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

Marc Mezzarobba

▶ **To cite this version:**

Marc Mezzarobba. NumGfun: a Package for Numerical and Analytic Computation with D-finite Functions. ISSAC - International Symposium on Symbolic and Algebraic Computation, Jul 2010, Munich, Germany. pp.139-146, 10.1145/1837934.1837965 . inria-00456983v2

HAL Id: inria-00456983

https://inria.hal.science/inria-00456983v2

Submitted on 8 May 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons CC0 - Public Domain Dedication 4.0 International License

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

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

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 improvements over existing algorithms. We also provide references to relevant ideas not currently used in NumGfun.

Categories and Subject Descriptors: I.1.2 [**Symbolic and Algebraic Manipulation**]: Algorithms

General Terms: Algorithms, Experimentation, Theory

Keywords: D-finite functions, linear differential equations, certified numerical computation, bounds, Maple

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, 30]. 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* [25] and its successor under development, the *Dynamic Dictionary of Mathematical Functions* [20], 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], the necessary error bounds also being computed from the differential equation and initial values [32]. However, these algorithms have remained theoretical [13, §9.2.1]. The ability of computer algebra systems to work with D-finite functions is (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 entire 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 that algorithm. NumGfun is distributed as a subpackage of gfun [29], 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 providing an overview of techniques relevant to the development of similar software. The following examples illustrate typical uses of NumGfun, first to compute a remote term from a combinatorial sequence, then to evaluate a special 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[1] 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 $M_{2\cdot10^5}$. On this (non-generic) example, `nth_term` could be made competitive for smaller indices by taking advantage of the fact that the divisions that occur while unrolling the recurrence are exact.

[1]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.

This work is in the public domain. As such, it is not subject to copyright. Where it is not legally possible to consider this work as released into the public domain, any entity is granted the right to use this work for any purpose, without any conditions, unless such conditions are required by law.
ISSAC 2010, 25–28 July 2010, Munich, Germany.

Example 2. The double confluent Heun function $U_{\alpha,\beta,\gamma,\delta}$ satisfies $(z^2-1)^3U''(z)+(2z^5-4z^3-\alpha z^4+2z+\alpha)U'(z)+(\beta z^2+(2\alpha+\gamma)z+\delta)U(z)=0$, $U(0)=1$, $U'(0)=0$. It is singular at $z=\pm1$. The command `evaldiffeq(eq,y(z), -0.99,1000)` where `eq` is this differential equation yields $U_{1,\frac{1}{3},\frac{1}{2},3}(-0.99) \approx 4.67755...(990\text{ digits})...05725$ in 22 s.

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]. It belongs to the family of binary splitting algorithms for D-finite series, hints at the existence of which go back to [2, §178], and generalizes earlier work of Brent [6] for specific elementary functions. Better known (thanks among others to [21]) binary splitting algorithms can be seen as special cases of the bit burst algorithm. One reason why, unlike these special cases, it 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. Van der Hoeven's version [32] 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.

These algorithms extend to the computation of limits of D-finite functions at singularities of their defining equation. The case of regular singularities is treated both in [11, 12], and more completely in [33], that of irregular singular points in [35]. See [4, §12.7], [35, §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, 23]. 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. Works on the implementation of classes of "less common" special functions that overlap those considered in NumGfun include [1, 14].

This work is based in part on the earlier [26].

Contribution. The main contribution presented in this article is NumGfun itself. We recall the algorithms it uses, and discuss various implementation issues. Some of these descriptions include improvements or details that do not seem to have appeared elsewhere. Specifically: (i) we give a new variant of the analytic continuation algorithm for D-finite functions that is faster with respect to the order and degree of the equation; (ii) we improve the complexity analysis of the bit burst algorithm by a factor $\log\log n$; (iii) we point out that Poole's method to construct solutions of differential equations at regular singular points can be rephrased in a compact way in the language of noncommutative polynomials, leading to faster evaluation of D-finite functions in these points; and (iv) we describe in some detail the practical computation of all the bounds needed to obtain a provably correct result.

What NumGfun is not. Despite sharing some of the algorithms used to compute mathematical constants to 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: asymptotic expansions, summation to the least term, and resummation of divergent power series are currently out of scope.

Terminology. Like the rest of gfun, NumGfun works with *D-finite functions* and *P-recursive sequences*. We recall only basic definitions here; see [30, §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

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

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

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

A sequence $(u_n)_{n\in\mathbb{N}}$ is P-recursive if and only if its 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 $u_0, \ldots, u_N$ takes $\Theta(N^2 M(\log N))$ bit operations, which is almost linear in the total size of $u_0, \ldots, u_N$, 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:

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

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 from [12, Theorem 2.2], except that the authors seem to have overlooked the cost of evaluating the polynomials at the leaves of the tree.

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 $H = h' + d' \log N$. 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. The assumption $d, s = o(N)$ allows to write $H = O(sd(h + \log N))$ and get rid of some terms, so that the total complexity simplifies as stated. □

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. The same algorithm works over any algebraic number field instead of $\mathbb{Q}(i)$. This is useful for evaluating D-finite functions "at singularities" (§4). More generally, similar complexity results hold for product tree computations in torsion-free $\mathbb{Z}$-algebras (or $\mathbb{Q}$-algebras: 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.

Constant-factor improvements. Several techniques 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 a discrete Fourier transform of size n (or of another related linear transform, depending on the algorithm). FFT-based integer multiplication algorithms adapt to reduce the multiplication of two matrices of height n in $\mathbb{Z}^{s \times s}$ to $O(n)$ multiplications of matrices of height $O(1)$, for a total of $O(s^2 M(n) + s^\omega n)$ bit operations. This is known as "FFT addition" [4], "computing in the FFT mode" [32], or "FFT invariance". A second improvement ("FFT doubling", attributed to R. Kramer in [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.

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 naïve 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 [38] 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.

Exploiting these ideas leads to the following refinement of Theorem 1. Similar results can be stated for general $\mathbb{Z}$-algebras, using their rank and multiplicative complexity [9].

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 $\mathrm{MM}(s) \leq (s^3 + 3s^2)/2$ 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 $\sum_{k=0}^{\lceil \log N \rceil} 2^k M(H_k) \leq \frac{1}{2} M(H_0)$ (a remark attributed to D. Stehlé in [39]). 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. □

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

Continuing on this remark, these matrices often 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

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

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 [39]). In practice, on Ex. 3 below, if one puts $u_n = s_{n+1} - s_n$ in (2, 3) instead of using (4), the computation time grows from 1.7 s to 2.7 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.

Remark 1. A weakness of binary splitting is its comparatively large space complexity $\Omega(n \log n)$. Techniques to reduce it are known and used by efficient implementations in the case of recurrences of the first order [10, 17, 19, 3].

Implementation. The implementation of binary splitting in NumGfun includes some of the tricks discussed in this section. FFT-based techniques are currently not used because they are not suited to implementation in the Maple language. This implementation is exposed through two user-level functions, `nth_term` and `fnth_term`, that allow to evaluate P-recursive sequences (`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.

3. HIGH-PRECISION EVALUATION OF D-FINITE FUNCTIONS

We now recall the numerical evaluation algorithms used in NumGfun, and discuss their implementation.

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 $\mathrm{height}(z) \le h$. 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 hence 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 $|y_{N;}(z)| \le 10^{-p}$, *i.e.* $N \sim p/\log(\rho/|z|)$ 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 [33, §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.

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 [12]. Rewrite (1) in matrix form

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

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

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

that form a basis of the solutions of (1) in the neighborhood of z_0. Stated otherwise, the vector $\boldsymbol{y}[z_0] = (y[z_0, j])_{0 \le j < r}$ 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:

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

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, we compute all entries of $\tilde{M}_\ell$ at once, as follows. For a generic solution y of (1), the coefficients u_n of $u(z) = y(z_\ell + z) = \sum_{n=0}^{\infty} u_n z^n$ are canceled by a recurrence of order s. Hence the partial sums $u_{;n}(z)$ of u satisfy

$$\begin{pmatrix} U_{n+1} \\ u_{;n+1}(z) \end{pmatrix} = \underbrace{\begin{pmatrix} C(n)z & 0 \\ K & 1 \end{pmatrix}}_{B(n)} \begin{pmatrix} U_n \\ u_{;n}(z) \end{pmatrix}, \tag{10}$$

where $K = (1, 0, \ldots, 0)$ and $C(n) \in \mathbb{Q}(n)^{s \times s}$. Introducing an indeterminate δ, we let $z = z_{\ell+1} - z_\ell + \delta \in \mathbb{Q}(i)[\delta]$ and compute $B(N-1)\ldots B(0) \mod \delta^r$ by binary splitting (an idea already used in [32]), for some suitable N. The upper left blocks of the subproducts are *kept factored* as a power of z times a matrix independent on z. In other words, clearing denominators, the computation of each subproduct

$$P = \frac{1}{d} \begin{pmatrix} D \cdot p & 0 \\ R & d \end{pmatrix} = P_{\mathrm{high}} P_{\mathrm{low}} \qquad (p = \mathrm{numer}(z)^m)$$

is performed as $D \leftarrow D_{\text{high}}D_{\text{low}}$; $p \leftarrow p_{\text{high}}p_{\text{low}}$; $R \leftarrow p_{\text{low}}(R_{\text{high}}C_{\text{low}}) + d_{\text{high}}R_{\text{low}}$; $d \leftarrow d_{\text{high}}d_{\text{low}}$. The powers of z can be computed faster, but the saving is negligible. Applying the row submatrix R of the full product to the matrix $U_0 = (\frac{1}{i!}y[z_\ell, j]^{(i)}(z_\ell))_{0\leqslant i<s, 0\leqslant j<r}$ yields a row vector equal to $(y[z_\ell, j]_{;N}(z_{\ell+1}+\delta))_{0\leqslant j<r} + O(\delta^r)$, each entry of which is a truncated power series whose coefficients are the entries of the corresponding column of $\tilde{M}_\ell$. This way of computing $\tilde{M}_\ell$ is roughly $\min(r, s^{\omega-1})$ times more efficient than the fastest of the variants from [12, 32].

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 compute only 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 on unsimplified fractions, controlling all errors to guarantee the final result. Let us stress that no numerical instability occurs since all numerical operations are performed on rational numbers. We postpone the discussion of approximation errors (including the choice of N) to §5.

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);
```

$$\begin{pmatrix} 1.0000000000000000000 & 3.1415926535897932384 \\ 0 & 1.0000000000000000000 \end{pmatrix}$$

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]).

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:

$$|z_{\ell+1} - z_\ell| \leq 2^{-2^\ell}, \qquad \text{height}(z_\ell) = O(2^\ell), \qquad (11)$$

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 [32, 37]. 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 (11), the step $z_\ell \to z_{\ell+1}$ of the bit-burst algorithm takes $O(M(p(2^\ell + \log p)\log p/2^\ell))$ bit operations. Now

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

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

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

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

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

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

after about 5 min. This example is among the "most general" that NumGfun can handle. Intermediate computations involve recurrences of order 8 and degree 17.

4. 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, 33]. 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" arising in analytic combinatorics [15, § VII.9]. We limit ourselves to a sketchy (albeit technical) discussion.

Recall that a finite 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 a formal property called Fuchs' criterion. The solutions in the neighborhood of z_0 then have a simple structure: for some $t \in \mathbb{N}$, there exist linearly independent formal solutions of the form (with $z_0 = 0$)

$$y(z) = z^\lambda \sum_{k=0}^{t-1} \phi_k(z)\log^k z, \quad \lambda \in \mathbb{C}, \quad \phi_k \in \mathbb{C}[[z]] \qquad (12)$$

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 (12) in fact form a basis of analytic solutions on any open sector with vertex at the origin contained in this disk. (See for instance [28] for proofs and references.)

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 revisit Poole's variant [28, §16] of Heffter's method [22, Kap. 8] using the "operator algebra point of view" on indeterminate coefficients: a recursive formula for the coefficients of the series expansion of a solution is obtained by applying simple rewriting rules to the equation.

This formulation makes the algorithm simpler for our purposes (compare [31, 11, 33]) and leads to a small complexity improvement. It also proves helpful in error control (§6).

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 (12), 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 $\theta\cdot y(z) = \sum_{n\in\lambda+\mathbb{Z}}\sum_{k\geq 0}(ny_{n,k}+y_{n,k+1})\frac{\log^k z}{k!}z^n$. 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, (12) implies that $y_{n,k} = 0$ for $k \geq t$, which translates into $S_k^t\cdot\boldsymbol{y} = 0$. Set $L(z,\theta) = Q_0(\theta) + R(z,\theta)z$, 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

$$S_k^{\mu(n)}\cdot\boldsymbol{y} = -T_n(S_k)R(S_n^{-1}, n+S_k)S_n^{-1}\cdot\boldsymbol{y}, \qquad (13)$$

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 $y_{n,k} = 0$ for $n-\lambda < 0$ following from (12), this implies that a solution y is characterized by $(y_{n,k})_{(n,k)\in E}$, where $E = \{(n,k) \mid k < \mu(n)\}$. As in §3, this choice of initial values 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 [33, §4] for a detailed discussion.

Equation (13) 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 $Y_u = (0,\ldots,0,\log^v z)^{\mathrm{T}}$. Compared to the algorithm of [33], 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 $M_{z_0\to z_1}$ may (and should) be computed at once.

5. ERROR CONTROL

Performing numerical analytic continuation rigorously requires a number of bounds that control the behaviour of the function under consideration. We now describe how error control is done in NumGfun. Some of the ideas used in this section appear scattered in [32]–[36]. NumGfun relies almost exclusively on *a priori* bounds; see [27, §5.2] for pointers to alternative approaches, and [36] 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 to achieve this precision. Finally, we propose a way to limit the cost of computing bounds in "bit burst" phases.

Remark 2. In practice, numerical errors that happen while computing the error bounds themselves are not always controlled, due to limited support for interval arithmetic in Maple. However, we have taken some care to ensure that crucial steps rely on rigorous methods.

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

$$\begin{gathered} y(z)\leq g(|z|), \qquad \tfrac{\mathrm{d}}{\mathrm{d}z}y(z)\trianglelefteq\tfrac{\mathrm{d}}{\mathrm{d}z}g(z), \\ y+\hat{y}\trianglelefteq g+\hat{g}, \qquad y\hat{y}\trianglelefteq g\hat{g}, \qquad y\circ\hat{y}\trianglelefteq g\circ\hat{g}. \end{gathered} \qquad (14)$$

We shall allow majorant series to be formal logarithmic sums or matrices. The relation $\trianglelefteq$ extends in a natural way: we write $\sum_{n,k} y_{n,k}z^n\log z^k \trianglelefteq \sum_{n,k} g_{n,k}z^n\log z^k$ 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 (14) still hold.

Bounds on canonical solutions.

The core of the error control is a function that computes $g(z)$ such that

$$\forall j, \quad y[z_0,j](z_0+z)\trianglelefteq g(z) \qquad (15)$$

(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 [27] ("SB" in what follows), designed to compute "asymptotically tight" *symbolic* bounds. Run over $\mathbb{Q}(i)$ instead of $\mathbb{Q}$, SB returns (in the case of convergent series) a power series satisfying (15) of the form $g(z) = \sum_{n=0}^{\infty} n!^{-\tau}\alpha^n\phi(n)z^n$, where $\tau\in\mathbb{Q}_+$, $\alpha\in\bar{\mathbb{Q}}$, and $\phi(n) = e^{o(n)}$. The series g is specified by τ, α, and other parameters of no interest here that 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. For space reasons, we only sketch the differences between SB and the variant we use for error control.

First, we take advantage of (i) to solve (ii). The most important change is that the parameter α is replaced by

an approximation $\tilde{\alpha} \geq \alpha$. This avoids computations with algebraic numbers, the bottleneck of SB. Strictly speaking, it also means that we are giving up the tightness of the bound. However, in constrast with the "plain" majorant series method [33, 34, 36], 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 algorithms from [27] adapt without difficulty. Specifically, Algorithms 3 and 4 are modified to work with $\tilde{\alpha}$ instead of α. Equality tests between "dominant roots" (Line 9 of Algorithm 3, Line 5 of Algorithm 4) can now be done numerically and are hence much less expensive. As a consequence, the parameter T from §3.3 is also replaced by an upper bound. This affects only the parameter $\phi(n)$ of the bound, which is already pessimistic.

The issue (iii) can be dealt with too. One may compute a series g such that $y(z) \trianglelefteq g(z)\sum_{k=0}^{t-1} \frac{\log^k z}{k!}$ (with the notations of §4) by modifying the choice of K in [27, §3.3] so that $\sum_{k=0}^{t-1}\left|\left(\frac{\partial^k}{\partial X^k}\frac{X^{\mu(n)}}{Q_0(X)}\right)_{X=n}\right| \leq K/n^r$, and replacing [27, Eq. (14)] by Eq. (13) above in the reasoning.

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 $\diamond(\tilde{r}) \in \bigcup_{n\in\mathbb{N}} 10^{-n}\mathbb{Z}[i]$ 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 $\|AB\|_{\mathrm{F}} \leq \|A\|_{\mathrm{F}}\|B\|_{\mathrm{F}}$ for any matrices A, B of compatible dimensions. If A is a square $r \times r$ matrix,

$$\|A\|_\infty \leq |||A|||_2 \leq \|A\|_{\mathrm{F}} \leq r\|A\|_\infty \tag{16}$$

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 $\|\tilde{A}\tilde{B} - AB\|_{\mathrm{F}} \leq \|\tilde{A}\|_{\mathrm{F}}\|\tilde{B} - B\|_{\mathrm{F}} + \|\tilde{A} - A\|_{\mathrm{F}}\|B\|_{\mathrm{F}}$. From there, we compute precisions ϵ_A such 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 (14, 15).

Local error control. Let us turn to the computation of individual transition matrices. We restrict to the case of ordinary points. Given a precision ϵ determined by the "global error control" step, our task is to choose N such that $\|\tilde{M} - M_{z_0\to z_1}\|_{\mathrm{F}} \leq \epsilon$ if $\tilde{M}$ is computed by truncating the entries of (8) to the order N. If g satisfies (15), it suffices that $g^{(i)}_{N;}(|z_1 - z_0|) \leq \frac{i!}{r}\epsilon$ for all i, as can be seen from (14, 16). We find a suitable N by dichotomic search. For the family of majorant series g used in NumGfun, the $g^{(i)}_{N;}(x)$ are not always easy to evaluate, so we bound them by expressions involving only elementary functions [27, §4.2]. The basic idea, related to the saddle-point method from 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.

Points of large bit-size. Computing the majorants (15) 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 $\boldsymbol{Y}[z_0](z_0 + z) = \boldsymbol{Y}[c](c + (z_0 - c) + z)M^{-1}_{c\to z_0}$ by (7), it follows that $\boldsymbol{Y}[z_0](z_0 + z) \trianglelefteq G(|z_0 - c| + z)\,\|M^{-1}_{c\to z_0}\|\mathbf{1}$ 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 $\|M_{c\to z_0} - \mathrm{Id}\,\|_{\mathrm{F}} \leq \delta := \|G(\eta) - G(0)\|_{\mathrm{F}}$. Choosing η small enough that $\delta < 1$, we get $\|M^{-1}_{c\to z_0}\|_{\mathrm{F}} \leq (1 - \delta)^{-1}$. If G was computed from (15), *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, a similar reasoning 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 [32, §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 [35, §5].

6. REPEATED EVALUATIONS

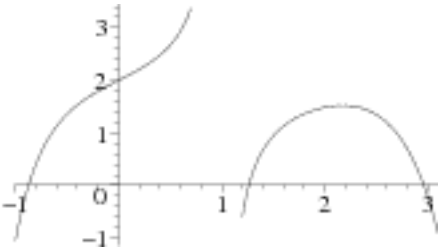
Drawing plots or computing integrals numerically requires to evaluate the same function at many points, often to a few digits of precision only. NumGfun provides limited support for this through the function `diffeqtoproc`, which takes as input a D-finite function y, a target precision ϵ, and a list of disks, and returns a Maple procedure that performs the numerical evaluation of y. For each disk $D = \{z, |z - c| < \rho\}$, `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 $\diamond$ again denotes rounding to complex decimal. 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. The way we choose expansion orders is similar to the error control techniques of §5: we first compute a bound B_{can} on the fundamental solutions and their first derivatives on the disk D. The vector $Y(c)$ of "initial values" at c is computed to the precision $\epsilon'/B_{\mathrm{can}}$ where $\epsilon' \leq \epsilon/(2r)$. We also compute $B_{\mathrm{ini}} \geq \|Y(c)\|_{\mathrm{F}}$. Each $y[c, j]$ is

expanded to an order N such that $\|y[c,j]_N;\|_{\infty,D} \le \epsilon'/B_{\text{ini}}$, so that finally $|p(z)-y(z)| \le \epsilon$ for $z \in D$.

The most important feature of `diffeqtoproc` is that it produces certified results. At low precisions and in the absence of singularities, we expect that interval-based numerical solvers will perform better while still providing (*a posteriori*) guarantees. Also note that our simple choice of p is far from optimal. If approximations of smaller degree or height are required, a natural approach is to aim for a slightly smaller error $\|y-p\|_{\infty,D}$ above, and then replace p by a polynomial $\tilde{p}$ for which we can bound $\|p-\tilde{p}\|_\infty$ [36, §6.2].

Example 6. The following plot of the function y defined by $(z-1)\,y'''-z(2z-5)\,y''-(4z-6)\,y'+z^2(z-1)\,y$ with the initial values $y(0)=2$, $y'(0)=1$, $y''(0)=0$ was obtained using polynomial approximations on several disks that cover the domain of the plot while avoiding the pole $z=1$. The whole computation takes about 9 s. Simple numerical integrators typically fail to evaluate y beyond $z=1$.



7. FINAL REMARKS

Not all of NumGfun was described in this article. The symbolic bounds mentioned in §5 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 [27].

Current work focuses on adding support for evaluation "at regular singular points" (as outlined in §4), and improving performance. The development version of NumGfun already contains a second implementation of binary splitting, written in C and called from the Maple code. In the longer term, I plan to rewrite other parts of the package "from the bottom up", both for efficiency reasons and to make useful subroutines independant of Maple.

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, bug reports, and/or comments on drafts of this article, and to the anonymous referees for helping make this paper more readable.

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

8. REFERENCES

[1] J. Abad, F. J. Gómez, and J. Sesma. An algorithm to obtain global solutions of the double confluent Heun equation. *Numer. 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. Computation of 2700 billion decimal digits of Pi using a Desktop Computer. 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 Trans. Math. Softw.*, 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). *J. 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. *J. Phys. A*, 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 Trans. Math. Softw.*, 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] `http://ddmf.msr-inria.inria.fr/`

[21] B. Haible and T. Papanikolaou. Fast multiprecision evaluation of series of rational numbers, 1997.

[22] L. Heffter. *Einleitung in die Theorie der linearen Differentialgleichungen*. Teubner, Leipzig, 1894.

[23] E. Jeandel. Évaluation rapide de fonctions hypergéométriques. Rapport technique RT-0242, INRIA-ENS Lyon, 2000.

[24] O. M. Makarov. An algorithm for multiplying 3×3 matrices. *Comput. Math. Math. Phys.*, 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. `http://algo.inria.fr/esf/`

[26] M. Mezzarobba. *Génération automatique de procédures numériques pour les fonctions D-finies*. Rapport de stage, Master parisien de recherche en informatique, 2007.

[27] M. Mezzarobba and B. Salvy. Effective bounds for P-recursive sequences. *J. Symbolic Comput.*, to appear. arXiv:0904.2452v2.

[28] E. G. C. Poole. *Introduction to the theory of linear differential equations*. The Clarendon Press, Oxford, 1936.

[29] B. Salvy and P. Zimmermann. Gfun: A Maple package for the manipulation of generating and holonomic functions in one variable. *ACM Trans. Math. Softw.*, 20(2):163–177, 1994. `http://algo.inria.fr/libraries/papers/gfun.html`

[30] R. P. Stanley. *Enumerative combinatorics*, volume 2. Cambridge University Press, 1999.

[31] É. Tournier. *Solutions formelles d'équations différentielles*. Thèse de doctorat, Université de Grenoble, 1987.

[32] J. van der Hoeven. Fast evaluation of holonomic functions. *Theoret. Comput. Sci.*, 210(1):199–216, 1999.

[33] J. van der Hoeven. Fast evaluation of holonomic functions near and in regular singularities. *J. Symbolic Comput.*, 31(6):717–743, 2001.

[34] J. van der Hoeven. Majorants for formal power series. Technical Report 2003-15, Université Paris-Sud, 2003.

[35] J. van der Hoeven. Efficient accelero-summation of holonomic functions. *J. Symbolic Comput.*, 42(4):389–428, 2007.

[36] J. van der Hoeven. On effective analytic continuation. *Mathematics in Computer Science*, 1(1):111–175, 2007.

[37] J. van der Hoeven. *Transséries et analyse complexe effective*. Mémoire d'habilitation, Université Paris-Sud, 2007.

[38] A. Waksman. On Winograd's algorithm for inner products. *IEEE Trans. Comput.*, C-19(4):360–361, 1970.

[39] 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.