the analytic continuation of the series

$$ {}_2\mathrm{F}_1(a,b;c;z) = 1 + \frac{ab}{c}\frac{z}{1!} + \frac{a(a+1)b(b+1)}{c(c+1)}\frac{z^2}{2!} + \cdots . \tag{1}$$

The parameters can be arbitrary complex numbers, provided that c is not a negative integer or zero. Defining

$$P_{2k} = {}_2\mathrm{F}_1(a+k, b+k; c+2k; z),$$
$$P_{2k+1} = {}_2\mathrm{F}_1(a+k, b+k+1; c+2k+1; z),$$

for $k = 0, 1, 2, \ldots$, it is shown in Jones and Thron [1980, p. 200] that

$$P_n = P_{n+1} + c_{n+1} z P_{n+2}, \quad n \geq 0$$

Authors' address: Middelheimlaan l, B-2020 Antwerpen, Belgium; email: annie.cuyt@ua.ac.be.

DOI 10.1145/2049673.2049675 http://doi.acm.org/10.1145/2049673.2049675

where

$$c_{2k+1} = -\frac{(a+k)(c-b+k)}{(c+2k)(c+2k+1)}, \quad k \geq 0, \tag{2}$$

$$c_{2k} = -\frac{(b+k)(c-a+k)}{(c+2k-1)(c+2k)}, \quad k \geq 1. \tag{3}$$

From this recurrence relation, one obtains the following continued fraction for the ratio of two hypergeometric functions [Jones and Thron 1980, p. 200],

$$\frac{{}_2\mathrm{F}_1(a,b;c;z)}{{}_2\mathrm{F}_1(a,b+1;c+1;z)} = 1 + \cfrac{c_1 z}{1 + \cfrac{c_2 z}{1 + \ddots}}, \tag{4}$$

where $z \notin [1,\infty)$. According to Pincherle's theorem [Pincherle 1894; Gautschi 1967], this continued fraction converges to the ratio of two *minimal* solutions of the three-term recurrence relation. Because ${}_2\mathrm{F}_1(a,0;c;z) = 1$ it is clear that ${}_2\mathrm{F}_1(a,n;c;z)$ for positive integer n can be computed as a product of continued fractions [Lorentzen 1992]

$${}_2\mathrm{F}_1(a,n;c;z) = \left[\prod_{k=1}^{n} \frac{{}_2\mathrm{F}_1(a,k-1;c-n+k-1;z)}{{}_2\mathrm{F}_1(a,k;c-n+k;z)}\right]^{-1}$$

or using the algorithm described in Section 3 of Gautschi [1967]. Both require the computation of one or more continued fractions of the form (4).

The case of the confluent hypergeometric function is very similar. This function is defined by the series

$${}_1\mathrm{F}_1(a;b;z) = 1 + \frac{a}{b}\frac{z}{1!} + \frac{a(a+1)}{b(b+1)}\frac{z^2}{2!} + \cdots, \tag{5}$$

whenever b is not a negative integer or zero. Now we define

$$P_{2k} = {}_1\mathrm{F}_1(a+k;b+2k;z),$$
$$P_{2k+1} = {}_1\mathrm{F}_1(a+k+1;b+2k+1;z),$$

for $k = 0, 1, 2, \ldots$ and we then have Jones and Thron [1980, p. 206]

$$P_n = P_{n+1} + d_{n+1} z P_{n+2}, \quad n \geq 0$$

where

$$d_{2k+1} = -\frac{b-a+k}{(b+2k)(b+2k+1)}, \quad k \geq 0, \tag{6}$$

$$d_{2k} = \frac{a+k}{(b+2k-1)(b+2k)}, \quad k \geq 1. \tag{7}$$

The corresponding continued fraction for the ratio of two confluent hypergeometric functions is given by Jones and Thron [1980, Chap. 6]

$$\frac{{}_1\mathrm{F}_1(a;b;z)}{{}_1\mathrm{F}_1(a+1;b+1;z)} = 1 + \cfrac{d_1 z}{1 + \cfrac{d_2 z}{1 + \ddots}},$$

where z can be anywhere in the complex plane. Also in this case ${}_1\mathrm{F}_1(0;b;z) = 1$ so that ${}_1\mathrm{F}_1(n;b;z)$ for positive integer n can be computed as described previously.

In addition, several functions appear for a special choice of the parameters in either (1) or (5) instead of as a (product of) quotient(s) of (confluent) hypergeometric funtions. Many of these also enjoy continued fraction representations [Cuyt et al. 2008], such as the incomplete gamma functions, the error and complimentary error function, the Bessel and spherical Bessel functions of integer order, Dawson's integral, the exponential integrals, to name just a few.

Details on the computation of the product (and how to distribute the error over the individual factors in the product) are given in Backeljauw et al. [2009]. Here we limit our attention to the computation of the continued fractions. Of course, there already exists a vast literature concerning the computation of hypergeometric functions, and several software implementations are available. We refer to Section 5.5 in Lozier and Olver [1994] for a survey of the literature and software up to 1994. An updated version from March 2000 can be obtained online at http://math.nist.gov/mcsd/Reports/2001/nesf+. It is far from our ambition to provide the ultimate algorithm to compute hypergeometric functions, but the continued fractions given above nicely illustrate the more general framework of this paper. Many special functions satisfy continued fraction expansions that, in combination with Taylor series expansions, often cover a large part of the domain. This article is part of on ongoing project to provide a generic framework for the validated multiprecision computation of special functions, as described in more detail in Backeljauw et al. [2009]. At the moment, this generic technique is based on a combination of Taylor series and continued fractions, but it will be extended in the future to incorporate other methods. The fact that we aim to provide *validated* software, sets this project aside from the available software mentioned in Lozier and Olver [1994], which for some functions (such as the hypergeometric) may be more efficient, but in general does not provide guaranteed reliable error bounds. In fact, as far as we know, existing validated software for mathematical functions (usually based on interval arithmetic) is mostly limited to the elementary functions.

So we discuss an algorithm for the validated high-precision computation of continued fractions of the form

$$\mathop{\mathrm{K}}_{n=1}^{\infty} \frac{a_n}{1} := \cfrac{a_1}{1+\cfrac{a_2}{1+\cfrac{a_3}{1+\ddots}}},$$

where the a_n are real. *Validated* in this context means that the returned result is *guaranteed* to have the requested accuracy. We limit our attention to fractions of this form because almost any continued fraction with real coefficients can easily be transformed into this form [Cuyt et al. 2008, Ch. 1].

The algorithm proceeds in two stages. First, a double precision computation uses forward error analysis and some heuristics to obtain an index n and an error estimate for the corresponding approximant. In the second stage, we compute this approximant in high precision and use a running error analysis and some techniques from interval arithmetic to validate the result. The working precision is dynamically adapted to at least the actually achieved accuracy at any given point. This greatly increases speed, as other algorithms use a large share of their time working with insignificant digits. On the other hand, it also enables the use of *more* precision if necessary to overcome a predicted loss of accuracy along the way.

The algorithm does not depend on any specific multiprecision library and can be implemented using any of the freely (or commercially) available libraries. Ideally, this library should support directional rounding, but workarounds can be made to simulate rounding operations [Kearfott 1996].

Sections 2 and 3 provide some necessary theoretical background. The error analysis underlying our algorithm is done in Section 4, and the algorithm itself is discussed in detail in Section 5. Some caveats and implementation issues are given in Section 6. To end this article, we explain how this algorithm is used to compute the Gauss and confluent hypergeometric functions, and some examples are given.

2. PRELIMINARIES

Two outstanding references for the theory of continued fractions are Jones and Thron [1980] and Lorentzen and Waadeland [1992]. We only give a very short and intuitive introduction to this subject and refer to these books for more information.

A continued fraction f is an infinite expression of the form

$$f = \mathop{\mathrm{K}}_{n=1}^{\infty} \frac{a_n}{b_n} := \cfrac{a_1}{b_1 + \cfrac{a_2}{b_2 + \cfrac{a_3}{b_3 + \ddots}}},$$

the value of which corresponds to the limit of the sequence of *approximants*

$$f_1 = \frac{a_1}{b_1}, \quad f_2 = \cfrac{a_1}{b_1 + \cfrac{a_2}{b_2}}, \quad f_3 = \cfrac{a_1}{b_1 + \cfrac{a_2}{b_2 + \cfrac{a_3}{b_3}}}, \quad \cdots$$

if this limit exists. In that case, the continued fraction is said to *converge*. The a_n and b_n are called *partial numerators* and *partial denominators*, respectively.

We define the *tail* t_i as

$$t_i := \mathop{\mathrm{K}}_{n=i+1}^{\infty} \frac{a_n}{b_n} = \cfrac{a_{i+1}}{b_{i+1} + \cfrac{a_{i+2}}{b_{i+2} + \ddots}}.$$

We also define the *linear fractional transformation*

$$s_n : x \to \frac{a_n}{b_n + x}.$$

Note that the continued fraction consists of a composition of such linear fractional transformations. In fact, the nth approximant is given by

$$f_n = s_1 \circ s_2 \circ \cdots \circ s_n(0) \tag{8}$$

and if we replace 0 by t_n then we obtain the continued fraction f. It is well known [Lorentzen 2003] that replacing 0 by a good tail estimate $\tilde{t}_n \approx t_n$ speeds up the convergence of the sequence of approximants. These are now called *modified* approximants, and we denote them by $f_n(\tilde{t}_n)$. Note that $f_n(0) = f_n$ and $f_n(t_n) = f$.

As discussed in the introduction, we assume that all b_n are equal to 1, since this is not an essential restriction (as long as all b_n are different from zero, the fraction can always be transformed to a fraction that has the same sequence of approximants, and where all b_n are equal to 1). A very simple example of such a continued fraction is the 1-*periodic* fraction where all a_n are equal to a. It is not difficult to prove that

$$\mathop{\mathrm{K}}_{n=1}^{\infty} \frac{a}{1} = \frac{-1 + \sqrt{1+4a}}{2}, \tag{9}$$

if $a \geq -1/4$. Otherwise, the fraction diverges. Similarly, the 2-*periodic* continued fraction with $a_{2k+1} = a$ and $a_{2k} = b$ equals (note the abuse of notation, which is used in the sequel also)

$$\mathop{\mathrm{K}}_{n=1}^{\infty} \frac{a, b}{1} = \frac{a - b - 1 \pm \sqrt{4a + (1 - a + b)^2}}{2}, \tag{10}$$

where the plus sign should be taken if $a + b > -1$ and the minus sign otherwise. This fraction also only converges when the number under the square root is positive.

3. TAIL ENCLOSURES

An essential part of the algorithm consists in finding an interval that is guaranteed to contain the tail t_n for a particular n (this interval need not be small, as long as it does not contain the value -1). The formulas in this section are given for t_0, but it is clear that formulas for t_i are obtained by simply replacing a_n with a_{n+i}. The formulas are easily proved by induction, so we omit the proofs. As usual, f_n refers to the nth approximant.

3.1. Using Finite Approximants

We distinguish between the case where all a_n are positive (eventually from a certain n on) and the case where all a_n are negative.

(1) If $a_n > 0$, then

$$0 < f_2 < f_4 < \cdots < \mathop{\mathrm{K}}_{n=1}^{\infty} \frac{a_n}{1} < \cdots < f_3 < f_1.$$

(2) If $-\frac{1}{4} < a_n < 0$, then

$$-\frac{1}{2} < 2a_1 < \frac{a_1}{1 + 2a_2} < \cdots < \mathop{\mathrm{K}}_{n=1}^{\infty} \frac{a_n}{1} < \cdots < f_3 < f_2 < f_1.$$

These inequalities are special cases of the following more general formulas:

(1) If $a_n > 0$ and $0 \leq w_{1,k} \leq t_k \leq w_{2,k}$, then

$$f_{2k}(w_{1,2k}) \leq \mathop{\mathrm{K}}_{n=1}^{\infty} \frac{a_n}{1} \leq f_{2k+1}(w_{1,2k+1}),$$

$$f_{2k}(w_{2,2k}) \geq \mathop{\mathrm{K}}_{n=1}^{\infty} \frac{a_n}{1} \geq f_{2k+1}(w_{2,2k+1}).$$

(2) If $-\frac{1}{4} < a_n < 0$ and $-\frac{1}{2} \leq w_{1,k} \leq t_k \leq w_{2,k}$, then

$$f_k(w_{1,k}) \leq \mathop{\mathrm{K}}_{n=1}^{\infty} \frac{a_n}{1} \leq f_k(w_{2,k}).$$

3.2. Using Periodic Continued Fractions

The values of 1-periodic and 2-periodic continued fractions are given by (9) and (10). These can be used to obtain tail estimates $\bar{t}_n$. We use the notation $a_\infty := \lim_{n\to\infty} a_n$ and $t_\infty := \lim_{n\to\infty} t_n = \mathrm{K}\frac{a_\infty}{1}$, whenever these values exist.

(1) If $-\frac{1}{4} \le L \le a_n \le R \le 0$, then

$$-\frac{1}{2} \le \mathop{\mathrm{K}}\frac{L}{1} \le \mathop{\mathrm{K}}_{n=1}^{\infty}\frac{a_n}{1} \le \mathop{\mathrm{K}}\frac{R}{1} \le 0.$$

(2) If $-\frac{1}{4} \le L \le a_n \le R \le 0$ and $a_{2n} \ge a_\infty$ and $a_{2n+1} \le a_\infty$, then

$$-\frac{1}{2} \le \mathop{\mathrm{K}}\frac{L}{1} \le \mathop{\mathrm{K}}\frac{L, a_\infty}{1} \le \mathop{\mathrm{K}}_{n=1}^{\infty}\frac{a_n}{1} \le \mathop{\mathrm{K}}\frac{a_\infty, R}{1} \le \mathop{\mathrm{K}}\frac{R}{1} \le 0.$$

(3) If $-\frac{1}{4} \le L \le a_n \le R \le 0$ and $a_{2n} \le a_\infty$ and $a_{2n+1} \ge a_\infty$, then

$$-\frac{1}{2} \le \mathop{\mathrm{K}}\frac{L}{1} \le \mathop{\mathrm{K}}\frac{a_\infty, L}{1} \le \mathop{\mathrm{K}}_{n=1}^{\infty}\frac{a_n}{1} \le \mathop{\mathrm{K}}\frac{R, a_\infty}{1} \le \mathop{\mathrm{K}}\frac{R}{1} \le 0.$$

(4) If $0 \le L \le a_n \le R$, then

$$0 \le \mathop{\mathrm{K}}\frac{L, R}{1} \le \mathop{\mathrm{K}}_{n=1}^{\infty}\frac{a_n}{1} \le \mathop{\mathrm{K}}\frac{R, L}{1}.$$

(5) If $0 \le L \le a_n \le R$ and $a_{2n} \ge a_\infty$ and $a_{2n+1} \le a_\infty$, then

$$\begin{aligned} 0 \;&\le\; \mathop{\mathrm{K}}\frac{L, R}{1} \le \mathop{\mathrm{K}}_{n=1}^{\infty}\frac{a_n}{1} \le \frac{a_1}{1+(a_2/1+t_\infty)} \\ &\le\; \frac{a_1}{1+t_\infty} \le \mathop{\mathrm{K}}\frac{a_1}{1} \le \mathop{\mathrm{K}}\frac{R, L}{1}. \end{aligned}$$

(6) If $0 \le L \le a_n \le R$ and $a_{2n} \le a_\infty$ and $a_{2n+1} \ge a_\infty$, then

$$\begin{aligned} 0 \;&\le\; \mathop{\mathrm{K}}\frac{L, R}{1} \le \mathop{\mathrm{K}}\frac{a_1}{1} \le \frac{a_1}{1+t_\infty} \le \frac{a_1}{1+(a_2/1+t_\infty)} \\ &\le\; \mathop{\mathrm{K}}_{n=1}^{\infty}\frac{a_n}{1} \le \mathop{\mathrm{K}}\frac{R, L}{1}. \end{aligned}$$

(7) If $0 \le a_\infty \le a_{n+1} \le a_n$, then

$$\begin{aligned} t_\infty &\le \frac{a_1}{1+(a_2/1+t_\infty)} \\ &\le \mathop{\mathrm{K}}_{n=1}^{\infty}\frac{a_n}{1} \le \frac{a_1}{1+t_\infty}. \end{aligned}$$

(8) If $0 \le a_n \le a_{n+1} \le a_\infty$, then

$$\frac{a_1}{1+t_\infty} \le \mathop{\mathrm{K}}_{n=1}^{\infty}\frac{a_n}{1} \le \frac{a_1}{1+(a_2/1+t_\infty)} \le t_\infty.$$

(9) If $-1 < L \le a_n \le R$ and $a_{2n}a_{2n+1} \le 0$ and $a_\infty = 0$ and

$$- \ a_1 > 0, \text{ then } 0 < a_1 \le f_4 \le \mathop{\mathrm{K}}_{n=1}^{\infty} \frac{a_n}{1} \le f_3 \le f_2,$$

$$- \ a_1 < 0, \text{ then } -1 \le a_1 \le f_2 \le \mathop{\mathrm{K}}_{n=1}^{\infty} \frac{a_n}{1} \le f_4 \le f_3 < 0.$$

Note that the bounds for positive partial numerators are much wider than the bounds for negative partial numerators.

Concerning cases (2) and (3), when $L \le a_n \le R \le 0$ and $a_{2n} \ge a_{2n+2} \ge a_\infty$ and $a_{2n+1} \le a_{2n+3} \le a_\infty$ and $L < -\frac{1}{4}$ and $a_\infty > -\frac{1}{4}$, the value $\mathrm{K}\frac{L}{1}$ does not exist and possibly not even $\mathrm{K}\frac{L,a_\infty}{1}$. We refer to the description of the algorithm of the Gauss continued fraction in Section 8.1.2 for an example of how to obtain a lower bound. The formulas for the upper bound remain valid.

4. ERROR ANALYSIS

It is well known [Gautschi 1967; Jones and Thron 1974] that a continued fraction approximant is best computed using the backward recurrence scheme

$$\begin{aligned} F_{n+1}^{(n)} &= \tilde{t}_n \\ F_k^{(n)} &= \frac{a_k}{1 + F_{k+1}^{(n)}}, \quad k = n, n-1, \ldots, 1 \\ f_n(\tilde{t}_n) &= F_1^{(n)} \end{aligned}$$

From this scheme and from Eq. (8), it follows that the error propagation in the computation of a continued fraction is best understood from a detailed error analysis of the linear fractional transformation s_n.

Let us assume that we have at our disposition a base β floating point environment of variable (adjustable) precision. When the working precision is p, then one unit in the last place (ULP) of a normalized floating point number corresponds to

$$\mathrm{ULP}(p) = \beta^{1-p}.$$

We often omit the explicit dependence on p (especially when it is clear what the current working precision is).

Let $\tilde{x}$ denote an approximation to the unknown quantity x, then the relative error r is defined by

$$r = \left| \frac{x - \tilde{x}}{x} \right|.$$

In interval arithmetic, one bounds the numerator (absolute error) of this expression, even though the quantity x remains unknown. In that case, it makes more sense to define

$$\tilde{r} = \left| \frac{x - \tilde{x}}{\tilde{x}} \right|.$$

It is easily seen that r and $\tilde{r}$ are related by

$$\frac{\tilde{r}}{1+\tilde{r}} \le r \le \frac{\tilde{r}}{1-\tilde{r}} \quad \text{and} \quad \frac{r}{1+r} \le \tilde{r} \le \frac{r}{1-r}, \tag{11}$$

whenever they are both smaller than one. So if we want that $r \leq \epsilon$ for a given ϵ, then it is sufficient to make sure that $\tilde{r} \leq \epsilon/(1+\epsilon)$.

We define the *accuracy* q of $\tilde{x}$ with respect to x, and the approximate accuracy $\tilde{q}$, by

$$q := -\log_\beta(r), \quad \tilde{q} := -\log_\beta(\tilde{r}). \tag{12}$$

These quantities correspond roughly to the number of significant β-digits of $\tilde{x}$.

Assume that we know an enclosing interval T_n for the tail t_n, that is, $t_n \in T_n$. Take as an approximation $\tilde{t}_n$ for t_n one of the endpoints of T_n, Then, we know that

$$\tilde{r}_n = \left|\frac{t_n - \tilde{t}_n}{\tilde{t}_n}\right| \leq \left|\frac{T_n}{\tilde{t}_n}\right| = \tilde{R}_n,$$

(where $|T_n|$ refers to the length of the interval), and because $\tilde{t}_n$ equals one of the endpoints of T_n, we also know the direction of the error, that is,

$$t_n = \tilde{t}_n(1 + \sigma_n \tilde{r}_n) \tag{13}$$

where $\sigma_n = \pm 1$ is known. Here, $\sigma_n = 1$ indicates the direction of larger absolute value (away from zero) and $\sigma_n = -1$ that of smaller absolute value (towards zero).

When computing $\tilde{t}_{n-1} \approx t_{n-1}$ from $\tilde{t}_n$ using the transformation s_n, we assume that the addition $1+\tilde{t}_n$ is performed exactly by adapting the working precision accordingly (this can be done by looking at the mantissa and exponent of $\tilde{t}_n$, but we omit the details). We then have

$$\tilde{t}_{n-1} = \frac{a_n(1+\alpha_n)}{1+\tilde{t}_n}(1+\delta_n)$$

where α_n is the relative error from computing a_n (which can, in general, not be represented exactly) and δ_n is the error from the division. Using Eq. (13), some computations yield that

$$\frac{t_{n-1} - \tilde{t}_{n-1}}{\tilde{t}_{n-1}} = -\frac{\dfrac{\tilde{t}_n \sigma_n \tilde{r}_n}{1+\tilde{t}_n}(1+\alpha_n)(1+\delta_n) + \alpha_n + \delta_n + \alpha_n\delta_n}{\left(1 + \dfrac{\tilde{t}_n \sigma_n \tilde{r}_n}{1+\tilde{t}_n}\right)(1+\alpha_n)(1+\delta_n)}. \tag{14}$$

This formula forms the basis of our entire algorithm. The sign of α_n and δ_n can be chosen in the case of *directional rounding*. We have to distinguish between two cases.

(1) If

$$\frac{\tilde{t}_n \sigma_n}{1+\tilde{t}_n} > 0,$$

then use rounding away from zero such that $\alpha_n > 0$ and $\delta_n > 0$. It is clear that in that case $\sigma_{n-1} = -1$. Furthermore, it is easily proved that the upper-bound $\tilde{R}_{n-1}$ for the relative error $\tilde{r}_{n-1}$ is given by the absolute value of (14), with $\sigma_n \tilde{r}_n$ replaced by $\tilde{R}_n$ (this follows essentially from the fact that the function $(a+bx)/(1+x)$ is increasing with x whenever $b > a > 0$).

(2) If

$$\frac{\tilde{t}_n \sigma_n}{1+\tilde{t}_n} < 0,$$

then use rounding towards zero such that $\alpha_n < 0$ and $\delta_n < 0$. In this case, $\sigma_{n-1} = 1$ and $\tilde{R}_{n-1}$ is given by the absolute value of (14) with $\sigma_n \tilde{r}_n$ replaced by $-\tilde{R}_n$. However, this only holds if $\tilde{R}_n |\tilde{t}_n/(1+\tilde{t}_n)| < 1$, of course. Otherwise, the error (14) can become unbounded.

Contrary to a fixed precision computation, where the values of α_n and δ_n are *given*, we *choose* those values by once more adjusting the working precision. Note that, in the first case, we may rewrite the upper-bound $\tilde{R}_{n-1}$ as

$$\tilde{R}_{n-1} = \tilde{R}''_{n-1}\left(1 + \frac{\alpha_n + \delta_n + \alpha_n\delta_n}{(1+\alpha_n)(1+\delta_n)} \cdot \frac{1}{\tilde{R}'_{n-1}}\right),$$

with

$$\tilde{R}'_{n-1} = \tilde{R}_n\left|\frac{\tilde{t}_n}{1+\tilde{t}_n}\right|, \quad \tilde{R}''_{n-1} = \frac{\tilde{R}'_{n-1}}{1+\tilde{R}'_{n-1}}.$$

The value $\tilde{R}''_{n-1}$ is an upper bound for the relative error in the *absence of rounding errors* (this can also be obtained from formula (14) by putting α_n and δ_n to zero, replacing $\sigma_n\tilde{r}_n$ by $\tilde{R}_n$ and taking the absolute value). So if we want to limit the loss of accuracy due to rounding errors to no more than λ digits, we should have that

$$1 \le 1 + \frac{\alpha_n + \delta_n + \alpha_n\delta_n}{(1+\alpha_n)(1+\delta_n)} \cdot \frac{1}{\tilde{R}'_{n-1}} \le \beta^\lambda. \tag{15}$$

If the precision to realize this is p_n, then we know that $|\delta_n| \le \mathrm{ULP}(p_n)$ because we use directional rounding. Furthermore, we demand that p_n is such that the partial numerators a_n are delivered with an error of at most $|\alpha_n| \le 2\mathrm{ULP}(p_n)$. Substituting these values into (15) gives a quadratic equation in $\mathrm{ULP}(p_n)$, from which we obtain the required precision p_n:

$$p_n \ge 1 - \log_\beta\left(\frac{1}{4}\sqrt{1 + \frac{8}{1-\tilde{R}'_{n-1}(\beta^\lambda - 1)}} - \frac{3}{4}\right).$$

This analysis can be repeated for the second case with only minor changes to the formulas. We find that in this case

$$p_n \ge 1 - \log_\beta\left(-\frac{1}{4}\sqrt{1 + \frac{8}{1+\tilde{R}'_{n-1}(\beta^\lambda - 1)}} + \frac{3}{4}\right).$$

If the errors are small, then $\tilde{R}''_{n-1} \approx \tilde{R}'_{n-1}$ and we thus have

$$\tilde{R}_0 \approx \tilde{R}_n \prod_{i=1}^{n}\left|\frac{\tilde{t}_i}{1+\tilde{t}_i}\right| \cdot \beta^{n\lambda}. \tag{16}$$

Note that this formula is obtained by ignoring second order error terms in the division. In this error estimate the factor $\tilde{R}_n$ is an upper bound of the error made by approximating the nth tail t_n by $\tilde{t}_n$. From the interval sequence theorem given in Cuyt et al. [2006] we learn that the factor $\prod_{i=1}^{n}|\tilde{t}_i/(1+\tilde{t}_i)|$ estimates by how much the initial error $\tilde{R}_n$ shrinks while you execute the backward recurrence to compute $f = t_0$. So the product $\tilde{R}_n\prod_{i=1}^{n}|\tilde{t}_i/(1+\tilde{t}_i)|$ is an estimate of the truncation error upperbound $\tilde{R}_0$. The factor $\beta^{n\lambda}$ bounds the roundoff error, which starts with β^λ and is magnified by the same factor in every step of the backward algorithm, hence the nth power. So if we want the total loss of accuracy due to roundoff errors to be limited to approximately m digits (m need not be integer, and in a large base β it better not), we simply choose $\lambda = m/n$.

5. ALGORITHM

We are now in a position to explain the algorithm. An important feature is that the error formulas derived in the previous section are used twice.

First, the computations are done in double precision under the assumption that we can separate roundoff error and truncation error as in formula (16). These double precision error computations are very accurate, although not guaranteed to be correct.

Next we compute the continued fraction in high precision (adapting the working precision as explained). Now every error is accounted for and the end result is validated.

The algorithm proceeds along the following steps, each of which is discussed in more detail as follows.

(1) Estimate the index n for which the modified approximant $f_n(\tilde{t}_n)$ has a relative error $r_0 = |t_0 - f_n(\tilde{t}_n)|/|t_0|$ not exceeding a given tolerance ϵ.
(2) Obtain a guaranteed enclosure for the corresponding tail t_n.
(3) Compute the expected accuracy of the approximant f_n in double precision.
(4) If the expected accuracy is too low, increase the index n and repeat the previous steps until a suitable n is found.
(5) Start computing f_n in double precision, using interval arithmetic to maintain a guaranteed enclosure.
(6) When the limits of machine precision are reached, continue the computations in high precision and start a running error analysis to obtain a guaranteed a posteriori error bound.
(7) Compare the a posteriori accuracy to the requested accuracy, restart if necessary.

The iterations (2)–(4) can be optimized to avoid duplicate calculations as much as possible. A restart in step (7) is only necessary in very difficult cases where the initial a priori estimate is inaccurate due to the restrictions of double precision. In the case of the continued fraction for the ratio of two Gauss hypergeometric functions, this only happens for z extremely close to 1 (see the example in Section 9).

5.1. A First Rough Guess for *n*

This rough estimate is not very critical, but a better estimate saves work in the rest of the computations.

For a requested relative error bound $R_0 \le \epsilon$, in other words $\tilde{R}_0 \le \epsilon/(1+\epsilon)$, compute the accuracy $\tilde{q}_0$ by (12). Then, add the number of digits m that are allowed to be lost to roundoff to obtain

$$\hat{q}_0 = \tilde{q}_0 + m.$$

For a limit 1-periodic fraction, that is, a continued fraction for which $\lim_{n\to\infty} a_n = a_\infty > -1/4$, we assume whenever $a_\infty \neq 0$ that for large n the tail t_n is approximately equal to $t_\infty = \mathrm{K}\frac{a_\infty}{1}$, given by (9). For a requested accuracy $\hat{q}_0$, we then get from formula (16) (assuming that the tail estimate $\tilde{t}_n$ has no correct digits and thus $\tilde{R}_n \approx 1$) that

$$n \approx \frac{\hat{q}_0}{\log_\beta |1 + 1/t_\infty|}.$$

If $a_\infty = 0$, then $1 + t_n \approx 1$ and $t_{n-1} = a_n/(1+t_n) \approx a_n$ so that the accuracy after n steps is approximately equal to

$$\hat{q}_0 \approx \sum_{i=1}^{n} \log_\beta \left| \frac{1 + a_{i+1}}{a_{i+1}} \right|.$$

Adding terms until the sum exceeds $\hat{q}_0$ gives the required value for n.

These two cases cover the continued fractions for the Gauss and confluent hypergeometric functions discussed in Section 8. For more complicated continued fractions, a different method for estimating n is needed. For limit 2-period fractions, formula (10) can be of use.

5.2. Tail Estimate and Enclosure

For the value of n just obtained, we need an interval that is guaranteed to contain the tail t_n. Several formulas are given in Section 3. These formulas cover most cases that occur in practice. However, it may be that a guaranteed enclosing interval cannot be found because n is not large enough (in many cases, the partial numerators a_n are only smooth functions of n when n is larger than some treshold, see Section 8 for examples). If this is the case, n is increased (which also increases the accuracy, so this is not a problem).

5.3. Expected Accuracy of $f_n(\check{t}_n)$

Taking one of the endpoints of the enclosing interval as an estimate $\tilde{t}_n$ for t_n and taking the width of the interval as an upper bound for the absolute error $|T_n| = \tilde{R}_n|\tilde{t}_n|$, we can apply formula (14) repeatedly to get an upper bound $\tilde{R}_0$ for the relative error. However, since the roundoff errors in $\tilde{t}_n$ have been accounted for, we put both α_n and δ_n equal to 0 (so, in fact, we compute $\tilde{R}_0''$). The expected accuracy is then

$$\tilde{q}_0'' = -\log_\beta \tilde{R}_0''.$$

If, in the course of these computations, it happens that $\tilde{R}_n|\tilde{t}_n/(1+\tilde{t}_n)| \geq 1$ while $\tilde{t}_n\sigma_n/(1+\tilde{t}_n) < 0$, we cannot continue (see the error analysis in the previous section). In this case, we simply put $\tilde{q}_0''$ equal to the accuracy at this index and proceed to Step 5.4 in the algorithm.

5.4. Increasing n

If the expected accuracy $\tilde{q}_0''$ of $f_n(\tilde{t}_n)$ is less than $\hat{q}_0$, we have to increase n. Since we have gained on average $\kappa = \tilde{q}_0''/n$ digits per linear fractional transformation, we increase n by $(\hat{q}_0 - \tilde{q}_0'')/\kappa$. There are two tacit assumptions here, which we should discuss.

First, there is the assumption that each linear fractional transformation increases the accuracy by the same amount (or that the final accuracy is a linear function of n). This is only the case when all $\tilde{t}_i$ are equal and the errors are small, as can be seen from formula (16).

The other assumption is that the values $\tilde{t}_i$ for $i \leq n$ are independent of the starting value $\tilde{t}_n$ (or that the same values are obtained when starting from a different n). This is obviously not true. However, when starting the recurrence from a certain n and corresponding $\tilde{t}_n$, and going back to $\tilde{t}_0$, the values $\tilde{t}_i$ more and more approach the actual tails t_i and the errors become smaller (i.e., accuracy increases).

Therefore, in practice it is better to disregard the fractional transforms with i close to n in the definition of κ, because in these transforms not enough accuracy has been built up yet. Instead, one should define κ as

$$\kappa = \frac{\tilde{q}_0'' - \tilde{q}_{n,n'}''}{n'},$$

where $\tilde{q}_{n,n'}''$ is some heuristically predetermined value and $n' < n$ is the index such that the accuracy gain between n and n' is at least $\tilde{q}_{n,n'}''$. The computations from n' to 0 do not have to be redone, because it is assumed that the tails in this part have converged sufficiently to their exact values.

Even if either of these assumptions is not satisfied, we are not facing a real problem, because we keep repeating 5.3 and 5.4 until a suitable value of n is found.

5.5. Double Precision Interval Computation

Since we want to limit the amount of work that needs to be done in multiprecision as much as possible, we start computing the modified approximant $f_n(\tilde{t}_n)$ in double

precision. The computation of the tail estimates $\tilde{t}_i$ is carried out in some implementation of double precision interval arithmetic, so that we have guaranteed enclosures. When the guaranteed accuracy of $\tilde{t}_i$ does not improve anymore from a certain index k on, then we have reached the limits of double precision interval arithmetic. The value of k and the corresponding accuracy $\tilde{q}_k$ of the tail estimate $\tilde{t}_k$ are stored for (possible) future use in Section 5.7.

5.6. Multiprecision Computation

We now start the computations in high precision, using the tail estimate $\tilde{t}_k$ at the index k just obtained. The working precision is adapted twice during the computation of each linear fractional transform. First, to incorporate exact addition and then to account for rounding errors, as explained in Section 4. It is clear that we take $\lambda = m/k$ here. We use formula (14) to compute the error estimate along with the continued fraction itself, using rounding away from zero or towards zero as explained.

5.7. Accuracy Check

In most cases, the accuracy is exactly what was requested and the algorithm ends here. However, in some exceptionally difficult cases (when some of the tails are very close to -1), a restart is needed. If the final accuracy $\tilde{q}_f < \tilde{q}_0$, we redo all the computations, but now requesting an accuracy of $\tilde{q}_k + (\tilde{q}_0 - \tilde{q}_f)$ at the index k.

In even more exceptional cases, the error has become so huge that the algorithm had to abort before reaching the top of the continued fraction. Since we have no idea what happens to the accuracy in the remaining iterations, we assume that it remains more or less constant. This means that we consider the missing accuracy to be equal to the requested final accuracy minus the achieved (possibly negative) accuracy at the aborting index. We then proceed as before. Examples to illustrate this are given in Section 9.

6. IMPLEMENTATION ISSUES

The previous section contains the core of our algorithm, but to turn this into a working program requires the discussion of several additional issues.

A working implementation of our algorithm that also incorporates the computations needed to evaluate (confluent) hypergeometric functions (see the next section), can be found at http://www.cfhblive.ua.ac.be/evaluate+. This is a C++ program that uses the `MpIeee` multiprecision floating point library [Verdonk et al. 2001a; 2001b; Backeljauw 2009]. this library supports computations in basis $\beta = 2^j$ or $\beta = 10^k$ for $j = 1, \ldots, 24$ and $k = 1, \ldots, 7$.

We list some of the implementation issues that are dealt with in this software.

(1) If the working precision needed for exact addition is too high (and thus too slow), we take a lower precision instead and include an additional error term.
(2) Error bound (14) is exact, that is, every error is accounted for. In practice, to speed up computations we overestimate second order error terms in the division by a constant amount as soon as the accuracy is high enough, instead of computing them exactly.
(3) The value of m (number of digits lost to roundoff) is chosen heuristically. When used to determine the working precision, as explained in Section 4, it is set to $1/128$. When used to increase the requested accuracy, as explained in Section 5.1, it is set to an even safer $1/8$. The latter is because $\tilde{q}_0''$ is merely an expected accuracy that satisfies $\tilde{q}_0'' \geq \hat{q}_0$. The correct $\tilde{q}_0$ is validated in Section 5.7.
(4) As an extra safety margin, the index n estimated as in Section 5.1 is increased by 10.

(5) The value of $\tilde{q}''_{n,n'}$ is set to 30 bits (approximately nine decimal digits). Again this is a heuristic value that works well in practice.
(6) It may happen that a partial numerator is zero (either because a_n is exactly zero, or because of underflow in double precision). It may also happen that $\tilde{t}_n = 0$ or $\tilde{t}_n = -1$ (because of roundoff). These cases are dealt with appropiately.
(7) The error computations that are done in double precision also suffer from roundoff errors themselves. If these accumulate too much, the double precision accuracy estimate becomes unreliable. We monitor these roundoff errors assuming worst-case behavior, and when, at a certain index n, they have become too large, the double precision computation must be aborted. We then assume that the accuracy remains more or less constant in the lower indices, and proceed immediately to the validated computation. The examples in Section 9 illustrate this case. Note that this is *not* the same as the case mentioned at the end of Section 5.3.
(8) When the algorithm succeeds, it returns an endpoint of the enclosing interval, and a relative error. Depending on the rounding mode requested by the user, we return this endpoint, the other endpoint or the midpoint of the corresponding interval. This requires some additional computations.

7. COMPUTING THE PARTIAL NUMERATORS

The implementation of the partial numerators a_n is of course different for each continued fraction and should be provided by the user. In this section, we explain the necessary steps in more detail.

The most important point is that *two implementations* are necessary: a double-precision implementation with guaranteed error bounds, and a high-precision implementation that delivers a_n for any n and any requested error and rounding direction.

The *double-precision implementation* is necessary in step (5) of the algorithm, when we start computing the continued fraction in double precision using some implementation of interval arithmetic, as explained in Section 5.5. It suffices that for any n the implementation returns an enclosing interval for a_n, or equivalently a value for a_n and a guaranteed error bound. This can be done using interval arithmetic, or using a classical error analysis as described below.

The *high-precision implementation* should return the value of a_n within the relative error α_n, which can be positive or negative (indicating the rounding mode), as explained in the error analysis in Section 4. These α_n are not known in advance, but determined at runtime. Obviously, variable precision arithmetic is required to achieve such implementation of a_n. A detailed error analysis (such as the one in Section 4) provides the precision required for the computation of a_n to yield a certain error α_n. We illustrate this for the case of the coefficient $a_{2k+1} = c_{2k+1}z$ in the continued fraction (4) for the ratio of two Gauss hypergeometric functions, when the parameter b is a positive integer n.

In the following error analysis, we assume that the simple integers n, k, $n-k$, $2k$ and $2k+1$ can be represented exactly. Of course, in the actual code it should be checked whether this is true and no overflow occurs. We also assume that the parameters a and c are both positive numbers. The analysis for the case where one or both is negative, is analogous. If we denote the computed coefficient by $\tilde{a}_{2k+1}$, it follows that

$$\begin{aligned}\tilde{a}_{2k+1} &= \frac{(a(1+\delta_a)+k)(1+\delta_+)(c(1+\delta_c)-(n-k))(1+\delta_-)(1+\delta_*)}{(c(1+\delta_c)+2k)(1+\delta_+)(c(1+\delta_c)+2k+1)(1+\delta_+)(1+\delta_*)} \\ &\quad \times z(1+\delta_z)(1+\delta_*)(1+\delta_/) \\ &= a_{2k+1}(1+\alpha_{2k+1})\end{aligned}$$

where each subscripted δ is bounded above by $\mathrm{ULP}(p_{2k+1,\alpha})/2$ and $p_{2k+1,\alpha}$ denotes the precision in which we compute a_{2k+1}.

From Chapter 3 of Higham [2004] and from Backeljauw et al. [2009], it follows that

$$|\alpha_{2k+1}| \leq \frac{(\nu(k)+12)\mathrm{ULP}(p_{2k+1,\alpha})}{2-(\nu(k)+12)\mathrm{ULP}(p_{2k+1,\alpha})} \tag{17}$$

where

$$\nu(k) = \left\lceil \frac{|c|+|n-k|}{|c-(n-k)|} \right\rceil$$

and $\lceil\cdot\rceil$ denotes upward rounding to integer. From (17), we can compute the precision $p_{2k+1,\alpha}$ that is needed to guarantee a relative error α_{2k+1} for a_{2k+1}. As explained in the paragraph following (15), the high precision implementation of a_{2k+1} is assumed to guarantee an upper bound for $|\alpha_{2k+1}|$ of at most $2\mathrm{ULP}(p_{2k+1})$ where p_{2k+1} is the precision in which the linear fractional transformation s_{2k+1} is computed. An implementation need not use directional rounding in the computation of a_{2k+1} but instead return one end of the enclosing interval $\tilde{a}_{2k+1}/(1 \pm |\alpha_{2k+1}|)$.

8. HYPERGEOMETRIC FUNCTIONS

In the introduction, we explained how the Gauss and the confluent hypergeometric function can be computed as a product of continued fractions whenever one of the numerator parameters is an integer. This is a nice application of the algorithm that we presented in the previous section. Of course, in order to deliver a validated result for the hypergeometric functions, the rounding errors in the product of the continued fractions also have to be taken into account. This can be done in a rather straightforward manner. A full analysis of this problem can be found in Backeljauw et al. [2009]. The only particularity that remains to be discussed here is how to obtain guaranteed enclosures for the tails.

8.1. Gauss Hypergeometric Function

From formulas (2)–(3) in the introduction, we see that the a_n are given by

$$a_n = -\frac{\left(a+\frac{n-1}{2}\right)\left(c-b+\frac{n-1}{2}\right)}{(c+n-1)(c+n)}z, \quad n \text{ odd},$$
$$a_n = -\frac{\left(b+\frac{n}{2}\right)\left(c-a+\frac{n}{2}\right)}{(c+n-1)(c+n)}z, \quad n \text{ even}.$$

A first thing to note is that $a_\infty = \lim_{n\to\infty} a_n = -z/4$. How the a_n approach this limit depends on the values of the parameters. For both even and odd n, one finds that

$$\frac{d}{dn}a_n = \frac{(-1)^n(1-2a+2b)n^2+O(n)}{4(c+n-1)^2(c+n)^2}z.$$

The coefficient of n^2 in this expression is zero if $a-b=\frac{1}{2}$. This obviously has a major impact on the behavior of the partial numerators, so we look at both possibilities separately.

8.1.1. First Case: $a-b=\frac{1}{2}$. Replacing b by $a-\frac{1}{2}$ in the definition of the partial numerators shows that, in this case, the formulas for odd and even indices are the same,

$$a_n = -\frac{\left(a+\frac{n-1}{2}\right)\left(c-a+\frac{n}{2}\right)}{(c+n-1)(c+n)}z,$$

while the derivative is given by

$$\frac{d}{dn}a_n = \frac{(c-2a+1)(2a-c)(2n+2c-1)}{4(c+n-1)^2(c+n)^2}z.$$

The sign of the derivative is constant for $n > 1/2 - c$, so for these n the partial numerators increase or decrease monotonically to the limit. This is not enough yet: we also need to make sure that the sign of the partial numerators themselves does not change. This is the case when n is greater than $(1-2a)$, $2(a-c)$ and $(1-c)$.

There is one more thing that needs to be checked. If $a_\infty < 0$, we also need to find out from which n on the partial numerators are greater than $-\frac{1}{4}$. This is the case for

$$n > \frac{1}{2}\left(1 - 2c + \sqrt{\frac{(1-4a+2c)^2 z - 1}{z-1}}\right).$$

Using all this knowledge we can now simply apply the appropriate formulas from Section 3. For negative a_∞ use case (2) from Section 3.1 or case (1) from Section 3.2. For positive a_∞ use cases (7) and (8) from Section 3.2.

8.1.2. Second Case: $a - b \neq \frac{1}{2}$. From the different signs of the coefficients of n^2 in the derivatives, we can immediately see that the limit is approached from both sides. Either the odd partial numerators increase and the even ones decrease, or vice-versa.

We now look at the sign of both the partial numerators and their derivatives. We omit the computations, which are tedious but straightforward, and simply state the result. The index has to be greater than the maximum of the numbers

$$(1-c), \quad (1-2a), \quad (1-2c+2b), \quad (-2b), \quad 2(a-c),$$
$$\frac{c + c^2 + 4ab - 4ac \pm \sqrt{(2a-c-1)(2a-c)(c-2b-1)(c-2b)}}{2a-2b-1} + 1,$$
$$\frac{c - c^2 + 4bc - 4ab \pm \sqrt{(2a-c-1)(2a-c)(c-2b-1)(c-2b)}}{2a-2b-1},$$

(the last two formulas correspond to the zeros of the derivatives), in order to guarantee no sign changes in a_n or its derivative.

If the partial numerators are positive, we can simply use cases 5 and 6 from Section 3.2.

For $a_\infty < 0$ we need to determine the index from which the partial numerators are greater than $-\frac{1}{4}$. These indices can be determined exactly, but the formulas look very complicated. Suffice it to say that, if z is very close to 1, and the values of a, b and c are rather large, the partial numerators only rise above $-\frac{1}{4}$ for an enormous index n. Since we do not want to calculate millions or even billions of iterations, a special algorithm is necessary for this case. As mentioned at the end of Section 3, the formulas for the upper bound are still valid, but to obtain the lower bound, we proceed as follows. This algorithm is based on the same kind of reasoning that lead to the different cases in Section 3.

(1) Let k be the index from which the partial numerators are greater than $-1/4$. Let d be equal to $(k-n)/10$, rounded down to the next even number.
(2) Calculate $\mathrm{K}\frac{a_k, a_\infty}{1}$, conservatively rounded down to underestimate the tail t_k.
(3) Let $k_0 = k$, and $k = k - d$.
(4) Calculate $\mathrm{K}\frac{a_k, a_{k_0-1}}{1}$, conservatively rounded down.
(5) If this value does not exist (divergent 2-periodic continued fraction), reduce d but keep it even. Then go back to step (3).
(6) Otherwise, the new lower bound is the minimum of the old bound and the value from step (4).

(7) Go back to step (3) if $k > n$, except if $d \leq 2$ and k is within integer range (since it will be quicker to just start with this tail rather than extending the algorithm here at such a slow rate d).

For other cases, a collection of formulas from Section 3 for different scenarios can be used.

8.2. Confluent Hypergeometric Function

In this case, it follows from (6)–(7) that

$$a_n = -\frac{b - a + (n-1)/2}{(b+n-1)(b+n)}z, \quad n \text{ odd},$$
$$a_n = \frac{a + n/2}{(b+n-1)(b+n)}z, \quad n \text{ even},$$

and thus $a_\infty = 0$. For large enough n, the odd and even partial numerators alternate in sign. Monotic convergence starts when n is larger than both

$$1 + 2a - 2b + \sqrt{(2a-b)(1+2a-b)}$$

and

$$-2a + \sqrt{(2a-b)(1+2a-b)},$$

obtained from the zeros of the derivative of a_n. Since a is assumed to be integer, the expression under the square root can never be negative. Furthermore, it can be checked that both odd and even partial numerators do not change sign if n is larger than both expressions above. In order to use case 9 from Section 3.2, we need to know the index n for which the partial numerators are certainly greater than -1. Some computations yield that this occurs (for both odd and even numerators) when

$$n > \frac{1}{4}(2 - 4b + |z| + \sqrt{4 - 4z - 16az + 8bz + z^2}).$$

If the expression under the square root is negative, then the partial numerators are greater than -1 for all n.

9. EXAMPLES

We conclude this article with some examples to illustrate the algorithm. Without loss of generality, all examples are done in base $\beta = 10$.

Note that the last two examples in this section do not illustrate the computation of the Gauss hypergeometric function itself, but only the Gauss continued fraction (4). The computation of the function as a product of continued fractions (as explained in the introduction) is detailed in Sections 2 and 8 of Backeljauw et al. [2009]. The only additional difficulty is distributing the requested accuracy over the individual factors in the product.

Although the examples in this section only illustrate the continued fraction for the Gauss hypergeometric function, it is clear that the case of the confluent hypergeometric function is similar.

9.1. An Easy Example

As a first example, we compute ${}_2\mathrm{F}_1(\frac{1}{2}, 1; \frac{3}{2}; -1)$, which equals $\arctan 1 = \frac{\pi}{4}$. We do this by calculating the Gauss fraction for $a = c = \frac{1}{2}$, $b = 0$ and $z = -1$, with an extra inversion afterwards. We take $\epsilon = 10^{-10000}$, which corresponds to approximately 10000 decimal digits.

The rough guess for n from Section 5.1 yields (with an extra safety margin of 10) an index $n = 13072$. The double precision runtime error analysis in Section 5.3 (detailed in Section 4) predicts that we can reach an accuracy of 33274.88 bits by starting with the tail $0.2071067814\ldots$, which is an underestimate ($\sigma_n = 1$) of t_n with an accuracy of 31.89147 bits (obtained from $|T_n|$). Since 10000 decimal digits correspond to 33219.28 bits, we have almost 56 bits of safety margin.

We then start computing the fraction in double precision interval arithmetic until we reach $n = 13044$, with an overestimating tail of $0.20710678144629\ldots$. In the process, we have improved the tail estimate accuracy to at least 48 bits.

Continuing the computations in high precision yields the result

$$f_0(\tilde{t}_n) = 0.7853981633974483(...)3814093919_{90740773\ldots}$$

which we know to be an overestimate because $\sigma_0 = -1$. The corresponding error is bounded by 0.54717 ULP on the 10000th digit. We return the middle of the interval, so we make a correction of half this error, which yields:

$$0.7853981633974483(...)3814093919_{6338206\ldots}$$

Now let's compare with the actual value of $\frac{\pi}{4}$:

$$0.7853981633974483(...)3814093919_{641680699\ldots}$$

The only thing left to check is whether or not our algorithm has been efficient. Was this number of iterations really necessary, and haven't we lost too much accuracy because of rounding errors? Well, here is the accurate result of doing 13044 iterations with the same starting tail in Mathematica:

$$f_0(\tilde{t}_n) = 0.7853981633974483(...)3814093919_{90723990\ldots}$$

9.2. Example with One Restart

We now look at a calculation that is considerably more difficult. With all the extra precautions listed in Section 6, a restart occurs only in a very limited number of situations. We calculate the Gauss continued fraction with

$$a = -\sqrt{640003}$$
$$b = \sqrt{40002}$$
$$c = \sqrt{10001}$$
$$z = \frac{\sqrt{99}}{10}$$

The square roots are only introduced to avoid having round numbers. We only request 1000 digits this time (3322 bits), or $\epsilon = 10^{-1000}$.

The educated guess for the number of iterations is $n = 16244$. Getting a tail estimate as explained in Section 5.2 is proving to be more difficult than last time. We obtain the interval $[-0.5173271\ldots, -0.4891708\ldots]$, which guarantees only about 4 bits of accuracy.

The estimated accuracy increases steadily, up to about 7526 bits around $n = 600$, but then stagnates and starts decreasing. At around $n = 500$, the estimated accuracy has decreased to 7510 bits and continues to go down. At $n = 427$, we have an estimated accuracy of 7484 bits, but the algorithm has detected that by now the error estimates in double precision have lost too much accuracy to be of any further use. As explained in Section 6, item (7), we assume that the accuracy remains constant from now on and proceed immediately to the validated computation.

We therefore compute the fraction in double precision interval arithmetic, as explained in Section 5.5. At $n = 3488$, we reach the limits of double precision with a new tail of

$$-0.535727099\ldots,$$

accurate up to at least 45 bits.

From here on, we switch to high precision. As before, the accuracy increases up to around $n = 600$, where it reaches 3363 bits. Then it starts decreasing. Contrary to what we hoped, this loss does not stop and we end up with only $\tilde{q}_f = 2792.10$ bits. A restart is needed.

The target is now to get at least 575 bits at $n = 3488$, which is 530 bits more than with the tail estimate at the last attempt (the difference between 3322 and 2792, as explained in Sections 5.5 and 5.7). Using again the rough guess from Section 5.1 gives a new starting index $n = 3488 + 2827$. The double precision error analysis shows that we get an accuracy of 1384 bits at $n = 3488$, which is certainly more than the 575 bits that we needed.

We start the double precision interval computations at $n = 3488 + 2827$ and this time we reach the limits of double precision at $n = 4399$, the new starting index for the high-precision computation. We obtain the following result:

$$8.947978583453710(...)659278566391_{688349088}\ldots.$$

The maximum error is estimated to be 0.4145328 ULP on the 1000th digit, so the midpoint value is

$$8.947978583453710(...)659278566391_{89561553}\ldots.$$

The correct value is

$$8.947978583453710(...)659278566391_{757938216}\ldots.$$

9.3. Diabolical

Now let us take z really close to one:

$$a = \sqrt{100003}$$
$$b = \sqrt{100001}$$
$$c = -\sqrt{8000002}$$
$$z = 1 - 2^{-40}.$$

We request the same accuracy as in the previous example. This turns out to be quite a stress test. First of all, due to the proximity of the partial numerators to $-1/4$, a theoretical tail estimate as in Section 5.2 can only be made for $n = 24110809$.

According to 5.3, this is expected to yield a final accuracy of $\tilde{q}_0'' = 98307$ bits, of course without certainty. The double precision estimates seem to degrade too much at $n = 2522$. Again, we refer to the explanation in Section 6, step (7).

We compute about 24 million iterations in double precision interval arithmetic up to $n = 5479$, when the limits of machine precision are reached and we switch to high precision computation.

The accuracy increases to 3362 bits around $n = 2800$, but then starts going down faster and faster. Around $n = 1000$, the tails are close to -1.5 so the relative error triples at every iteration, or a loss of about 1.6 bits. At lower n, the tails get even closer to -1 and we lose 3 bits per iteration or more. Finally, the accuracy goes below 0 at $n = 162$.

A restart is needed to get 3322 bits (1000 decimal digits) of extra accuracy at $n = 5479$. The new index for the high precision computation becomes $n = 6260$. Now the accuracy increases to 6694 bits. Just like the first time, we start losing accuracy from here. At $n = 162$, the accuracy has dropped to 3329 bits. This is still more than the requested final accuracy but the losses continue. We reach the top of the fraction $f_0(\tilde{t}_n)$ with only $\tilde{q}_f = 2822.73$ bits left.

At least on this second attempt we managed to do the entire continued fraction, so now we know exactly how much accuracy is missing. After another restart, a new starting point for the high precision computation is $n = 6314$. The calculation is started again, the accuracy climbs, then goes down again, and finally reaches about 3330 bits. The final result (interval midpoint) is:

$$-0.11180394391500178784(...)81551034637840259_{044053\ldots}.$$

The error bound is 0.000247 ULP on the 1000th digit. The actual correct value is:

$$-0.11180394391500178784(...)81551034637840259_{044052349\ldots}.$$

9.4. Comparison with Other Software

In order to convince the reader that these examples are really hard and that a lot of existing software is not capable to deal with them, we have carried out the same numerical examples in the software packages listed for hypergeometric function evaluation in the software index of Olver et al. [2010]:

—the Gnu Scientific Library version 1.13,
—Thompson's Atlas for computing mathematical functions [Thompson 1997],
—the multiprecision Cephes library (see Netlib),
—Maple 14.

The first two libraries only provide double precision implementations, so we only expect double precision evaluations in return. The Cephes library provides implementations at several (higher than double) precisions to verify a double precision special function evaluation. We measure the duration of all computations using the C-function `time`. In Maple, we ask for the same high number of digits as in the examples using its decimal multiprecision capabilities. We also measure the elapsed time in the successful cases and bound it to 24 hours in the unsuccessful cases. All runs were performed on a machine equipped with an Intel T9300 CPU (2.5GHz) with 4GB of memory, using only one single core. Only the operating system was running in the background.

GSL returns the first example in less than a second but fails in the second and third example where it returns an error.

The Thompson library returns completely erroneous results in the second and third case and is unable to produce double precision output for the first example within 12 hours of runtime.

The multiprecision Cephes library gets the first example wrong, returns 28 reliable decimal digits for the second example and an error for the third example.

Maple returns the first function evaluation within 48 seconds, is unable to produce an answer for the second case within 24 hours and returns an undefined for the third case. Note that a computer algebra system may rewrite and simplify a requested function evaluation symbolically before starting the actual computation that gives them an unfair advantage over floating-point libraries.

Our implementation compares very favorably with this. We return high precision validated results in all three cases, not merely approximations, and this in just over 6 minutes for the diabolical example. When requiring only double-precision results (say $\epsilon = 10^{-15}$), then the first example also terminates in less than a second.

REFERENCES

ABRAMOWITZ, M. AND STEGUN, I. A. 1964. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Applied Mathematics Series, vol. 55. National Bureau of Standards, Washington, D.C.

BACKELJAUW, F. 2009. A library for radix-independent multiprecision IEEE-compliant floating-point arithmetic. Tech. rep. 2009-01, Department of Mathematics and Computer Science, Universiteit Antwerpen.

BACKELJAUW, F., BECUWE, S., CUYT, A., AND VAN DEUN, J. 2009. Validated evaluation of special mathematical functions. Tech. rep. 2009–04, Department of Mathematics and Computer Science, Universiteit Antwerpen.

CUYT, A., PETERSEN, V. B., VERDONK, B., WAADELAND, H., AND JONES, W. B. 2008. *Handbook of Continued Fractions for Special Functions*. Springer, New York.

CUYT, A., VERDONK, B., AND WAADELAND, H. 2006. Efficient and reliable multiprecision implementation of elementary and special functions. *SIAM J. Sci. Comput. 28,* 4, 1437–1462 (electronic).

GAUTSCHI, W. 1967. Computational aspects of three-term recurrence relations. *SIAM Rev. 9*, 24–82.

HIGHAM, N. J. 2004. The numerical stability of barycentric Lagrange interpolation. *IMA J. Numer. Anal. 24,* 4, 547–556.

JONES, W. B. AND THRON, W. J. 1974. Numerical stability in evaluating continued fractions. *Math. Comp. 28,* 127, 795–810.

JONES, W. B. AND THRON, W. J. 1980. *Continued fractions*. Encyclopedia of Mathematics and its Applications, vol. 11. Addison-Wesley, Reading, MA.

KEARFOTT, R. B. 1996. *Rigorous Global Search: Continuous Problems*. Nonconvex Optimization and its Applications, vol. 13. Kluwer Academic Publishers, Dordrecht.

LORENTZEN, L. 1992. Computations of hypergeometric functions by means of continued fractions. In *Computational and Applied Mathematics, I*, C. Brezinski and U. Kulish, Eds., Elsevier Science Publishers, North Holland, 305–314.

LORENTZEN, L. 2003. A priori truncation error bounds for continued fractions. *Rocky Mountain J. Math. 33,* 2, 409–474.

LORENTZEN, L. AND WAADELAND, H. 1992. *Continued Fractions with Applications*. Studies in Computational Mathematics, vol. 3. North-Holland Publishing Co., Amsterdam.

LOZIER, D. W. AND OLVER, F. W. J. 1994. Numerical evaluation of special functions. In *Mathematics of Computation 1943–1993: A Half-Century of Computational Mathematics*. Proceedings of Symposia in Applied Mathematics, vol. 48. American Mathematical Society, Providence. 79–125.

OLVER, F. W., LOZIER, D. W., BOISVERT, R. F., AND CLARK, C. W. 2010. *NIST Handbook of Mathematical Functions*. Cambridge University Press, Cambridge, UK. Software Index: http://dlmf.nist.gov/software.

PINCHERLE, S. 1894. Delle funzioni ipergeometriche e di varie questioni ad esse attinenti. *Giorn. Mat. Battaglini 32*, 209–291.

THOMPSON, W. 1997. *Atlas for Computing Mathematical Functions*. Wiley.

VERDONK, B., CUYT, A., AND VERSCHAEREN, D. 2001a. A precision- and range-independent tool for testing floating-point arithmetic. II: Conversions. *ACM Trans. Math. Softw. 27,* 1, 119–140.

VERDONK, B., CUYT, A., AND VERSCHAEREN, D. 2001b. A precision- and range-independent tool for testing floating-point arithmetric. I: Basic operations, square root, and remainder. *ACM Trans. Math. Softw. 27,* 1, 92–118.

Received July 2009; revised May 2010 and March 2011; accepted May 2011