NumGfun: a Package for Numerical and Analytic Computation with D-finite Functions

Marc Mezzarobba

NUMGFUN: A PACKAGE FOR NUMERICAL AND ANALYTIC COMPUTATION WITH D-FINITE FUNCTIONS

MARC MEZZAROBBA

Abstract. This article describes the implementation in the software package NumGfun of classical algorithms that operate on solutions of linear differential equations or recurrence relations with polynomial coefficients, including what seems to be the first general implementation of the fast high-precision numerical evaluation algorithms of Chudnovsky & Chudnovsky. In some cases, our descriptions contain small improvements over existing algorithms. We also provide references to relevant ideas not currently used in NumGfun.

1. Introduction

Support for computing with *D-finite functions*, that is, solutions of linear differential equations with polynomial coefficients, has become a common feature of computer algebra systems. For instance, Mathematica now provides a data structure called `DifferentialRoot` to represent arbitrary D-finite functions by differential equations they satisfy and initial values. Maple's `DESol` is similar but more limited.

An important source of such general D-finite functions is combinatorics, due to the fact that many combinatorial structures have D-finite generating functions. Moreover, powerful methods allow to get from a combinatorial description of a class of objects to a system of differential equations that "count" these objects, and then to extract precise asymptotic information from these equations, even when no explicit counting formula is available [15, 29]. A second major application of D-finiteness is concerned with special functions. Indeed, many classical functions of mathematical physics are D-finite (often by virtue of being defined as "interesting" solutions of simple differential equations), which allows to treat them uniformly in algorithms. This is exploited by the *Encyclopedia of Special Functions*[1] [25] and its successor under development, the *Dynamic Dictionary of Mathematical Functions*[2], an interactive computer-generated handbook of special functions.

These applications require at some point the ability to perform "analytic" computations with D-finite functions, starting with their numerical evaluation. Relevant algorithms exist in the literature. In particular, D-finite functions may be computed with an absolute error bounded by 2^{-n} in $n \log^{O(1)} n$ bit operations—that is, in softly linear time in the size of the result written in fixed-point notation—at any point of their Riemann surfaces [12], all necessary error bounds also being computed from the differential equation and initial values [30]. However, most of these algorithms have remained theoretical (as reported for instance in [13, §9.2.1]). The ability of current computer algebra systems to work with D-finite functions is

[1] `http://algo.inria.fr/esf/`
[2] `http://ddmf.msr-inria.inria.fr/`

(mostly) limited to symbolic manipulations, and the above-mentioned fast evaluation algorithm has served as a recipe to write numerical evaluation routines for specific functions rather than as an algorithm for the complete class of D-finite functions.

This article introduces NumGfun, a Maple package that attempts to fill this gap, and contains, among other things, a general implementation of said algorithm. NumGfun is distributed as a subpackage of gfun[3] [28], under the GNU LGPL. Note that it comes with help pages: the goal of the present article is not to take the place of user documentation, but rather to describe the features and implementation of the package, with supporting examples, while giving as much relevant information as possible to people wishing to develop similar software. The following examples illustrate typical uses of NumGfun, first to compute a remote term from a combinatorial sequence, then to evaluate a Heun function to high precision near one of its singularities.

Example 1. The Motzkin number M_n is the number of ways of drawing non-intersecting chords between n points placed on a circle. Motzkin numbers satisfy $(n+4)M_{n+2} = 3(n+1)M_n + (2n+5)M_{n+1}$. Using NumGfun, the command

```
nth_term({(n+4)*M(n+2)=3*(n+1)*M(n)+(2*n+5)*M(n+1),
                                    M(0)=1,M(1)=1},M(n),k)
```

computes $M_{10^5} = 6187\ldots7713 \simeq 10^{47\,706}$ in[4] 4.7 s and $M_{10^6} = 2635\ldots9151 \simeq 10^{477\,112}$ in 1 min. Naïvely unrolling the recurrence (using Maple) takes 10.7 s for M_{10^5}, and 41 s for $M_{2.10^5}$. The large overhead of `nth_term` for small indices comes from the fact that it unnecessarily computes a huge denominator (see §2).

Example 2. The double confluent Heun function $U_{\alpha,\beta,\gamma,\delta}$ satisfies

$$(z^2-1)^3U''(z) + (2z^5 - 4z^3 - \alpha z^4 + 2z + \alpha)U'(z) + (\beta z^2 + (2\alpha+\gamma)z + \delta)U(z) = 0, \quad U(0) = 1,\ U'(0) = 0.$$

The command

```
evaldiffeq(eq,y(z),-0.99,1000)
```

(where `eq` denotes this differential equation) yields

$$U_{1,\frac{1}{3},\frac{1}{2},3}(-0.99) \approx 4.67754...(990\text{ digits})...05725$$

in 50 s.

1.1. **Related work.** Most algorithms implemented in NumGfun originate in work of Chudnovsky & Chudnovsky and of van der Hoeven. Perhaps the most central of these is the "bit burst" numerical evaluation method [12], that generalizes earlier work of Brent [6] for specific elementary functions. It belongs to the family of binary splitting algorithms for D-finite series, hints at the existence of which go back to [2, §178]. Better known (thanks among others to [20]) binary splitting algorithms can be seen as special cases of the bit burst algorithm. One reason why, unlike these special cases, the bit burst algorithm was not used in practice is that in [12], none of the error control needed to ensure the accuracy of the computed result is part of the algorithm. The same algorithm was published independently

[3] `http://algo.inria.fr/libraries/papers/gfun.html`

[4] All timings reported in this article were obtained with the following configuration: Intel T7250 CPU at 2 GHz, 1 GB of RAM, Linux 2.6.32, Maple 13, GMP 4.2.1.

by van der Hoeven [30], whose version addresses this issue, thus giving a full-fledged evaluation algorithm for the class of D-finite functions, as opposed to a method to compute any D-finite function given certain bounds depending on the function.

All these algorithms extend to the computation of limits of D-finite functions at singularities of their defining equation. The case of regular singular points was treated both in [11, 12], and (more completely) in [31]. The more difficult case of irregular singular points appears in [33]. We refer to [4, §12.7], [33, §1] for more history and context.

On the implementation side, routines based on binary splitting for the evaluation of various elementary and special functions are used in general-purpose libraries such as CLN [19] and MPFR [16, 22]. Binary splitting of fast converging series is also the preferred algorithm of software dedicated to the high-precision computation of mathematical constants on standard PCs, including the current record holder for π [3]. Finally, even the impressive range of built-in functions of computer algebra systems is not always sufficient for applications. Work on the implementation of classes of "less common" special functions that overlap those considered in NumGfun include [1, 14].

1.2. **Contribution.** The main contribution presented in this article is NumGfun itself. We recall or give pointers to the algorithms it uses, and discuss some implementation issues. Some of these descriptions contain small improvements or details that do not seem to have appeared elsewhere. Specifically: (i) we improve the complexity analysis of the bit-burst algorithm by a factor $\log\log n$; (ii) we point out that Poole's method to construct solutions of differential equations at regular singular points can be described in a compact way in the language of noncommutative polynomials; this also leads to an improvement of the complexity of evaluating D-finite functions at singular points; and (iii) we describe in some detail the practical computation of all the bounds needed to obtain a provably correct result.

1.3. **What NumGfun is not.** Despite sharing some of the algorithms used to compute mathematical constants to tens of billions of digits, our code aims to cover as much as possible of the class of D-finite functions, not to break records. Also, it is limited to "convergent" methods: algorithms based on asymptotic expansions, including summation to the least term and resummation of divergent power series, are currently out of scope.

1.4. **Terminology.** Like the rest of gfun, NumGfun works with *D-finite functions* and *P-recursive sequences*. We recall only basic definitions here; see [29, §6.4] for further properties. A formal power series $y \in \mathbb{C}[[z]]$ is *D-finite* over $K \subseteq \mathbb{C}$ if it solves a non-trivial linear differential equation

$$y^{(r)}(z) + a_{r-1}(z)\, y^{(r-1)}(z) + \cdots + a_0(z)\, y(z) = 0 \tag{1}$$

with coefficients $a_k \in K(z)$. The same definition applies to analytic functions. A sequence $u \in \mathbb{C}^{\mathbb{N}}$ is *P-recursive* over K if it satisfies a non-trivial linear recurrence relation

$$u_{n+s} + b_{s-1}(n)\, u_{n+s-1} + \cdots + b_0(n)\, u_n = 0,\ b_k \in K(n). \tag{2}$$

A sequence $(u_n)_{n\in\mathbb{N}}$ is P-recursive if and only if it generating series $\sum_{n\in\mathbb{N}} u_n z^n$ is D-finite.

The poles of the coefficients a_k of (1) are its *singular points*; nonsingular points are called *ordinary*. In gfun, a D-finite function is represented by a differential equation of the form (1) and initial values at the origin, which we assume to be an ordinary point. Similarly, P-recursive sequences are encoded by a recurrence relation plus initial values, as in Ex. 1 above. If $y(z) = \sum_{k=0}^{\infty} y_k z^k$, we let $y_{;n}(z) = \sum_{k=0}^{n-1} y_k z^k$ and $y_{n;}(z) = \sum_{k=n}^{\infty} y_k z^k$.

The *height* of an object is the maximum bit-size of the integers appearing in its representation: the height of a rational number p/q is $\max(\lceil \log p \rceil, \lceil \log q \rceil)$, and that of a complex number (we assume that elements of number fields $\mathbb{Q}(\zeta)$ are represented as $\sum_i x_i \zeta^i / d$ with $x_i, d \in \mathbb{Z}$), polynomial, matrix, or combination thereof with rational coefficients is the maximum height of its coefficients. We assume that the bit complexity $M(n)$ of n-bit integer multiplication satisfies $M(n) = n(\log n)^{O(1)}$, $M(n) = \Omega(n \log n)$, and $M(n+m) \geq M(n) + M(m)$, and that $s \times s$ matrices can be multiplied in $O(s^\omega)$ operations in their coefficient ring.

2. Binary splitting

"Unrolling" a recurrence relation of the form (2) to compute $u_0, \dots, u_N$ takes $\Theta(N^2 M(\log N))$ bit operations, which is almost linear in the total size of $u_0, \dots, u_N$, but quadratic in that of u_N. The binary splitting algorithm computes a single term u_N in essentially linear time, as follows: (2) is first reduced to a matrix recurrence of the first order with a single common denominator:

$$q(n) U_{n+1} = B(n) U_n, \quad B(n) \in \mathbb{Z}[n]^{s \times s}, q(n) \in \mathbb{Z}[n], \tag{3}$$

so that $U_N = P(0,N)\, U_0 / \big(\prod_{i=0}^{N-1} q(i)\big)$, where

$$P(j,i) = B(j-1) \cdots B(i+1) B(i).$$

One then computes $P(0,N)$ recursively as

$$P(0,N) = P(\lfloor N/2 \rfloor, N) P(0, \lfloor N/2 \rfloor),$$

and the denominator $\prod_{i=0}^{N-1} q(i)$ in a similar fashion (but separately, in order to avoid expensive gcd computations).

The idea of using product trees to make the most of fast multiplication dates back at least to the seventies [4, §12.7]. The general statement below is essentially from [12, Theorem 2.2], except that the authors do not explicitly take into account the cost of evaluating the polynomials at the leaves of the product tree. Since the height of u_N may be as large as $\Omega((N+h)\log N)$, this result is optimal with respect to h and N, up to logarithmic factors.

Theorem 1 (Chudnovsky, Chudnovsky)**.** *Let u be a P-recursive sequence over $\mathbb{Q}(i)$, defined by (2). Assume that the coefficients $b_k(n)$ of (2) have no poles in $\mathbb{N}$. Let d, h denote bounds on their degrees (of numerators and denominators) and heights, and d', h' corresponding bounds for the coefficients of $B(n)$ and $q(n)$ in (3). Assuming $N \gg s, d$, the binary splitting algorithm outlined above computes one term u_N of u in*

$$O(s^\omega M(n(h' + d' \log N)) \log(Nh')),$$

that is,

$$O(s^\omega M(sdn(h + \log N)) \log(Nh)),$$

bit operations.

Proof sketch. Write $H = O(h' + d' \log N)$. Computing the product tree $P(0, N)$ takes $O(s^\omega M(NH) \log N)$ bit operations [12, §2] (see also Prop. 1 below), and the evaluation of each leaf $B(i)$ may be done in $O(M(H) \log d')$ operations [5, §3.3]. This gives u_N as a fraction that is simplified in $O(M(NH) \log(NH))$ operations [8, §1.6].

Now consider how (2) is rewritten into (3). With coefficients in $\mathbb{Z}[i]$ rather than $\mathbb{Q}(i)$, the $b_k(n)$ have height $h'' \leq (d+1)h$. To get $B(n)$ and $q(n)$, it remains to reduce to common denominator the whole equation; hence $d' \leq sd$ and $h' \leq s(h'' + \log s + \log d)$. These two conversion steps take $O(M(sdh \log^2 d))$ and $O(M(d'h' \log s))$ operations respectively, using product trees. Using the assumption $d, s = o(N)$ to write $H = O(sd(h + \log N))$ and get rid of some terms, the total complexity simplifies as stated. □

The same algorithm works over any algebraic number field instead of $\mathbb{Q}(i)$. This is useful for evaluating D-finite functions "at singularities" (see §5). More generally, similar complexity results hold for product tree computations in any torsion-free $\mathbb{Z}$-algebra (or $\mathbb{Q}$-algebra A: we then write $A = \mathbb{Q} \otimes_{\mathbb{Z}} A'$ for some $\mathbb{Z}$-algebra A' and multiply in $\mathbb{Z} \times A'$), keeping in mind that, without basis choice, the height of an element is defined only up to some additive constant.

2.1. **Implementation.** NumGfun contains two implementations of the binary splitting algorithm, one written in Maple, the other one in C on top of the `mpz` layer of GMP [18]. The C code is still at an early stage and was not used in the preparation of the examples of this article. Two user-level functions, `nth_term` and `fnth_term`, evaluate P-recursive sequences by binary splitting (`fnth_term` replaces the final gcd by a less expensive division and returns a floating-point result). Additionally, `gfun[rectoproc]`, which takes as input a recurrence and outputs a procedure that evaluates its solution, automatically calls the binary splitting code when relevant. Examples 1 and 3 illustrate the use of these functions on integer sequences and convergent series respectively.

Example 3. [7, §6.10] Repeated integration by parts of the integral representation of Γ yields for $0 < \mathrm{Re}(z) < 1$

$$\Gamma(z) = \sum_{n=0}^{\infty} \frac{e^{-t} t^{n+z}}{z(z+1)\cdots(z+n)} + \int_t^{\infty} e^{-u} u^{z-1} \mathrm{d}u.$$

Taking $t = 29^3$, it follows that the sum $\sum_{n=0}^{65000} u_n$, where $u_0 = 87e^{-t}$ and $(3n+4)u_{n+1} = 3tu_n$, is within 10^{-10000} of $\Gamma(1/3)$, whence

$$\Gamma(1/3) \simeq 2.67893\ldots(9990 \text{ digits})\ldots 99978.$$

This computation takes 1.7 s.

2.2. **Constant-factor improvements.** Several tricks permit to improve the constant hidden in the $O(\cdot)$ of Theorem 1, by making the computation at each node of the product tree less expensive. We consider two models of computation.

In the *FFT model*, we assume that the complexity of long multiplication decomposes as $M(n) = 3F(2n) + O(n)$, where $F(n)$ is the cost of computing a discrete Fourier transform (or a linear transformation made of a combination of DFTs) of size n. The most important technique to speed up products of integer polynomial or integer matrices is known "FFT caching and addition" [4], "computing in the

FFT mode" [30], or "FFT invariance". All major FFT-based integer multiplication algorithms adapt to reduce one multiplication of integer matrices of height n to a number of multiplications of matrices with smaller entries, and ultimately to $O(n)$ multiplications of matrices of height $O(1)$. This allows to multiply integer matrices of size s and height n in $O(s^2M(n)+s^\omega n)$ bit operations. Variants based on multiplying matrices of hardware floating-point numbers (encoding either real numbers or modular integers) are likely to be the most efficient in practice for recurrences of high order.

A second improvement ("FFT doubling", attributed to R. Kramer [4, §12.8]) is specific to the computation of product trees. The observation is that, at an internal node where operands of size n get multiplied using three FFTs of size $2n$, every second coefficient of the two direct DFTs is already known from the level below.

FFT-based techniques are not suited to implementation in Maple, and none of them is currently used in NumGfun.

The second model is *black-box multiplication*. There, we may use fast multiplication formulae that trade large integer multiplications for additions and multiplications by constants. The most obvious example is that the products in $\mathbb{Q}(i)$ may be done in four integer multiplications using Karatsuba's formula instead of five with the naive algorithm. In general, elements of height h of an algebraic number field of degree d may be multiplied in $2dM(h)+O(h)$ bit operations using the Toom-Cook algorithm.

The same idea applies to the matrix multiplications. Most classical matrix multiplication formulae such as Strassen's are so-called bilinear algorithms. Since we are working over a commutative ring, we may use more general *quadratic* algorithms [9, §14.1]. In particular, for all s, Waksman's algorithm [36] multiplies $s \times s$ matrices over a commutative ring R using $s^2\lceil s/2\rceil + (2s-1)\lfloor s/2\rfloor$ multiplications in R, and Makarov's [24] multiplies 3×3 matrices in 22 scalar multiplications. These formulas alone or as the base case of a Strassen scheme achieve what seems to be the best known multiplication count for matrices of size up to 20.

Remark 1. Note that the previous techniques save time only for dense objects. In particular, in the case of recurrences of the form (3), one should not use the "fast" matrix multiplication formulae in the few bottom levels of the tree, since the matrices at the leaves are companion.

Proposition 1. *Let d' and h' denote bounds on the degrees and heights (respectively) of $B(n)$ and $q(n)$ in Eq. (3). As $N, h' \to \infty$ (s and d' being fixed), the number of bit operations needed to compute the product tree $P(0,N)$ is at most $\big(C+o(1)\big)M\big(N(h'+d'\log N)\log(Nh')\big)$, with $C=(2s^2+1)/6$ in the FFT model, and $C=(3\,\mathrm{MM}(s)+1)/4$ in the black-box model. Here $\mathrm{MM}(s)\le(s^3+3s^2)/2$ is the algebraic complexity of $s\times s$ matrix multiplication over $\mathbb{Z}$.*

Proof. Each node at the level k (level 0 being the root) of the tree essentially requires multiplying $s\times s$ matrices with entries in $\mathbb{Z}[i]$ of height $H_k = N(h'+d'\log N)/2^{k+1}$, plus denominators of the same height. In the FFT model, this may be done in $(2s^2+1)M(H_k)$ operations. Since we assume $M(n)=\Omega(n\log n)$, we have $\sum_{k=0}^{\lceil\log N\rceil} 2^kM(H_k) \le \frac{1}{2}M(H_0)$ (a remark attributed to D. Stehlé in [37]). Kramer's trick saves another factor $\frac{3}{2}$. In the black-box model, the corresponding cost for one node is $(3\,\mathrm{MM}(s)+1)M(H_k)$ with Karatsuba's formula. Stehlé's argument applies again. □

Similar results can be stated for general $\mathbb{Z}$-algebras, using the rank and multiplicative complexity [9] of the algebra.

2.3. **Sparsity and structure.** Continuing on Remark 1, in many cases, the matrices at the leaves of the product tree have some structure that is preserved by successive multiplications. For instance, let $s_n = \sum_{k=0}^{n-1} u_k$ where (u_n) satisfies (2). It is easy to compute a recurrence and initial conditions for (s_n) and go on as above. However, unrolling the recurrences (2) and $s_{n+1} - s_n = u_n$ simultaneously as

$$\begin{pmatrix} u_{n+1} \\ \vdots \\ u_{n+s-1} \\ u_{n+s} \\ s_{n+1} \end{pmatrix} = \begin{pmatrix} & 1 & & & 0 \\ & & \ddots & & \vdots \\ & & & 1 & 0 \\ * & * & \cdots & * & 0 \\ 1 & 0 & \cdots & 0 & 1 \end{pmatrix} \begin{pmatrix} u_n \\ \vdots \\ u_{n+s-2} \\ u_{n+s-1} \\ s_n \end{pmatrix} \tag{4}$$

is more efficient. Indeed, all matrices in the product tree for the numerator of (4) then have a rightmost column of zeros, except for the value in the lower right corner, which is precisely the denominator. With the notation $\mathrm{MM}(s)$ of Proposition 1, each product of these special matrices uses $\mathrm{MM}(s) + s^2 + s + 1$ multiplications, *vs.* $\mathrm{MM}(s+1)+1$ for the dense variant. Hence the formula (4) is more efficient as soon as $\mathrm{MM}(s+1) - \mathrm{MM}(s) \geq s(s+1)$, which is true both for the naïve multiplication algorithm and for Waksman's algorithm (compare [37]). In practice, on Ex. 3 above, if one puts $u_n = s_{n+1} - s_n$ in (2, 3) instead of using (4), the timing grows from 1.7 s to 2.9 s.

The same idea applies to any recurrence operator that can be factored. Further examples of structure in product trees include even and odd D-finite series (*e.g.*, [8, §4.9.1]). In all these cases, the naïve matrix multiplication algorithm automatically benefits from the special shape of the problem (because multiplications by constants have negligible cost), while fast methods must take it into account explicitly.

2.4. **Memory issues.** One weakness of binary splitting is its space complexity $\Omega(n \log n)$ even when the size of the result is only $O(n)$. Several approaches have been developed to deal with this, usually in the case of recurrences of the first order [10, §2.1]. In particular, replacing the rational numbers with $O(n)$-digits floating-point approximations in the $O(\log \log n)$ upper levels of the product tree has been reported to give good results [17, 23]. However, for this to be rigorous, we would need to prove that no cancellation occurs, which is hard to do save for special cases.

3. High-precision evaluation of D-finite functions

3.1. **Within the disk of convergence.** Let $y(z) = \sum_n y_n z^n$ be a D-finite series with radius of convergence ρ at the origin. Let $z \in \mathbb{Q}(i)$ be such that $|z| < \rho$ and $\mathrm{height}(z) \leq h$. The sequence $(y_n z^n)$ is P-recursive, so that the binary splitting algorithm yields a fast method for the high-precision evaluation of $y(z)$. Here "high-precision" means that we let the precision required for the result go to infinity in the complexity analysis of the algorithm.

More precisely, $(y_n z^n)$ is canceled by a recurrence relation of height $O(h)$. By Theorem 1, $y(z)$ may be computed to the precision 10^{-p} in

$$O\big(M\big(N(h + \log N)\big) \log(Nh)\big) \tag{5}$$

bit operations, where N is chosen so that $|y_{N;}(z)| \leq 10^{-p}$, *i.e.* $N \sim p/\log(p/|z|)$ if $\rho < \infty$, and $N \sim p/(\tau \log p)$ for some τ if $\rho = \infty$.

In practice, the numbers of digits of (absolute) precision targeted in NumGfun range from the hundreds to the millions. Accuracies of this order of magnitude are useful in some applications to number theory [12], and in heuristic equality tests [31, §5]. It can also happen that the easiest way to obtain a moderate-precision output involves high-precision intermediate computations, especially when the correctness of the result relies on pessimistic bounds.

3.2. **Analytic continuation.** Solutions of the differential equation (1) defined in the neighborhood of 0 extend by analytic continuation to the universal covering of $\mathbb{C} \setminus S$, where S is the (finite) set of singularities of the equation. D-finite functions may be evaluated fast at any point by a numerical version of the analytic continuation process that builds on the previous algorithm. Rewrite (1) in matrix form

$$Y'(z) = A(z)Y(z), \qquad Y(z) = \left(\frac{y^{(r-1)}(z)}{(r-1)!}\right)_{0 \leq i < r}. \tag{6}$$

This choice of $Y(z)$ induces, for all $z_0 \in \mathbb{C} \setminus S$, that of a family of *canonical solutions* of (1) defined by

$$y[z_0, j](z) = (z - z_0)^j + O\big((z - z_0)^r\big), \qquad 0 \leq j < r,$$

that form a basis of the solutions of (1) in the neighborhood of z_0. Stated otherwise, the vector $\boldsymbol{y}[z_0] = (y[z_0, j])_{0 \leq j < r}$ of canonical solutions at z_0 is the first row of the fundamental matrix $\boldsymbol{Y}[z_0](z)$ of (6) such that $\boldsymbol{Y}[z_0](z_0) = \mathrm{Id}$.

By linearity, for any path $z_0 \to z_1$ in $\mathbb{C} \setminus S$, there exists a *transition matrix* $M_{z_0 \to z_1}$ that "transports initial conditions" (and canonical solutions) along the path:

$$Y(z_1) = M_{z_0 \to z_1} Y(z_0), \quad \boldsymbol{y}[z_0](z) = \boldsymbol{y}[z_1](z) M_{z_0 \to z_1}. \tag{7}$$

This matrix is easy to write out explicitly:

$$M_{z_0 \to z_1} = \boldsymbol{Y}[z_0](z_1) = \left(\frac{1}{i!} y[z_0, j]^{(i)}(z_1)\right)_{0 \leq i,j < r}, \tag{8}$$

evaluations at z_1 being understood to refer to the analytic continuation path $z_0 \to z_1$. Transition matrices compose:

$$M_{z_0 \to z_1 \to z_2} = M_{z_1 \to z_2} M_{z_0 \to z_1}, \quad M_{z_1 \to z_0} = M_{z_0 \to z_1}^{-1}. \tag{9}$$

NumGfun provides functions to compute M_γ for a given path γ (`transition_matrix`), and to evaluate the analytic continuation of a D-finite function along a path starting at 0 (`evaldiffeq`). In both cases, the path provided by the user is first subdivided into a new path $z_0 \to z_1 \to \cdots \to z_m$, $z_\ell \in \mathbb{Q}(i)$, such that, for all ℓ, the point $z_{\ell+1}$ lies within the disk of convergence of the Taylor expansions at z_ℓ of all solutions of (1). Using (8), approximations $\tilde{M}_\ell \in \mathbb{Q}(i)^{r \times r}$ of $M_{z_\ell \to z_{\ell+1}}$ are computed by binary splitting. More precisely, let $w(z) = y^{(k)}(z_\ell + z) = \sum_{n=0}^{\infty} w_n z^n$ where y is a generic solution of (1) and (w_n) satisfies a recurrence of order s deduced from the differential equation. For some suitable N, we compute by binary splitting the linear form $(w_0, \ldots, w_{s-1}) \mapsto w_{;N}(z_{\ell+1} - z_\ell)$. Applying it to the matrix of initial conditions

$$\big(\frac{1}{k!} y[z_0, j]_i^{(k)} (z_{\ell+1} - z_\ell)^i\big)_{0 \leq i < s, 0 \leq j < r}$$

yields the $(k+1)$th row of $\tilde{M}_\ell$.

In the function `transition_matrix`, we then form the product $\tilde{M}_{m-1}\cdots\tilde{M}_0$, again by binary splitting. In the case of `evaldiffeq`, we skip the computation of all but the first row $\tilde{R}$ of $\tilde{M}_{m-1}$, and we form the product $\tilde{R}\tilde{M}_{m-2}\cdots\tilde{M}_0\tilde{I}$, where $\tilde{I}$ is an approximation with coefficients in $\mathbb{Q}(i)$ of the vector $Y(z_0)$ of initial conditions (or this vector itself, if it has symbolic entries). The whole computation is done keeping fractions unsimplified and controlling all errors to guarantee the final result. We postpone the discussion of approximation errors to §6.

Example 4. Transition matrices corresponding to paths that "go round" exactly one singularity once are known as local monodromy matrices. A simple example is that of the equation $(1+z^2)y'' + 2zy' = 0$, whose solution space is generated by 1 and $\arctan z$. Rather unsurprisingly, around i:

```
> transition_matrix((1+z^2)*diff(y(z),z,y(z),z)
  +2*z*diff(y(z),z), y(z), [0,I+1,2*I,I-1,0], 20);
```

$$\begin{pmatrix} 1.00000000000000000000 & 3.14159265358979323846 \\ 0 & 1.00000000000000000000 \end{pmatrix}$$

More generally, expressing them as entries of monodromy matrices is a way to compute many mathematical constants.

Another use of analytic continuation appears in Ex. 2: there, despite the evaluation point lying within the disk of convergence, NumGfun performs analytic continuation along the path $0 \to \frac{-1}{2} \to \frac{-3}{4} \to \frac{-22}{25} \to \frac{-47}{50} \to \frac{-99}{100}$ to approach the singularity more easily (*cf.* [12, Prop. 3.3]).

3.3. **The "bit burst" algorithm.** The complexity result (5) is quasi-optimal for $h = O(\log p)$, but becomes quadratic in p for $h = \Theta(p)$, which is the size of the approximation $\tilde{z} \in \mathbb{Q}(i)$ needed to compute $y(z)$ for arbitrary $z \in \mathbb{C}$. This issue can be avoided using analytic continuation to approach z along a path $z_0 \to z_1 \to \cdots \to z_m = \tilde{z}$ made of approximations of z whose precision grow geometrically:

$$\text{height}(z_\ell) = O(2^\ell), \qquad |z_{\ell+1} - z_\ell| \le 2^{-2^\ell}, \tag{10}$$

thus balancing h and $|z|$. This idea is due to Chudnovsky and Chudnovsky [12], who called it the bit burst algorithm, and independently to van der Hoeven with a tighter complexity analysis [30, 35]. The following proposition improves this analysis by a factor $\log\log p$ in the case where y has a finite radius of convergence. No similar improvement seems to apply to entire functions.

Proposition 2. *Let y be a D-finite function. One may compute a 2^{-p}-approximation of y at a point $z \in \mathbb{Q}(i)$ of height $O(p)$ in $O(M(p\log^2 p))$ bit operations.*

Proof. By (5) and (10), the step $z_\ell \to z_{\ell+1}$ of the bit-burst algorithm takes

$$O\left(M\left(p(2^\ell + \log p)\log p / 2^\ell\right)\right)$$

bit operations. Now

$$\sum_{\ell=0}^{m} M\Big(\frac{p(2^\ell+\log p)}{2^\ell}\Big)\log p \le M\Big(mn\log p + \sum_{\ell=0}^{\infty}\frac{p\log^2 p}{2^\ell}\Big)$$

and $m = O(\log p)$, hence the total complexity. □

Example 5. Consider the "generic" D-finite function y defined by the following equation, picked at random:

$$\begin{aligned}\Big(\frac{5}{12}-\frac{1}{4}z+\frac{19}{24}z^2-\frac{5}{24}z^3\Big)\,y^{(4)}+\Big(-\frac{7}{24}+\frac{2}{3}z+\frac{13}{24}z^2+\frac{1}{12}z^3\Big)\,y'''\\ +\Big(\frac{7}{12}-\frac{19}{24}z+\frac{1}{8}z^2+\frac{1}{3}z^3\Big)\,y''+\Big(-\frac{3}{4}+\frac{5}{12}z+\frac{5}{6}z^2+\frac{1}{2}z^3\Big)\,y'\\ +\Big(\frac{5}{24}+\frac{23}{24}z+\frac{7}{8}z^2+\frac{1}{3}z^3\Big)\,y=0,\\ y(0)=\frac{1}{24}, y'(0)=\frac{1}{12}, y''(0)=\frac{5}{24}, y'''(0)=\frac{5}{24}.\end{aligned}$$

The singular points are $z \approx 3.62$ and $z \approx 0.09 \pm 0.73i$. By analytic continuation (along a path $0 \to \pi i$ homotopic to a segment) followed by bit-burst evaluation, we obtain

$$y(\pi i) \approx -0.52299...(990\text{ digits})...53279 - 1.50272...90608i.$$

This example is among the "most general" that NumGfun can handle. It somehow "combines all possible difficulties", and intermediate computations involve recurrences of order 8 and degree 17. With the current (unoptimized) analytic continuation code, the computation takes about 50 min.

4. Repeated evaluations

Drawing plots or computing integrals numerically requires to evaluate the same function at many points, often to a few digits of precision only. NumGfun provides limited support for this through the function `diffeqtoproc`, which takes as input a D-finite function y, a target precision ϵ, and a list of disks, and returns a Maple procedure that performs the numerical evaluation of y. For each disk $D = \{z, |z - c\lfloor < \rho\}$, `diffeqtoproc` computes a polynomial $p \in \mathbb{Q}(i)[z]$ such that $|\diamond(p(z)) - y(z)| < \epsilon$ for $z \in D \cap \mathbb{Q}(i)$, where $\diamond : \mathbb{Q}(i) \to 10^{-\infty}\mathbb{Z}[i]$ denotes the rounding operation. The procedure returned by `diffeqtoproc` uses the precomputed p when possible, and falls back on calling `evaldiffeq` otherwise.

The approximation polynomial p is constructed as a linear combination of truncated Taylor series of fundamental solutions $y[c, j]$, with coefficients obtained by numerical analytic continuation from 0 to c. Its computation relies on pessimistic bounds. A natural (but not implemented yet) improvement would be to first aim for a slightly smaller error $\|y - p\|_{\infty,D}$, then drop high-degree terms of p (and, possibly, replace its coefficients by approximations) to replace it by a polynomial $\tilde{p}$ of smaller degree and height for which we can bound $\|p - \tilde{p}\|_\infty$ [34, §6.2].

Example 6. The plot of the "random" D-finite function of Ex. 5 on $I = [-3.4, 3.6]$ appearing on Fig. 1 was obtained using polynomial approximations on several disks that cover the interval I while avoiding the singular points of the equation. The precomputation phase takes 2 min. After that, plotting is instantaneous.

5. Regular singular points

The algorithms of the previous section extend to the case where the analytic continuation path passes through *regular singular points* of the differential equation [11, 12, 31]. Work is in progress to support this in NumGfun, with two main applications in view, namely special functions (such as Bessel functions) defined using their local behaviour at a regular singular point, and "connection coefficients"

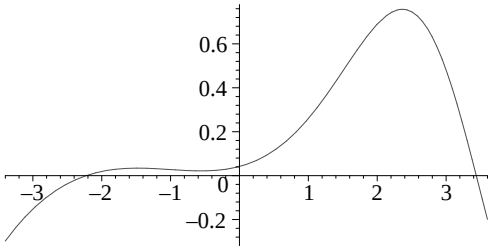

FIGURE 1. The function of Example 5.

arising in analytic combinatorics [15, § VII.9]. Since this extension is not yet implemented in NumGfun, we limit ourselves to a sketchy (albeit technical) discussion.

Recall that a singular point z_0 of a linear differential equation with analytic coefficients is called *regular* when all solutions $y(z)$ have moderate growth $y(z) = 1/(z - z_0)^{O(1)}$ as $z \to z_0$ in a sector with vertex at z_0, or equivalently when the equation satisfies an easy-to-check formal property called Fuchs' criterion. (We refer to textbooks such as [27] for precise statements and references concerning the theory of regular singular points.) As also shown by Fuchs, the solutions in the neighborhood of a regular singular point, say $z_0 = 0$, have a simple structure: for some $t \in \mathbb{N}$, there exist linearly independent formal solutions of the form

$$y(z) = z^\lambda \sum_{k=0}^{t-1} \varphi_k(z) \log^k z, \quad \lambda \in \mathbb{C}, \quad \varphi_k \in \mathbb{C}[[z]] \tag{11}$$

in number equal to the order r of the differential equation. Additionally, the φ_k converge on the open disk centered at 0 extending to the nearest singular point, so that the solutions (11) in fact form a basis of analytic solutions on any open sector with vertex at the origin contained in this disk.

Several extensions of the method of indeterminate coefficients used to obtain power series solutions at ordinary points allow to determine the coefficients of the series φ_k. We proceed to rephrase Poole's simpler variant [27, §16] of Heffter's method [21, Kap. 8] as a generalization of the "operator algebra point of view" on indeterminate coefficients, in which a recursive formula for the coefficients of the series expansion of a solution is obtained by applying simple rewriting rules to the differential equation.

Since our interest lies in the D-finite case, we assume that 0 is a regular singular point of Equation (1). Letting $\theta = z\frac{\mathrm{d}}{\mathrm{d}z}$, the equation rewrites as $L(z, \theta) \cdot y = 0$ where L is a polynomial in two noncommutative indeterminates (and $L(z, \theta)$ has no nontrivial left divisor in $K[z]$). Based on (11), we seek solutions as *formal logarithmic sums*

$$y(z) = \sum_{n \in \lambda + \mathbb{Z}} \sum_{k \geq 0} y_{n,k} \frac{\log^k z}{k!} z^n, \qquad \lambda \in \mathbb{C}.$$

Let us call the double sequence $\boldsymbol{y} = (y_{n,k})_{n \in \lambda + \mathbb{Z}, k \in \mathbb{N}}$ the *coefficient sequence* of y. The *shift operators* S_n and S_k on such double sequences are defined by $S_n \cdot \boldsymbol{y} = (y_{n+1,k})$, and $S_k \cdot \boldsymbol{y} = (y_{n,k+1})$. The heart of the method lies in the following observation.

Proposition 3. *Let $y(z)$ be a formal logarithmic sum. The relation $L(z,\theta)\cdot y = 0$ holds (formally) iff the coefficient sequence $\boldsymbol{y}$ of y satisfies $L(S_n^{-1}, n+S_k)\cdot \boldsymbol{y} = 0$.*

Proof. The operators z and θ act on logarithmic sums by

$$z \cdot y(z) = \sum_{n\in\lambda+\mathbb{Z}} \sum_{k\geq 0} y_{n-1,k} \frac{\log^k z}{k!} z^n$$

and

$$\theta \cdot y(z) = \sum_{n\in\lambda+\mathbb{Z}} \sum_{k\geq 0} (n y_{n,k} + y_{n,k+1}) \frac{\log^k z}{k!} z^n.$$

Thus the coefficient sequence of $L(z,\theta)\cdot y$ is $L(S_n^{-1}, n+S_k)\cdot \boldsymbol{y}$. □

Assume that $y(z)$ satisfies (1). Then $L(S_n^{-1}, n+S_k)\cdot \boldsymbol{y} = 0$; additionally, (11) implies that $y_{n,k} = 0$ for $k \geq t$, which translates into $S_k^t \cdot \boldsymbol{y} = 0$. Set $L(z,\theta) = Q_0(z) + R(z,\theta)z$, and fix $n \in \lambda + \mathbb{Z}$. If $Q_0(n) \neq 0$, then $Q_0(n+X)$ is invertible in $K(\lambda)[[X]]$, and

$$\boldsymbol{y} = -\big(Q_0(n+S_k)^{-1} \bmod S_k^t\big) R(S_n^{-1}, n+S_k) S_n^{-1} \cdot \boldsymbol{y}.$$

In general, let $\mu(n)$ be the multiplicity of n as a root of Q_0. Take $T_n \in K(\lambda)[X]$ such that

$$T_n(X)\big(X^{-\mu(n)} Q_0(n+X)\big) = 1 + O(X^t)$$

(explicitly, $T_n(S_k) = \sum_{v=0}^{t-1} \big(\frac{\partial^v}{\partial X^v} \frac{X^{\mu(n)}}{Q_0(X)}\big)_{X=n} S_k^v$). Then, it holds that

$$S_k^{\mu(n)} \cdot \boldsymbol{y} = -T_n(S_k) R(S_n^{-1}, n+S_k) S_n^{-1} \cdot \boldsymbol{y}, \tag{12}$$

hence the $y_{n',k}$ with $n' < n$ determine $(y_{n,k})_{k\geq\mu(n)}$, while the $y_{n,k}$, $k < \mu(n)$ remain free.

Coupled with the condition following from (11) that the solutions must satisfy $y_{n,k} = 0$ for $n-\lambda < 0$, this implies that a solution y is characterized by $(y_{n,k})_{(n,k)\in E}$, where $E = \{(n,k) | k < \mu(n)\}$. As in §3, this choice of initial conditions induces that of canonical solutions (at 0) $y[n,k]$ defined by $y[n,k]_{n,k} = 1$ and $y[n,k]_{n',k'} = 0$ for $(n',k') \in E \setminus \{(n,k)\}$. The notion of transition matrix extends, see [31, §4] for a detailed discussion.

Equation (12) is a "recurrence relation" that lends itself to binary splitting. The main difference with the setting of §2 is that the companion matrix of the "recurrence operator" now contains truncated power series, *i.e.*, $B \in K(n)[[S_k]]/(S_k^t)$. The coefficients $y[u,v]_n = \sum_{k=0}^{t-1} y[u,v]_{n,k} \log^k z$ of canonical solutions may be computed fast by forming the product $B(n-1)\cdots B(u) \in K(\lambda)[[S_k]]/(S_k^t)$ and applying it to the initial values $Y_u = (0,\dots,0,\log^v z)^{\mathrm{T}}$. Compared to the algorithm of [31], multiplications of power series truncated to the order t replace multiplications of $t\times t$ submatrices, so that our method is slightly faster. As in §3, all entries of each row of $M_{z_0\to z_1}$ may (and should) be computed at once.

6. Error control

Performing numerical analytic continuation rigorously requires various bounds that control the behaviour of the function under consideration. We now describe how error control is done in NumGfun. Many ideas used in this section appear scattered in [30, 31, 32, 33, 34]. NumGfun relies almost exclusively on *a priori* bounds; see [26, §5.2] for pointers to alternative approaches, and [34] for further

useful techniques, including how to refine rough *a priori* bounds into better *a posteriori* bounds.

We start by discussing the computation of *majorant series* for the canonical solutions of the differential equation. Then we describe how these are used to determine, on the one hand, at which precision each transition matrix should be computed for the final result to be correct, and on the other hand, where to truncate the series expansions of canonical solutions to achieve this precision. The last paragraphs deal with the special case of `diffeqtoproc`, and with a technique to limit the cost of computing bounds in "bit burst" phases.

Remark 2. In practice, some of the error control is still done in "heuristic" floating-point arithmetic, meaning that numerical errors that happen during the computation of the error bounds themselves are not always controlled. However, we have taken some care to ensure that crucial steps rely on rigorous methods such as interval arithmetic.

6.1. **Majorant series.** A formal power series $g \in \mathbb{R}_+[[z]]$ is a *majorant series* of $y \in \mathbb{C}[[z]]$, which we denote by $y \trianglelefteq g$, if $|y_n| \leq g_n$ for all n. If $y(z) \trianglelefteq g(z)$ and $\hat{y}(z) \trianglelefteq \hat{g}(z)$, then

$$\begin{gathered} y(z) \leq g(|z|), \qquad y'(z) \trianglelefteq g'(z), \\ y + \hat{y} \trianglelefteq g + \hat{g}, \qquad y\hat{y} \trianglelefteq g\hat{g}, \qquad y \circ \hat{y} \trianglelefteq g \circ \hat{g}. \end{gathered} \tag{13}$$

We shall allow majorant series to be formal logarithmic sums or matrices. The relation $\trianglelefteq$ extends in a natural way: $\sum_{n,k} y_{n,k} z^n \log z^k \trianglelefteq \sum_{n,k} g_{n,k} z^n \log z^k$ iff $|y_{n,k}| \leq g_{n,k}$ for all n and k, and $Y = (y_{i,j}) \trianglelefteq G = (g_{i,j})$ iff Y and G have the same format and $y_{i,j} \trianglelefteq g_{i,j}$ for all i, j. In particular, for scalar matrices, $Y \trianglelefteq G$ if $|y_{i,j}| \leq g_{i,j}$ for all i, j. The inequalities (13) still hold.

6.2. **Bounds on canonical solutions.** The core of the error control is a function that computes $g(z)$ such that

$$\forall j, \quad y[z_0, j](z_0 + z) \trianglelefteq g(z) \tag{14}$$

(in the notations of §3) given Equation (1) and a point z_0. This function is called at each step of analytic continuation.

The algorithm is based on that of [26] ("SB" in what follows), designed to compute "asymptotically tight" *symbolic* bounds. For space reasons, we only sketch the differences between SB and the variant we use for error control. Once adapted to work over $\mathbb{Q}(i)$ instead of $\mathbb{Q}$ (and restricted to convergent series), SB returns a power series satisfying (14) of the form $g(z) = \sum_{n=0}^{\infty} n!^{-\tau} \alpha^n \varphi(n) z^n$, where $\tau \in \mathbb{Q}_+$, $\alpha \in \bar{\mathbb{Q}}$, and $\varphi(n) = e^{o(n)}$. The series g is specified by τ, α, and other parameters (of no interest here) to define φ. The tightness property means that τ and α are the best possible.

However, the setting of numerical evaluation differs from that of symbolic bounds in several respects: (i) the issue is no more to obtain human-readable formulae, but bounds that are easy to evaluate; (ii) bound computations should be fast; (iii) at regular singular points, the fundamental solutions are logarithmic sums, not power series.

We take advantage of (i) to solve (ii). The most important change is that the parameter α is replaced by an approximation $\tilde{\alpha} \geq \alpha$. Algorithms 3 and 4 from [26] are adapted to use $\tilde{\alpha}$ instead of α. As a consequence, the parameter T from [26,

§3.3] is also replaced by an upper bound. These changes avoid computations with algebraic numbers, which are the main bottleneck in SB. In particular, equality tests of "dominant roots" of polynomials can be done numerically (Line 9 of Algorithm 3 and Line 5 of Algorithm 4 become much less expensive). Strictly speaking, introducing $\tilde\alpha$ means that we give up the tightness property. However, in contrast with the "plain" majorant series method [31, 32, 34], we are free to take $\tilde\alpha$ arbitrarily close to α, since we do not use the ratio $(\alpha/\tilde\alpha)^n$ to mask subexponential factors.

The issue (iii) can be dealt with too, but like the algorithms of §5, this part is not implemented in NumGfun. With the notations of §5, one may compute a series g such that $y(z) \trianglelefteq g(z)\sum_{k=0}^{t-1}\frac{\log^k z}{k!}$ by modifying the choice of K in [26, §3.3] so that $\sum_{k=0}^{t-1}\left|\left(\frac{\partial^k}{\partial X^k}\frac{X^{\mu(n)}}{Q_0(X)}\right)_{X=n}\right| \le K/n^r$, and replacing [26, Eq. (14)] by Eq. (12) above in the reasoning.

6.3. **Global error control.** We now consider the issue of controlling how the approximations at each step of analytic continuation accumulate. Starting with a user-specified error bound ϵ, we first set ϵ' so that an ϵ'-approximation $\tilde r \in \mathbb{Q}(i)$ of the exact result r rounds to $\diamond(\tilde r) \in 10^{-\infty}\mathbb{Z}[i]$ with $|\diamond(\tilde r) - r| < \epsilon$. No other *rounding* error occur, since the rest of the computation is done on objects with coefficients in $\mathbb{Q}(i)$. However, we have to choose the precision at which to compute each factor of the product $\Pi = \tilde R\tilde M_{m-1}\cdots\tilde M_0\tilde I$ (in `evaldiffeq`, and of similar products for other analytic continuation functions) so that $|\tilde r - r| < \epsilon'$.

As is usual for this kind of applications, we use the Frobenius norm, defined for any (not necessarily square) matrix by

$$\|(a_{i,j})\|_{\mathrm F} = (\sum_{i,j}|a_{i,j}|^2)^{1/2}.$$

The Frobenius norm satisfies $\|AB\|_{\mathrm F} \le \|A\|_{\mathrm F}\|B\|_{\mathrm F}$ for any matrices A, B of compatible dimensions. If A is a square $r\times r$ matrix,

$$\|A\|_\infty \le |||A|||_2 \le \|A\|_{\mathrm F} \le r\|A\|_\infty \tag{15}$$

where $|||\cdot|||_2$ is the matrix norm induced by the Euclidean norm while $\|\cdot\|_\infty$ denotes the *entrywise* uniform norm. Most importantly, the computation of $\|A\|_{\mathrm F}$ is numerically stable, and if A has entries in $\mathbb{Q}(i)$, it is easy to compute a good upper bound on $\|A\|_{\mathrm F}$ in floating-point arithmetic.

We bound the total error ϵ_Π on Π using repeatedly the rule

$$\|\tilde A\tilde B - AB\|_{\mathrm F} \le \|\tilde A\|_{\mathrm F}\|\tilde B - B\|_{\mathrm F} + \|\tilde A - A\|_{\mathrm F}\|B\|_{\mathrm F},$$

and we compute precisions ϵ_A so that having $\|\tilde A - A\|_{\mathrm F} < \epsilon_A$ for each factor A of Π ensures that $\epsilon_\Pi < \epsilon'$. Upper bounds on the norms $\|\tilde A\|_{\mathrm F}$ and $\|A\|_{\mathrm F}$ appearing in the error estimate are computed either from an approximate value of A (usually $\tilde A$ itself) if one is available at the time, or from a matrix G such that $A \trianglelefteq G$ given by (13, 14).

6.4. **Local error control.** We turn to the computation of individual transition matrices—in the case of ordinary points, for simplicity and fidelity to the implementation. Given a precision ϵ determined by the "global error control" step, our task is to choose N such that $\|\tilde M - M_{z_0\to z_1}\|_{\mathrm F} \le \epsilon$ if $\tilde M$ is computed by truncating the entries of (8) to the order N. If g satisfies (14), it suffices that $g^{(i)}_{N;}(|z_1 - z_0|) \le \frac{i!}{r}\epsilon$

for all i, as can be seen from (13, 15). We find a suitable N by dichotomic search. For the family of majorant series g used in NumGfun, the $g_{N;}^{(i)}(x)$ are not always easy to evaluate, so we bound them by expressions involving only elementary functions as explained in [26, §4.2]. The basic idea, known as the saddle-point method in asymptotic analysis, is that if $x \geq 0$, inequalities like $g_{n;}(x) \leq (x/t_n)^n g(t_n)(1 - x/t_n)^{-1}$ are asymptotically tight for suitably chosen t_n.

6.5. **Precomputation of local expansions.** The way we choose expansion orders for the function `diffeqtoproc` of §4 is similar: we first compute a bound B_{can} on the values of the fundamental solutions and their first derivatives on the disk D. If the target precision is ϵ, the vector $Y(c)$ of "initial values" at c is computed to the precision ϵ'/B_{can} where $\epsilon' \leq \epsilon/(2r)$. We also compute $B_{\text{ini}} \geq \|Y(c)\|_{\text{F}}$. Each $y[c, j]$ is expanded to an order N such that $\|y[c, j]_{N;}\|_{\infty,D} \leq \epsilon'/B_{\text{ini}}$, so that finally $|p(z) - y(z)| \leq \epsilon$ for $z \in D$.

6.6. **Points of large bit-size.** Computing the majorants (14) is expensive when the point z_0 has large height. This can be fixed by working with an approximate value c of z_0 to obtain a bound valid in a neighborhood of c that contains z_0. This technique is useful mainly in "bit burst" phases (where, additionally, we can reuse the same c at each step).

Assume that $\boldsymbol{Y}[c](c + z) \trianglelefteq G(z)$ for some $c \approx z_0$. Since

$$\boldsymbol{Y}[z_0](z_0 + z) = \boldsymbol{Y}[c](c + (z_0 - c) + z)M_{c\to z_0}^{-1}$$

by (7), it follows that

$$\boldsymbol{Y}[z_0](z_0 + z) \trianglelefteq G(|z_0 - c| + z)\, \|M_{c\to z_0}^{-1}\|\mathbf{1}$$

where $\mathbf{1}$ is a square matrix filled with ones. Now $M_{c\to c+z} = \mathrm{Id} + O(z)$, whence $M_{c\to c+z} - \mathrm{Id} \trianglelefteq G(z) - G(0)$. For $|z_0 - c| < \eta$, this implies $\|M_{c\to z_0} - \mathrm{Id}\,\|_{\text{F}} \leq \delta := \|G(\eta) - G(0)\|_{\text{F}}$. Choosing η small enough that $\delta < 1$, we get $\|M_{c\to z_0}^{-1}\|_{\text{F}} \leq (1-\delta)^{-1}$. If G was computed from (14), *i.e.*, for $G(z) = \left(\frac{1}{j!}g^{(j)}(z)\right)_{i,j}$, this finally gives the bound

$$y[z_0, j](z_0 + z) \trianglelefteq \frac{r}{j!\,(1 - \delta)} g^{(j)}(\eta + z),$$

valid for all z_0 in the disk $|z_0 - c| < \eta$.

Note however that simple differential equations have solutions like $\exp(K/(1-z))$ with large K. The condition $\delta < 1$ then forces us to take c of size $\Omega(K)$. Our strategy to deal with this issue in NumGfun is to estimate K using a point c of size $O(1)$ and then to choose a more precise c (as a last resort, $c = z_0$) based on this value if necessary.

Remark 3. If the evaluation point z is given as a program, essentially the same reasoning as above allows to choose automatically an approximation $\tilde{z} \in \mathbb{Q}(i)$ of z such that $|y(\tilde{z}) - y(z)|$ is below a given error bound [30, §4.3]. In other applications, it is useful to have bounds on transition matrices that hold uniformly for *all* small enough steps in a given domain. Such bounds may be computed from a majorant differential equation with constant coefficients [33, §5].

7. Final remarks

Not all of NumGfun was described in this article. The symbolic bounds mentioned in §6 are also implemented, with functions that compute majorant series or other kinds of bounds on rational functions (`bound_ratpoly`), D-finite functions (`bound_diffeq` and `bound_diffeq_tail`) and P-recursive sequences (`bound_rec` and `bound_rec_tail`). This implementation was already presented in [26].

Current work focuses on adding support for evaluation "at regular singular points" (as outlined in §5), and improving performance. In the longer term, I plan to rewrite a significant part of the code "from the bottom up" to make it more efficient and less dependent on Maple.

7.1. **Acknowledgements.** I am grateful to my advisor, B. Salvy, for encouraging me to conduct this work and offering many useful comments. Thanks also to A. Benoit, F. Chyzak, P. Giorgi, J. van der Hoeven and A. Vaugon for stimulating conversations and/or comments on drafts of this article.

This research was supported in part by the MSR-INRIA joint research center.

References

[1] J. Abad, F. J. Gómez, and J. Sesma. An algorithm to obtain global solutions of the double confluent Heun equation. *Numerical Algorithms*, 49(1-4):33–51, 2008.
[2] M. Beeler, R. W. Gosper, and R. Schroeppel. Hakmem. AI Memo 239, MIT Artificial Intelligence Laboratory, 1972.
[3] F. Bellard. 2010. `http://bellard.org/pi/pi2700e9/`
[4] D. J. Bernstein. Fast multiplication and its applications. In J. Buhler and P. Stevenhagen, editors, *Algorithmic Number Theory*, pages 325–384. Cambridge University Press, 2008.
[5] A. Bostan, T. Cluzeau, and B. Salvy. Fast algorithms for polynomial solutions of linear differential equations. In *ISSAC'05*, pages 45–52. ACM press, 2005.
[6] R. P. Brent. The complexity of multiple-precision arithmetic. In R. S. Anderssen and R. P. Brent, editors, *The Complexity of Computational Problem Solving*, pages 126–165, 1976.
[7] R. P. Brent. A Fortran multiple-precision arithmetic package. *ACM Transactions on Mathematical Software*, 4(1):57–70, 1978.
[8] R. P. Brent and P. Zimmermann. *Modern Computer Arithmetic*. Version 0.4. 2009.
[9] P. Bürgisser, M. Clausen, and M. A. Shokrollahi. *Algebraic complexity theory*. Springer Verlag, 1997.
[10] H. Cheng, G. Hanrot, E. Thomé, E. Zima, and P. Zimmermann. Time- and space-efficient evaluation of some hypergeometric constants. In *ISSAC'07*. ACM press, 2007.
[11] D. V. Chudnovsky and G. V. Chudnovsky. On expansion of algebraic functions in power and Puiseux series (I, II). *Journal of Complexity*, 2(4):271–294, 1986; 3(1):1–25, 1987.
[12] D. V. Chudnovsky and G. V. Chudnovsky. Computer algebra in the service of mathematical physics and number theory. In *Computers in mathematics (Stanford, CA, 1986)*, pages 109–232. 1990.
[13] R. Dupont. *Moyenne arithmético-géométrique, suites de Borchardt et applications*. Thèse de doctorat, École polytechnique, Palaiseau, 2006.
[14] P. Falloon, P. Abbott, and J. Wang. Theory and computation of spheroidal wavefunctions. *Journal of Physics A: Mathematical and General*, 36:5477–5495, 2003.
[15] P. Flajolet and R. Sedgewick. *Analytic Combinatorics*. Cambridge University Press, 2009.
[16] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software*, 33(2):13:1–13:15, 2007.
[17] X. Gourdon and P. Sebah. Binary splitting method, 2001. `http://numbers.computation.free.fr/`
[18] T. Granlund. *GMP*. `http://gmplib.org/`
[19] B. Haible and R. B. Kreckel. *CLN*. `http://www.ginac.de/CLN/`

[20] B. Haible and T. Papanikolaou. Fast multiprecision evaluation of series of rational numbers, 1997.
[21] L. Heffter. *Einleitung in die Theorie der linearen Differentialgleichungen*. Teubner, Leipzig, 1894.
[22] E. Jeandel. Évaluation rapide de fonctions hypergéométriques. Rapport technique RT-0242, INRIA-ENS Lyon, 2000.
[23] R. B. Kreckel. decimal(γ) =$\sim$ "0.57721566[0-9]{1001262760}39288477", 2008.
[24] O. M. Makarov. An algorithm for multiplying 3×3 matrices. *USSR Computational Mathematics and Mathematical Physics*, 26(1):179–180, 1987.
[25] L. Meunier and B. Salvy. ESF: an automatically generated encyclopedia of special functions. In *ISSAC'03*, pages 199–206. ACM Press, 2003.
[26] M. Mezzarobba and B. Salvy. Effective bounds for P-recursive sequences. *Journal of Symbolic Computation*, to appear. arXiv:0904.2452v2.
[27] E. G. C. Poole. *Introduction to the theory of linear differential equations*. The Clarendon Press, Oxford, 1936.
[28] B. Salvy and P. Zimmermann. Gfun: A Maple package for the manipulation of generating and holonomic functions in one variable. *ACM Transactions on Mathematical Software*, 20(2):163–177, 1994.
[29] R. P. Stanley. *Enumerative combinatorics*, volume 2. Cambridge University Press, 1999.
[30] J. van der Hoeven. Fast evaluation of holonomic functions. *Theoretical Computer Science*, 210(1):199–216, 1999.
[31] J. van der Hoeven. Fast evaluation of holonomic functions near and in regular singularities. *Journal of Symbolic Computation*, 31(6):717–743, 2001.
[32] J. van der Hoeven. Majorants for formal power series. Technical Report 2003-15, Université Paris-Sud, 2003.
[33] J. van der Hoeven. Efficient accelero-summation of holonomic functions. *Journal of Symbolic Computation*, 42(4):389–428, 2007.
[34] J. van der Hoeven. On effective analytic continuation. *Mathematics in Computer Science*, 1(1):111–175, 2007.
[35] J. van der Hoeven. *Transséries et analyse complexe effective*. Mémoire d'habilitation, Université Paris-Sud, 2007.
[36] A. Waksman. On Winograd's algorithm for inner products. *IEEE Transactions on Computers*, C-19(4):360–361, 1970.
[37] P. Zimmermann. The bit-burst algorithm. Slides of a talk at the workshop "Computing by the Numbers: Algorithms, Precision, and Complexity" for R. Brent 60th birthday, 2006.

Marc Mezzarobba, Algorithms Project-Team, INRIA Paris-Rocquencourt, France
E-mail address: marc.mezzarobba@inria.fr