Relax, but don't be too lazy

Joris van der Hoeven (C.N.R.S.)

Dept. de Mathématiques (bât. 425)
Université Paris-Sud
91405 Orsay Cedex, France
Email: Joris.VANDERHOEVEN@math.u-psud.fr

(16 November 1999)

Assume that we wish to expand the product $h = fg$ of two formal power series f and g. Classically, there are two types of algorithms to do this: *zealous* algorithms first expand f and g up to order n, multiply the results and truncate at order n. *Lazy* algorithms on the contrary compute the coefficients of f, g and h gradually and they perform no more computations than strictly necessary at each stage. In particular, at the moment we compute h_i only $f_0, \ldots, f_i$ and $g_0, \ldots, g_i$ are known.

Lazy algorithms have the advantage that the coefficients of f and g may actually depend on "previous" coefficients of h, as long as they are computed before they are needed in the multiplication. I.e. the coefficients f_i and g_i may depend on $h_0, \ldots, h_{i-1}$. For this reason, lazy algorithms are extremely useful when solving functional equations in rings of formal power series. However, lazy algorithms have the disadvantage that the classical asymptotically fast multiplication algorithms on polynomials — such as the divide and conquer algorithm and fast Fourier multiplication — can not be used.

In a previous paper, we therefore introduced *relaxed* algorithms, which share the property concerning the resolution of functional equations with lazy algorithms, but perform slightly more computations than lazy algorithms during the computation of a given coefficient of h. These extra computations anticipate the computations of the next coefficients of h and dramatically improve the asymptotic time complexities of such algorithms.

In this paper, we survey several classical and new zealous algorithms for manipulating formal power series, including algorithms for multiplication, division, resolution of differential equations, composition and reversion. Next, we give various relaxed algorithms for these operations. All algorithms are specified in great detail and we prove theoretical time and space complexity bounds. Most algorithms have been experimentally implemented in C++ and we provide benchmarks. We conclude by some suggestions for future developments and a discussion of the fitness' of the lazy and relaxed approaches for specific applications.

The paper is intended both for those who are interested in the most recent algorithms for the manipulation of formal power series and for those who want to actually implement a power series library into a computer algebra system.

1. Introduction

Let C be an effective ring, which means that we have algorithms for addition, subtraction and multiplication. In this paper, we describe several fast algorithms for manipulating formal univariate power series in the ring $\mathsf{C}[[z]]$. In principle, it is not necessary to assume that C is commutative or an integral domain. Nevertheless, for certain algorithms we need to assume that C contains the rational numbers, or more modestly, that C is divisible. Here C is said to be *divisible*, if we have a division algorithm for elements x in C by integers n, which raises an exception if x is not divisible by n.

Because of the infinite nature of formal power series, we will always be concerned with the computation of the first n coefficients of a given power series. The time and space complexities of our algorithms will be measured in terms of the number of ring operations in C and the number of elements in C stored in memory. Only in the case of finite rings, these complexity measures coincide with bitwise complexities.

1.1. The different approaches

Assume that we want to compute the first n coefficients of the product $h = fg$ of two power series f and g. We will distinguish three approaches in order to do this; the first and the second are classical, while the third one was (briefly) introduced in (van der Hoeven, 1997b). For simplicity, we discuss the approaches in the case of multiplication, but they apply to *any* operation on formal power series and in this paper, we will also consider composition, reversion, etc.

1.1.1. The zealous approach

This approach consists of expanding f and g up to order n, to multiply

$$(f_0 + \cdots + f_{n-1} z^{n-1})(g_0 + \cdots + g_{n-1} z^{n-1})$$

the results and truncate at order n. This yields the first n coefficients of h.

The advantage of the zealous approach is that we may compute the coefficients $h_0, \ldots, h_{n-1}$ *together* as a function of $f_0, \ldots, f_{n-1}$ and $g_0, \ldots, g_{n-1}$. Therefore, many fast algorithms on (truncated) polynomials can be used, such as divide and conquer and fast Fourier multiplication (shortly: DAC- and FFT-multiplication). In the cases of composition, reversion and resolution of differential equations, Brent and Kung's algorithms may be used. We will briefly recall some of these classical zealous algorithms in section 3 and a few new ones will be added.

1.1.2. The lazy approach

Another approach is to compute the coefficients of h one by one and to do no more work than strictly necessary at each stage. In particular, at stage i (i.e. for the computation of h_i), we compute only those coefficients of f and g which are really needed — that is $f_0, \ldots, f_i$ and $g_0, \ldots, g_i$.

Lazy algorithms have the advantage that the coefficients f and g may actually depend on "previous" coefficients of h, as long as they are computed before they are needed in the multiplication algorithm. In other words, f_i and g_i may depend on $h_0, \ldots, h_{i-1}$. For this reason, lazy algorithms are extremely useful for the resolution of functional equations. For instance, consider the formula

$$e^{\varphi} = \int \varphi' e^{\varphi}$$

for exponentiating a formal power series φ with $\varphi_0 = 0$. When evaluating the product fg lazily, where $f = \varphi'$ and $g = e^{\varphi}$, this formula yields a method to compute e^{φ}.

A second advantage of the lazy approach is that the computation process can be resumed in order to compute more than n coefficients of h. In the case of the zealous approach all coefficients would have to be recomputed.

1.1.3. The relaxed approach

Lazy algorithms have the disadvantage that the classical asymptotically fast algorithms on polynomials, such DAC- and FFT-multiplication, can no longer be used. This is what motivated us in the introduction of a slightly different, *relaxed* approach (van der Hoeven, 1997b).

Relaxed algorithms share with lazy algorithms the fact that the coefficients of h are computed gradually and that at each stage we only compute those coefficients of f and g which are needed for the computation of the next coefficient of h. In particular, relaxed algorithms can be used in a similar manner as lazy algorithms in order to solve functional equations.

The difference between the lazy and the relaxed approaches is that at the computation of a given coefficient of h lazy algorithms only perform "the strictly necessary operations", while relaxed algorithms "anticipate the computation of the next coefficients". Let us illustrate this by an example.

Assume that we want to compute the first three coefficients of the product of two power series $f = f_0 + f_1 z + f_2 z^2 + \cdots$ and $g = g_0 + g_1 z + g_2 z^2 + \cdots$. When using a lazy algorithm, we do the following:

0. We compute f_0, g_0 and $(fg)_0 = f_0 g_0$.
1. We compute f_1, g_1 and $(fg)_1 = f_0 g_1 + f_1 g_0$.
2. We compute f_2, g_2 and $(fg)_2 = f_0 g_2 + f_1 g_1 + f_2 g_0$.

Of course, the values of f_0 and g_0 are stored somewhere, so that they do not have to be reevaluated at stage 1 and similarly for f_1 and g_1 at stage 2. When using a relaxed algorithm, we would rather do the following:

0. We compute f_0, g_0 and $(fg)_0 = f_0 g_0$.
1. We compute f_1, g_1 and $(fg)_1 = (f_0 + g_0)(f_1 + g_1) - f_0 g_0 - f_1 g_1$.
2. We compute f_2, g_2 and $(fg)_2 = f_0 g_2 + f_1 g_1 + f_2 g_0$.

Here we used a trick in order to evaluate $(f_0 + f_1 z)(g_0 + g_1 z)$ using 3 multiplications only. Indeed, the three multiplications $f_0 g_0, (f_0 + g_0)(f_1 + g_1), f_1 g_1$ yield $(f_0 + f_1 z)(g_0 + g_1 z) = f_0 g_0 + ((f_0 + g_0)(f_1 + g_1) - f_0 g_0 - f_1 g_1) z + f_1 g_1 z^2$. Although we perform some extra additions at stage 1, we anticipate the computation of $f_1 g_1$ at stage 2. Consequently, we only perform 5 multiplications in total, against 6 for the lazy approach.

1.2. Outline of the paper

In section 3, we mainly recall classical zealous algorithms for manipulating formal power series. The corresponding complexity results are summarized in table 1. In the table, $M(n)$ denotes the time complexity for fast multiplication (see section 3.1); basic references for fast integer and polynomial multiplication algorithms are (Knuth, 1981; Nussbaumer, 1981) and (Karatsuba and Ofman, 1962; Toom, 1963b; Cooley and Tukey, 1965; Cook, 1966; Schönhage and Strassen, 1971; Cantor and Kaltofen, 1991; Heideman *et al.*, 1984). Most of the remaining results are due to Brent and Kung (Brent and Kung, 1975; Brent and Kung, 1978). The result about general composition in finite characteristic is due to Bernstein (Bernstein, 1998).

Algorithm	Time complexity	Space complexity
DAC-multiplication	$M(n) = O(n^{\log 3/\log 2})$	$O(n)$
FFT-multiplication	$M(n) = O(n \log n)$	$O(n)$
Division	$O(M(n))$	$O(n)$
Solving implicit equations and o.d.e.'s	$O(M(n))$	$O(n)$
Algebraic and holonomic functions	$O(n)$	$O(n)$
Right composition with polynomials	$O(M(n) \log n)$	$O(n)$
Right composition with algebraic functions	$O(M(n) \log n)$	$O(n)$
Composition and reversion (divisible ring C)	$O(M(n)\sqrt{n \log n})$	$O(n \log n)$
Composition and reversion (finite ring C)	$O(M(n) \log n)$	$O(n)$

Table 1. Time and space complexities of zealous algorithms.

Although most algorithms from section 3 are classical, we have given several variants which we could not find in the standard literature:

- In section 3.1.2 we give a digest fast multiplication algorithm for polynomials using the FFT-transform. This algorithm is an analogue of Schönhage-Strassen's algorithm and simplifies Cantor and Kaltofen's algorithm for the frequent case when 2 does not divide zero in C.
- In section 3.2.4, we specify and prove Brent and Kung's algorithm for the resolution of o.d.e.'s for general orders. In the original papers, only first and second order equations were considered and the latter only by means of examples.
- In section 3.4.2, we observe that Brent and Kung's method for right composition with polynomials generalizes to right composition with rational and algebraic functions; in the relaxed case, this observation will be useful for solving certain difference equations.

In section 4, we propose several relaxed multiplication algorithms. We first observe that the divide and conquer algorithm can easily be transformed into a relaxed algorithm with the same time complexity but a logarithmic space overhead. We next give an asymptotically better algorithm, which also has the best possible space complexity. However, this algorithm may be slower for small input sizes, which makes it difficult to choose a best overall strategy (see section 4.4). Some examples of problems to which relaxed multiplication can be applied are given in section 4.5.

In section 5, we give algorithms for relaxed composition. Actually, Brent and Kung's and Bernstein's algorithms can easily be adapted to this case, while preserving the same time complexity (modulo replacing a zealous multiplication algorithm by a relaxed one).

An overview of our complexity results for relaxed algorithms is given in table 2, where $M^*(n)$ stands for the time complexity of relaxed multiplication. The space complexities assume that we use a relaxed multiplication with linear time complexity; when using relaxed DAC-multiplication, these complexities should be multiplied by $\log n$.

In section 6 we suggest how to improve the performance of the algorithms for particular coefficient rings. In section 6.1, we outline how to take more advantage of the FFT-transform. In section 6.2, we study the numerical stability of our algorithms. In section 6.3, we discuss the issue of multivariate power series. Several approaches will be

Algorithm	Time complexity	Space complexity
Relaxed DAC-multiplication	$M^*(n) = O(n^{\log 3/\log 2})$	$O(n \log n)$
Fast relaxed multiplication	$M^*(n) = O(M(n) \log n)$	$O(n)$
Division	$O(M^*(n))$	$O(n)$
Algebraic and holonomic functions	$O(n)$	$O(n)$
Right composition with rational functions	$O(M^*(n) \log n)$	$O(n)$
Right composition with algebraic functions	$O(M^*(n) \log n)$	$O(n)$
Composition and reversion (divisible ring C)	$O(M^*(n)\sqrt{n \log n})$	$O(n\sqrt{n} \log n)$
Composition and reversion (finite ring C)	$O(M^*(n) \log n)$	$O(n \log n)$

Table 2. Time and space complexities of relaxed algorithms.

proposed in an informal style and the development of a more detailed theory remains a challenge.

Most of the algorithms in this paper have been implemented in an experimental C++-package and we have included several tables with benchmarks. In the last section 7, we draw some final conclusions and discuss the relevance of the different algorithms presented in this paper for specific applications such as symbolic computation, combinatoris, the analysis of algorithms and numerical analysis. We also give some suggestions for those who want to implement a power series library in a computer algebra system and for those who want to "upgrade" an existing lazy power series implementation.

2. Implementation conventions

We have presented most of the algorithms in this paper in detail in the hope that this will be helpful for actual implementations. For our specifications, we have chosen an object oriented pseudo-language (see (Stroustrup, 1995) for some basic terminology for such languages), with explicit memory control for the user (this will allow us to study in detail the space and time complexities of the relaxed algorithms). Below, we will discuss some general implementation issues and fix some notational conventions.

2.1. Zealous algorithms

Truncated power series will be represented by elements of the class $\mathsf{TPS}(\mathsf{C})$. An instance of this class consists of a pointer to an array of elements in C and the length n of the array; it represents a truncated power series at order n, i.e. an element of $\mathsf{C}[[z]]/(z^n) \cong \mathsf{C}[z]/(z^n)$. For notational convenience, we will also denote by $\mathsf{TPS}(\mathsf{C}, n)$ the "subclass" of instances in $\mathsf{TPS}(\mathsf{C})$ with truncation order n.

We will use the following shorthands for the most elementary operations on truncated power series $f = f_0 + \cdots + f_{n-1} z^{n-1}$:

- $\sharp f$: the order n of f.
- C, z: implicit conversion to instances of $\mathsf{TPS}(\mathsf{C}, n)$.
- $+, -$: addition resp. subtraction.
- f': derivative $f_1 + \cdots + (n-1) f_{n-1} z^{n-2}$.
- $\int f$: integral $f_0 z + \cdots + \frac{1}{n} z^n$ (for rings C which contain the rationals).

- $f \circ z^p$: right composition $f_0 + \cdots + f_{(n-1)p} z^{(n-1)p} + 0 z^{(n-1)p+1} + \cdots + 0 z^{np-1}$ with power of z.
- $f \ll k$: multiplication $f_0 z^k + \cdots + f_{n+k-1} z^{n+k-1}$ with z^k.
- $f \gg k$: division $f_k + \cdots + f_{n-1} z^{n-k-1}$ by z^k.
- $f_{i \cdots j}$ $(j \leqslant n)$: the truncated series $f_i + \cdots + f_{j-1} z^{j-i}$.
- $f_{0 \cdots m}$ $(m > n)$: the truncated series $f_0 + \cdots + f_{n-1} z^{n-1} + 0 z^n + \cdots + 0 z^{m-1}$.
- $f_{i \cdots j} \mathrel{+}= g$: sets $f_i := f_i + g_0, \ldots, f_{j-1} := f_{j-1} + g_{j-i-1}$.

We will not detail the implementation of these operations and assume that memory management is taken care of. Notice that the operations all require linear time and space.

In section 3.1, we will recall algorithms for the fast multiplication of dense polynomials. Modulo truncation, this also yields a multiplication algorithm in $\mathsf{TPS}(\mathsf{C}, n)$, as well as a binary powering algorithm. In section 3.2.1, we give a fast division algorithm in $\mathsf{TPS}(\mathsf{C}, n)$. In the sequel we use the following abbreviations for these operations:

- $\star : \mathsf{TPS}(\mathsf{C}, p) \times \mathsf{TPS}(\mathsf{C}, q) \to \mathsf{TPS}(\mathsf{C}, p+q-1)$ stands for polynomial multiplication.
- $\times : \mathsf{TPS}(\mathsf{C}, n) \times \mathsf{TPS}(\mathsf{C}, n) \to \mathsf{TPS}(\mathsf{C}, n)$ stands for truncated multiplication. Notice that $f \times g = (f \star g)_{0 \cdots n}$.
- $/ : \mathsf{TPS}(\mathsf{C}, n) \times \mathsf{TPS}(\mathsf{C}, n) \to \mathsf{TPS}(\mathsf{C}, n)$ stands for division.
- $\cdot^p : \mathsf{TPS}(\mathsf{C}, n) \to \mathsf{TPS}(\mathsf{C}, n)$ stands for binary powering.

Remark. In low level languages, operations on truncated power series can be implemented more efficiently by routines which take directly pointers to the destination and argument segments on input as well as their lengths. This approach also avoids memory allocations, except for temporary ones on the heap. Nevertheless, it should not be hard to rewrite the algorithms from this paper in this style.

2.2. Lazy and relaxed algorithms

From the point of view of the user a lazy or relaxed power series f should be some object with a method which yields the coefficients of f one by one. In object oriented languages, we may therefore implement a series as a pointer to an abstract "series representation class" Series_Rep, which is given by

Class Series_Rep(C)

  φ : TPS(C)
  n : Integer
  virtual next : Void $\to$ C

Here φ contains the already computed coefficients and n their number. *The order of φ is allowed to exceed n in order to anticipate future computations.* The virtual method **next** is private and should compute the next, n-th, coefficient of the series. The public method to compute the n-th coefficient, which is detailed below, ensures that the coefficients $\varphi_0, \ldots, \varphi_{n-1}$ are already available before calling **next** and that φ_n is updated after calling **next**.

All representation classes in this paper, like Series_Rep will contain a reference counter, which is increased each time an instance is copied and decreased each time a copy is deleted. The instance is physically removed, only when the reference counter vanishes. We will use $p := \mathbf{new}\ \mathsf{D\_Rep}(a_1, \ldots, a_k)$ to create a pointer p to a concrete class $\mathsf{D\_Rep}$ and to call the corresponding constructor with arguments $a_1, \ldots, a_k$. A member or member function x of $\mathsf{D\_Rep}$ will be accessed through $p.x$. We denote by **null** the symbolic "null" pointer.

Given a pointer $f : \mathsf{Series}(\mathsf{C})$ to an instance of $\mathsf{Series\_Rep}(\mathsf{C})$, let us now detail the algorithm to compute the n-th coefficient f_n of f. We first look whether $n < f.n$. If so, then we return $f.\varphi_n$. Otherwise, we increase the order of φ to $n+1$ (if necessary), compute the coefficients $f_{f.n}, \ldots, f_n$ by repeatedly calling $f.\mathbf{next}()$, and return $f.\varphi_n$. We also implement an algorithm to compute $f_{i \cdots j} = f_i + \cdots + f_{j-1} z^{j-i-1} : \mathsf{TPS}(\mathsf{C}, j-i)$. This algorithm first computes f_{j-1} and then returns $f.\varphi_{i \cdots j}$.

EXAMPLE. In order to implement a constant series, we first define a concrete class

CLASS $\mathsf{Constant\_Series\_Rep}(\mathsf{C}) \triangleright \mathsf{Series\_Rep}(\mathsf{C})$

$c \quad : \mathsf{C}$

The symbol $\triangleright$ stands for class inheritance. The constructor for $\mathsf{Constant\_Series\_Rep}(\mathsf{C})$ takes a constant $c' : \mathsf{C}$ on input and sets $c := c'$. The member function **next** returns c if $n = 0$ and 0 otherwise. See section 4.1 for another easy and detailed example.

The following easily implemented operations on series will not be specified out:

- Conversion from $\mathsf{TPS}(\mathsf{C})$ to $\mathsf{Series}(\mathsf{C})$, where we fill up with zero coefficients.
- Addition and subtraction $+, -$.
- Differentiation and integration $', \int$.
- $f \ll k, f \gg k$ multiplication and division by z^k.

3. Zealous algorithms

3.1. MULTIPLICATION

There are several well-known algorithms to multiply two polynomials $f = f_0 + \cdots + f_{n-1} z^{n-1}$ and $g = g_0 + \cdots + g_{n-1} z^{n-1}$ of degrees $< n$ with coefficients in an effective ring C. The naive algorithm, based on the formula $(fg)_k = \sum_i f_i g_{k-i}$, has complexity $O(n^2)$. Below, we recall DAC-, FFT- and truncated multiplication.

In the remainder of this paper, we assume that we have fixed once and for all a multiplication method of time complexity $M(n)$, such that $M(n)/n$ is an increasing function of n and $M(O(n)) = O(M(n))$.

3.1.1. DAC-MULTIPLICATION

Given polynomials $f = f_0 + \cdots + f_{n-1} z^{n-1}$ and $g = g_0 + \cdots + g_{n-1} z^{n-1}$, we define their *lower* and *higher* parts by $f_* = f_0 + \cdots + f_{\lceil n/2 \rceil - 1} z^{\lceil n/2 \rceil - 1}$ resp. $f^* = f_{\lceil n/2 \rceil} z^{\lceil n/2 \rceil} + \cdots + f_{n-1} z^{n-1}$ and similarly for g. Hence, f and g decompose as

$$\begin{aligned} f &= f_* + f^* z^{\lceil n/2 \rceil}; \\ g &= g_* + g^* z^{\lceil n/2 \rceil}, \end{aligned}$$

Now we have the following identity (a similar, but slightly more complicated identity was first found by Karatsuba (Karatsuba and Ofman, 1962)):

$$\begin{aligned} fg &= f_* g_* \\ &+ ((f_* + f^*)(g_* + g^*) - f_* g_* - f^* g^*) z^{\lceil n/2 \rceil} \\ &+ f^* g^* z^{2\lceil n/2 \rceil}. \end{aligned} \tag{3.1}$$

Applying this formula recursively, except for small $n < \mathit{Threshold}_{\mathsf{C}}$ (with $\mathit{Threshold}_{\mathsf{C}} \geqslant 2$), we obtain the DAC-multiplication algorithm below. Since the multiplication of two polynomials of degrees $< n$ involves only three multiplications of polynomials of degrees $< \lceil n/2 \rceil$, the asymptotic time complexity of this algorithm is $O(n^{\log 3/\log 2})$.

ALGORITHM **DAC_multiply**(f, g)

INPUT: Polynomials $f = f_0 + \cdots + f_{n-1} z^{n-1}$ and $g = g_0 + \cdots + g_{n-1} z^{n-1}$ in $\mathsf{C}[z]$.
OUTPUT: Their product fg.

D1. [Base]
if $n < \mathit{Threshold}_{\mathsf{C}}$ **then return** $\sum_{i=0}^{2n-2} \left(\sum_{j=\max(0,i+1-n)}^{\min(n-1,i)} f_j g_{i-j} \right) z^i$

D2. [Divide and conquer]
lo := **DAC_multiply**(f_*, g_*)
mid := **DAC_multiply**$(f_* + f^*, g_* + g^*)$
hi := **DAC_multiply**(f^*, g^*)
return $lo + mid \times z^{\lceil n/2 \rceil} + hi \times z^{2\lceil n/2 \rceil}$

3.1.2. FFT-MULTIPLICATION

The fastest known multiplication algorithm is based on the discrete Fourier transform (DFT). We recall that the DFT transforms a sequence of coefficients $a_0, \ldots, a_{n-1}$ in C and an n-th root of unity ω in C (which is assumed to exist) into the sequence of evaluations of the polynomial $a_0 + a_1 z + \cdots + a_{n-1} z^{n-1}$ in the n-th roots of unity $1, \omega, \ldots, \omega^{n-1}$. The trick is that applying the DFT twice w.r.t. ω and $\omega^{-1} = \omega^{n-1}$, we obtain n times the original sequence $a_0, \ldots, a_{n-1}$. Moreover, if n is a power of two, then the DFT can be performed in almost linear time $O(n \log n)$ by the following recursive algorithm (in practice, when multiplication in C is fast, the recursion should rather be transformed into a double loop):

ALGORITHM **DFT**(a, ω)

INPUT: An n-tuple $(a_0, \ldots, a_{n-1})$ and an n-th root of unity in C, where $n = 2^p$;
OUTPUT: The n-tuple $(\hat{a}_0, \ldots, \hat{a}_{n-1})$ with $\hat{a}_j = \sum_{i=0}^{n-1} a_i \omega^{ij}$.

if $n = 1$ **then return** (a_0)
$(\hat{b}_0, \ldots, \hat{b}_{n/2-1})$:= **DFT**$((a_0, a_2, \ldots, a_{n-2}), \omega^2)$
$(\hat{c}_0, \ldots, \hat{c}_{n/2-1})$:= **DFT**$((a_1, a_3, \ldots, a_{n-1}), \omega^2)$
return $(\hat{b}_0 + \hat{c}_0, \ldots, \hat{b}_{n/2-1} + \hat{c}_{n/2-1} \omega^{n/2-1}, \hat{b}_0 - \hat{c}_0, \ldots, \hat{b}_{n/2-1} - \hat{c}_{n/2-1} \omega^{n/2-1})$

Now assume that we want to multiply two polynomials $A = a_0 + \cdots + a_{n-1} z^{n-1}$ and $B = b_0 + \cdots + b_{n-1} z^{n-1}$ with $\deg AB < n = 2^p$. We first apply **DFT** to $(a_0, \ldots, a_{n-1})$

resp. $(b_0, \ldots, b_{n-1})$ and ω. Denoting by $\hat{a}_0, \ldots, \hat{a}_{n-1}$ resp. $\hat{b}_0, \ldots, \hat{b}_{n-1}$ the results, we next apply **DFT** to $(\hat{a}_0\hat{b}_0, \ldots, \hat{a}_{n-1}\hat{b}_{n-1})$ and ω^{n-1}. This yields n times the sequence of coefficients of AB. Assuming that C is *2-divisible* (i.e. we have an algorithm to divide the multiples of two in C by two), we can finally retrieve AB from this sequence, since n is a power of two.

If C does not contain an n-th root of unity, then it is still possible to use the fast Fourier transform, using a trick due to Schönhage and Strassen (Schönhage and Strassen, 1971). Actually, assuming that $n \geqslant \mathit{FFT\_Threshold}_{\mathsf{C}}$ is a sufficiently large power of two, we will show how to multiply efficiently in the "cyclotomic polynomial ring" $\mathsf{C}[x]/(x^n + 1)$; this method will then be used to multiply polynomials $A, B \in \mathsf{C}[z]$ for which $\deg AB < n$. Notice that x is a $2n$-th root of unity in $\mathsf{C}[x]/(x^n + 1)$.

Decompose $n = md$ with $m = 2^{\lceil (p+1)/2 \rceil}$. Then any polynomial

$$\sum_{i=0}^{n-1} c_i x^i$$

in $\mathsf{C}[x]/(x^n + 1)$ may be rewritten as a polynomial

$$\sum_{j=0}^{d-1} \left(\sum_{i=0}^{m-1} a_{di+j} y^i \right) x^j.$$

where $y = x^d$ is an $2m$-th root of unity. In other words, it suffices to show how to multiply polynomials of degrees $\leqslant d$, whose coefficients lie in the smaller cyclotomic polynomial ring $\mathsf{C}[y]/(y^m + 1)$. But we may use the DFT for this, since $\mathsf{C}[y]/(y^m + 1)$ contains $2m$-th roots of unity and $d \leqslant m$. Notice that the DFT only involves additions and copying in C, since the multiplications by powers of y in $\mathsf{C}[y]/(y^m + 1)$ involve only copying, additions and subtractions. Finally, we may apply the method recursively in order to multiply elements of $\mathsf{C}[y]/(y^m + 1)$. This gives us the following general multiplication algorithm:

ALGORITHM **FFT_multiply**(A, B)

INPUT: Polynomials $A = a_0 + \cdots + a_{n-1}x^{n-1}$ and $B = b_0 + \cdots + b_{n-1}x^{n-1}$ in $\mathsf{C}[x]/(x^n + 1)$, where $n = 2^p$.

OUTPUT: Their product AB.

F1. [Base]
 if $n \geqslant \mathit{FFT\_Threshold}_{\mathsf{C}}$ **then go to F2**
 $C :=$ **DAC_multiply**$(a_0 + \cdots + a_{n-1}z^{n-1}, b_0 + \cdots + b_{n-1}z^{n-1})$,
 Denote $C = c_0 + \cdots + c_{2n-1}z^{2n-1}$
 return $(c_0 - c_n) + \cdots + (c_{n-1} - c_{2n-1})x^{n-1}$

F2. [Encode]
 $m := 2^{\lfloor (p+1)/2 \rfloor}, d := n/m, y := x^d$
 for $j := 0$ **to** $d - 1$ **do**
  $A_j := \sum_{i=0}^{m-1} a_{di+j} y^i$
  $B_j := \sum_{i=0}^{m-1} b_{di+j} y^i$

 for $j := d$ **to** $2d - 1$ **do** $A_j := B_j := 0$

F3. [FFT]
$\omega := y^{m/d}, \overline{\omega} := \omega^{2d-1}$
$(\hat{A}_0, \ldots, \hat{A}_{2d-1}) := \mathbf{DFT}((A_0, \ldots, A_{2d-1}), \omega)$
$(\hat{B}_0, \ldots, \hat{B}_{2d-1}) := \mathbf{DFT}((B_0, \ldots, B_{2d-1}), \omega)$
for $j := 0$ **to** $2d-1$ **do** $\hat{C}_j :=$ **FFT_multiply**$(\hat{A}_j, \hat{B}_j)$
$(C_0, \ldots, C_{2d-1}) := \mathbf{DFT}((\hat{C}_0, \ldots, \hat{C}_{2d-1}), \overline{\omega})$

F4. [Decode]
return $\frac{C_0+C_d y}{2d} + \frac{C_1+C_{d+1} y}{2d}\omega_n + \cdots + \frac{C_{d-1}+C_{2d-1} y}{2d}\omega_n^{n-1}$

It can be shown that this algorithm has time complexity $O(n \log n \log \log n)$ and space complexity $O(n)$. The algorithm is a simplified version of the algorithm from (Cantor and Kaltofen, 1991), which also works when C is not 2-divisible (in this case, one may for instance compute both $2^p AB$ and $3^q AB$, using a similar, tricotomic FFT-multiplication algorithm, and then apply the Chinese remainder theorem).

3.1.3. Truncated multiplication

When multiplying formal power series f and g up to order n, we are usually only interested in the first n coefficients of fg. In other words, although multiplying $f_0 + \cdots + f_{n-1} z^{n-1}$ and $g_0 + \cdots + g_{n-1} z^{n-1}$ as polynomials and truncating the product does the job, it might be possible to find a faster algorithm, which does not perform superfluous computations.

When we use the naive multiplication algorithm, we may indeed gain a factor of two by evaluating only the products $(fg)_k \sum_{i=0}^{k} f_i g_{k-i}$ for $k < n$. We can also have a truncated DAC-multiplication: first compute $f_* g_*$ using the usual algorithm and next recursively compute $f_* g^*$ and $f^* g_*$ modulo $z^{\lfloor n/2 \rfloor}$. Finally, apply the formula

$$fg \mod z^n = [f_* g_* \mod z^n] + [f_* g^* \mod z^{\lfloor n/2 \rfloor}] z^{\lceil n/2 \rceil} + [f^* g_* \mod z^{\lfloor n/2 \rfloor}] z^{\lceil n/2 \rceil}.$$

Although this algorithm has the same asymptotic complexity (and the same constant factor), we do gain for moderate values of n, since less additions and subtractions are needed. However, when using FFT-multiplication, the constant in the asymptotic complexity becomes worse for this method.

3.2. Applications of Newton's method

Many zealous algorithms for operations on formal power series are based on Newton's method, which doubles the number of correct coefficients at each iteration. The method can in particular be used for division, reversion, exponentiation and the resolution of ordinary differential equations.

3.2.1. Division

In this section, we will give an algorithm to invert a power series f, such that f_0 is invertible in C; this clearly yields a division algorithm too. In order to invert f, we have to solve the equation

$$fg - 1 = 0.$$

If g is an approximate solution whose first $n > 0$ terms are correct, then the modified Newton iteration

$$g := g - \frac{fg - 1}{z^n} g z^n. \tag{3.2}$$

yields $2n$ correct terms. Indeed, if $g = f^{-1} + O(z^n)$, then we have $fg = 1 + O(z^n)$, whence $f(g - (fg-1)g) = 1 - (fg-1)^2 = 1 + O(z^{2n})$ and $g - (fg-1)g = f^{-1}(f(g-(fg-1)g)) = f^{-1} + O(z^{2n})$. Furthermore, $(fg-1)g = ((fg-1)/z^n) g z^n$, since the first n terms of $fg-1$ vanish. Notice that the real Newton iteration would be

$$g := g - \frac{fg - 1}{f}.$$

In fact, we obtain (3.2) when approximating f^{-1} by g in this iteration. Using the iteration (3.2), we get the following inversion algorithm of time complexity $O(M(n))$:

ALGORITHM **invert**(f)

INPUT: $f : \mathsf{TPS}(\mathsf{C}, n)$, such that f_0 is invertible in C.
OUTPUT: $f^{-1} : \mathsf{TPS}(\mathsf{C}, n)$

if $n = 1$ **then return** $(1/f_0)_{0\cdots 1}$.
$m := \lceil \frac{n}{2} \rceil$
$g :=$ **invert**$(f_{0\cdots m})_{0\cdots n}$
return $g - ((((f \times g) \gg m) \times g_{0\cdots n-m}) \ll m)$

3.2.2. EXPONENTIATION AND LOGARITHM

Assume that C is a ring which contains the rational numbers and that f is a power series, such that f_0 is invertible and $\log f_0$ well defined in C. Then the inversion algorithm also yields a straightforward way to compute $\log f$, since

$$\log f = \log f_0 + \int \frac{f'}{f},$$

where the integral is taken with integration constant zero. Solving the equation

$$\log g = f$$

using Newton's method, we also have the following algorithm for exponentiation, which again has time complexity $O(M(n))$:

ALGORITHM **exp**(f)

INPUT: $f : \mathsf{TPS}(\mathsf{C}, n)$, such that $\exp f_0$ is defined and invertible in C.
OUTPUT: $\exp f : \mathsf{TPS}(\mathsf{C}, n)$

if $n = 1$ **then return** $(\exp f_0)_{0\cdots 1}$.
$m := \lceil \frac{n}{2} \rceil$
$g :=$ **exp**$(f_{0\cdots m})_{0\cdots n}$
return $g - (\mathbf{log}(g) - f) \times g$

3.2.3. Reversion

If we have an algorithm **compose** for the composition of power series with time complexity $O(C(n))$ (where $C(n)/n$ is an increasing function), then Newton's method can still be applied in order to solve the equation

$$f \circ g - z = 0.$$

This yields the following $O(C(n))$ reversion algorithm for f:

Algorithm **revert**(f)
Input: $f : \mathsf{TPS}(\mathsf{C}, n)$ with $f_0 = 0$ and f_1 is invertible in C.
Output: f^{inv}

if $n = 1$ **then return** $0_{0\cdots 1}$
if $n = 2$ **then return** $(z/f_1)_{0\cdots 2}$
$m := \lceil \frac{n}{2} \rceil$
$g :=$ **revert**$(f_{0\cdots m})_{0\cdots n}$
$N :=$ **compose**$(f, g) - z$
$D :=$ **compose**$(f', g_{0\cdots n-1})$
return $g - (((N \gg 1)/D) \ll 1)$

3.2.4. Resolution of ordinary differential equations

In this section, we assume that C contains the rational numbers. Let $\Phi(y_0, \ldots, y_r, z)$ be a multivariate polynomial in $\mathsf{C}[y_0, \ldots, y_r, z]$. We wish to solve the ordinary differential equation

$$\Phi(f(z), f'(z), \ldots, f^{(r)}(z), z) = 0, \tag{3.3}$$

where we assume that the *separant* of Φ is invertible in C for the initial conditions:

$$\frac{\partial \Phi}{\partial y_r}(f(0), \ldots, f^{(r)}(0), 0) \in \mathsf{C}^*. \tag{3.4}$$

This condition ensures that (3.3) admits a unique formal solution. Indeed, modulo one differentiation of (3.3), we may assume without loss of generality that Φ is linear in y_r:

$$\Phi = \Phi_0(y_0, \ldots, y_{r-1}, z) + \Phi_1(y_0, \ldots, y_{r-1}, z) y_r.$$

Now (3.4) means that $\Phi_1(0, \ldots, 0)$ is invertible in C. Hence (3.3) may be solved formally by repeated integration:

$$f = -\int \overset{r \text{ times}}{\cdots} \int \frac{\Phi_0(f, \ldots, f^{(r-1)}, z)}{\Phi_1(f, \ldots, f^{(r-1)}, z)}, \tag{3.5}$$

where the integration constants are taken appropriately, so that they match the initial conditions.

The repeated integral (3.5) is useful for solving (3.3) by lazy or relaxed evaluation. In this section we show that (3.3) can also be solved using Newton's method. For this, we assume that we have implemented an $O(M(n))$ algorithm **subst**, which takes a polynomial $\Psi \in \mathsf{C}[y_0, \ldots, y_r, z]$ and a truncated power series $f : \mathsf{TPS}(\mathsf{C}, n)$ on input and which returns the first n terms of $\Psi(f, \ldots, f^{(r)}, z)$ (where we take $f_n = \cdots = f_{n+r-1} = 0$).

Now let $n > 3r$ and assume that f is an approximate solution (at order n) to (3.3) with

$$\Phi(f, \ldots, f^{(r)}, z) = O(z^{n-r}).$$

Then the Newton iteration consists of replacing

$$f := f - \varphi,$$

where φ is the unique solution to the linear differential equation

$$\left\{ \begin{array}{rcl} L\varphi &=& g; \\ L &=& \dfrac{\partial \Phi}{\partial y_0}(f(z), \ldots, f^{(r)}(z)z) + \cdots + \dfrac{\partial \Phi}{\partial y_r}(f(z), \ldots, f^{(r)}(z)z)\dfrac{\partial^r}{\partial z^r}; \\ g &=& \Phi(f, \ldots, f^{(r)}, z), \end{array} \right. \tag{3.6}$$

with $\varphi_0 = \cdots = \varphi_{r-1} = 0$. Notice that the existence and uniqueness of φ again follows from condition (3.4). Notice also that $\varphi = O(z^{n-r})$. Therefore,

$$\begin{array}{rl} & \Phi(f - \varphi, \ldots, (f - \varphi)^{(r)}, z) \\ = & \displaystyle\sum_{k_0, \ldots, k_r} \frac{(-1)^{k_0 + \cdots + k_r}}{k_0! \cdots k_r!} \frac{\partial^{k_0 + \cdots + k_r}}{\partial y_0^{k_0} \cdots \partial y_r^{k_r}}(f, \ldots, f^{(r)}, z)\varphi^{k_0} \cdots (\varphi^{(r)})^{k_r} \\ = & g - L\varphi + O(z^{(2n-3r)-r}). \end{array}$$

Hence, repeating the Newton iteration, the sequence of successive approximations tends to the unique solution to (3.3). Modulo an algorithm **linear** to solve (3.6), this yields the following algorithm:

ALGORITHM **ode**(Φ, f, n)

INPUT: A polynomial $\Phi \in \mathsf{C}[y_0, \ldots, y_r, z]$, an approximation $f : \mathsf{TPS}(\mathsf{C}, r+1)$ to a solution to (3.3), so that (3.4) is satisfied, and an order $n > r$.

OUTPUT: A better approximation $f : \mathsf{TPS}(\mathsf{C}, n)$ of the unique solution to (3.3), with $\Phi(f, \ldots, f^{(r)}, z) = O(z^{n-r})$.

O1. [Separate cases]
  $m := \lceil \frac{n+3r}{2} \rceil$
  if $n > m$ **and** $r = 0$ **then go to** step 3
  if $n > m$ **and** $r \neq 0$ **then go to** step 4

O2. [Compute first coefficients]
  for $i := r+1$ **to** $n-1$ **do**
    for $j := 0$ **to** r **do** $D_j :=$ **subst**$(\frac{\partial \Phi}{\partial y_j}, f)_{0 \cdots i-r}$
    $S := D_0 \times f'_{0 \cdots i-r} + \cdots + D_{r-1} \times f^{(r)}_{0 \cdots i-r} +$ **subst**$(\frac{\partial \Phi}{\partial z}, f)_{0 \cdots i-r}$
    $t := -S/D_r$
    $f := f_{0 \cdots i+1} + \frac{i!}{(i-r-1)!} t_{i-r-1} z^i$

  return $f_{0 \cdots n}$

O3. [Newton iteration when $r = 0$]
  $f :=$ **ode**$(\Phi, f, m)_{0 \cdots n}$
  return $f -$ **subst**$(\Phi, f)/$**subst**$(\frac{\partial \Phi}{\partial y_0}, f)$

O4. [Newton iteration when $r \neq 0$]
$f := \mathbf{ode}(\Phi, f, m)_{0\cdots n}$
$L := \mathbf{subst}(\frac{\partial \Phi}{\partial y_0}, f) + \cdots + \mathbf{subst}(\frac{\partial \Phi}{\partial y_r}, f)\frac{\partial^r}{\partial z^r}$
return $f - \mathbf{linear}(L, \mathbf{subst}(\Phi, f), n)$

In order to solve (3.6) up till n terms, we first compute a non trivial solution to the homogeneous differential equation

$$Lh = 0.$$

This is done by solving the associated Ricatti equation. More precisely, we rewrite each $h^{(k)}$ as h times a polynomial $R_k(\hat{h}, \ldots, \hat{h}^{k-1})$ in the logarithmic derivative $\hat{h} = h'/h$ of h. This amounts to computing the sequence

$$\begin{cases} R_0 = 1; \\ R_{k+1} = y_0 R_k + \frac{\partial R_k}{\partial y_0} y_1 + \cdots + \frac{\partial R_k}{\partial y_{k-1}} y_k, \end{cases} \tag{3.7}$$

with $R_k \in \mathsf{C}[[y_0, \ldots, y_{k-1}]]$ up till order r. We now compute the first n terms of $\hat{h}$ by a recursive application of **ode** with equation

$$R = L_0 R_0 + \cdots + L_r R_r \in \mathsf{C}[[y_0, \ldots, y_{r-1}, z]]$$

and initial conditions $\hat{h}_0 = \cdots = \hat{h}_{r-1} = 0$. This yields the first n terms of a solution h to (3.6) with $h_0 = 1$ after exponentiation and integration $h = \exp \int \hat{h}$.

Finally, we apply the method of variation of constants and write $\varphi = \psi h$. Then (3.6) transforms into $L'(\psi') = g$, with

$$L'_j = \sum_{i=j+1}^{r} \binom{i}{j+1} L_i h^{(i-j-1)}.$$

The order of L' is $r-1$ and $L'_{r-1}(0) = L_r(0) h_0$ is invertible in C. Hence, we can solve the equation $L'\xi = g$ by a recursive application of **linear**. Integration $\psi = \int \xi$ yields ψ.

ALGORITHM **linear**(L, g, n)

INPUT: A linear differential operator L of order r with coefficients in $\mathsf{TPS}(\mathsf{C}, n)$ and such that $L_r(0)$ is invertible in C, a truncated power series $g : \mathsf{TPS}(\mathsf{C}, n)$, and an order $n > r$.

OUTPUT: The first n terms of the unique solution to $L\varphi = g$, with $\varphi_0 = \cdots = \varphi_r = 0$.

L1. [Homogeneous equation]
Compute $R_0, \ldots, R_r$ using (3.7)
$R := L_0 R_0 + \cdots + L_r R_r$
$\hat{h} := \mathbf{ode}(R, 0, n-1)$
$h := \mathbf{exp}(\int \hat{h})$

L2. [Variation of constants]

$$L' := \sum_{j=0}^{r-1} \left[\sum_{i=j+1}^{r} \binom{i}{j+1} (L_i)_{0\cdots n-1} \times (h^{(i-j-1)})_{0\cdots n-1} \right] \frac{\partial^j}{\partial z^j}$$

$\xi := \mathbf{linear}(L', g, n-1)$
return $h \times \int \xi$

As to the time complexities of **ode** and **linear**, we observe that **ode** calls **linear** with the same r and **linear** calls **ode** and **linear** with r decreased by one. Hence, the time complexity is exponential in r. The following time complexity in n is easily proved by induction over r, using that $M(n)+M(\lceil(n+3r)/2\rceil)+M(\lceil(\lceil(n+3r)/2\rceil+3r)/2\rceil)+\cdots = O(M(n))$:

THEOREM 3.1. *Let $\Phi \in \mathsf{C}[y_0, \ldots, y_r, z]$ be a multivariate polynomial and consider the differential equation (3.3) with initial conditions $f(0), \ldots, f^{(r)}(0)$ that satisfy (3.4). Then this equation admits a unique solution $f \in \mathsf{C}[[z]]$ and there exists an algorithm which computes the first n coefficients of f in time $O(M(n))$.* □

REMARK. The algorithm **ode** generalizes to the case when Φ is a multivariate power series instead of a polynomial. In this case, we need assume that the algorithm **subst** also applies to $\Psi = \Phi$ and all its partial derivatives.

3.3. Algebraic and holonomic power series

An algebraic function is a function $f(z)$, which satisfies a polynomial relation of the form

$$P_d(z)f(z)^d + \cdots + P_0(z) = 0,$$

where $P_0, \ldots, P_d \in \mathsf{C}[z]$ are polynomials with $P_d \neq 0$. Such functions are special cases of holonomic functions, which are functions $f(z)$, that satisfy a linear differential equation

$$L_r(z)f^{(r)}(z) + \cdots + L_0(z)f(z) = 0, \tag{3.8}$$

where $L_0, \ldots, L_r \in \mathsf{C}[z]$ are polynomials with $L_r \neq 0$. An algebraic resp. holonomic power series is an algebraic resp. holonomic function which is also a power series in $\mathsf{C}[[z]]$.

Holonomic power series are interesting, because their coefficients can be computed sequentially in linear time and space. Indeed, the coefficients $f_0, f_1, \ldots$ of such power series satisfy a linear polynomial recurrence relation

$$Q_q(n)f_{n+q} + \cdots + Q_0(n)f_n = 0, \tag{3.9}$$

where $Q_0, \ldots, Q_q$ are polynomials in $\mathsf{C}[n]$. Here (3.9) is derived from (3.8) by extracting the coefficient of z^n from (3.8), while using the rules $(zf)_n = f_{n-1}$ and $(f')_n = nf_{n+1}$.

Furthermore, the class of holonomic functions enjoys many closure properties: it is (algorithmically) stable under addition, multiplication, right composition with algebraic functions, differentiation and integration, Hadamard product, etc. We refer to (Stanley, 1980; Lipshitz, 1989; Zeilberger, 1990) for more information on this subject. Holonomic functions are also available in some computer algebra systems (Salvy and Zimmermann, 1994).

3.4. Composition

3.4.1. Right composition with polynomials

Let $f = f_0 + \cdots + f_{p-1}z^{p-1}$ and $g = g_1 z + \cdots g_{q-1}z^{q-1}$ be polynomials, considered as truncated power series in $\mathsf{TPS}(\mathsf{C})$. In order to efficiently compute $f_{0\cdots n} \circ g_{0\cdots n} : \mathsf{TPS}(\mathsf{C}, n)$

for given n, we may use a divide and conquer method based on the formula

$$f \circ g = f_* \circ g + (f^* \circ g) g^{\lfloor p/2 \rfloor},$$

in which $f_* = f_{0 \cdots \lfloor p/2 \rfloor}$ and $f^* = f_{\lfloor p/2 \rfloor \cdots p}$ denote the lower and upper parts of f.

Although all computations will be done with truncated power series in our implementation, we will really compute with polynomials as long as their degrees remain inferior to n. Assuming that $g^i : \mathsf{TPS}(\mathsf{C}, \min((q-1)i+1, n))$ has been precomputed and stored in a hashtable H for all i of the form $\lfloor p/2^k \rfloor$ or $\lceil p/2^k \rceil$ with $k > 0$, we obtain the following algorithm:

Algorithm **compose_pol**(f, H, n)

Input: $f : \mathsf{TPS}(\mathsf{C}, p)$, a hashtable H and an integer n;
$H[i]$ contains $g^i_{0 \cdots \min((q-1)i+1, n)}$ for all $i \in \lfloor p/2^{\mathbb{N}^*} \rfloor \cup \lceil p/2^{\mathbb{N}^*} \rceil$.

Output: $f_{0 \cdots l} \circ g_{0 \cdots l}$, where $l = \min((p-1)(q-1)+1, n)$.

P1. [Start]
if $p = 0$ **then return** $f_{0 \cdots 1}$

P2. [Divide and conquer]
$l := \min((p-1)(q-1)+1, n)$
$h_* :=$ **compose_pol**$(f_{0 \cdots \lfloor p/2 \rfloor}, H, n)$
$h^* :=$ **compose_pol**$(f_{\lfloor p/2 \rfloor \cdots p}, H, n)$
return $(h_*)_{0 \cdots l} + (h^* \star H[\lfloor p/2 \rfloor])_{0 \cdots l}$

Theorem 3.2. *Let $f : \mathsf{TPS}(\mathsf{C}, p)$ and $g : \mathsf{TPS}(\mathsf{C}, q)$ be such that $g_0 = 0$ and let $n \geqslant 0$. There exists an algorithm to compute $f_{0 \cdots n} \circ g_{0 \cdots n}$ in time $O(\frac{pq}{n} M(n) \log n)$ and space $O(n \log \frac{pq}{n})$.*

Proof. Since the time and space complexities of the algorithm are increasing functions in p, q and n, we may assume without loss of generality that p, q and n are powers of two, We may also assume that $pq \geqslant n$.

The precomputation of the powers of g takes a time $O(\log n M(n))$. Denoting by $T(n, p, q)$ the time complexity apart from the precomputation, we have $T(n, 1, q) = O(1)$ and for $p > 1$:

$$T(n, p, q) \leqslant 2T(n, \tfrac{p}{2}, q) + O(M(\min(pq, n))). \tag{3.10}$$

This leads to the time complexity bound:

$$\begin{aligned} T(n, p, q) &\leqslant O(M(n) + 2M(n) + \cdots + \tfrac{pq}{n} M(n) + \\ &\quad\ \tfrac{pq}{2n} M(\tfrac{n}{2}) + \cdots + \tfrac{p}{2} M(2q)) + O(p) \\ &\leqslant O(\tfrac{pq}{n} M(n)) + O(\tfrac{pq}{n} M(n) \log n). \end{aligned}$$

We need a space $O(\min(q, n) + \min(2q, n) + \cdots + \min(pq, n)) \leqslant O(n \log q)$ in order to store the powers of g. For the remaining space $S(n, p, q)$ needed by the algorithm, we have $S(n, 1, q) = O(1)$ and for $p > 1$:

$$S(n, p, q) \leqslant S(n, \tfrac{p}{2}, q) + O(\min(\tfrac{pq}{2}, n)). \tag{3.11}$$

This yields the space complexity bound:

$$\begin{aligned} S(n,p,q) &\leqslant O(n\log\tfrac{pq}{n}+\tfrac{n}{2}+\cdots+q)+O(1)\\ &\leqslant O(n\log\tfrac{pq}{n}). \end{aligned}$$

□

3.4.2. Right composition with algebraic power series

The algorithm **compose_pol** generalizes to the case when g is an algebraic power series with $g_0=0$, i.e.

$$P_d g^d+\cdots+P_0=0, \tag{3.12}$$

with $P_0,\cdots,P_d\in \mathsf{C}[z]$ and $P_d\neq 0$. We will denote by v the valuation of P_d and by q the maximum of the degrees of the P_i plus one.

Algebraic functions

Instead of computing with (truncated) polynomials, we will now compute with (truncated) algebraic functions in $\mathsf{C}[z,g]$, which are conveniently represented by fractions

$$F=\frac{F_{d-1}g^{d-1}+\cdots+F_0}{P_d^{k_F}}, \tag{3.13}$$

where $F_0,\cdots,F_{d-1}\in\mathsf{C}[z]$ and $k_F\in\mathbb{N}$. The *degree* $\deg F$ of F is defined to be $\deg F=\max_{0\leqslant i<d}\deg F_i+i(q-1)$. We also define the *multiplicity* μ_F of the pole P_d in F as $\mu_F=\max\{i|F_i\neq 0\}$ if $k_f=0$ and $\mu_F=k_f+d-1$ otherwise.

The addition of two fractions like (3.13) is done as usual: we multiply one of the numerators with a suitable power of P_d in order to obtain a common denominator ($k_{F+G}=\max(k_F,k_G)$) and we add up the numerators. Notice that we have

$$\left\{\begin{array}{rcl} \deg F+G &\leqslant& \max(\deg F,\deg G);\\ \mu_{F+G} &\leqslant& \max(\mu_F,\mu_G). \end{array}\right. \tag{3.14}$$

The asymptotic cost of the addition $F+G$ is

$$T_{F+G}=O(dM(\deg F+\deg G+\deg P_d^{k_f}+\deg P_d^{k_g})).$$

In order to multiply fractions like (3.13), we first precompute $g^d,\ldots,g^{2d-2}$ as fractions (3.13), using (3.12):

$$g^i=\frac{g^i_{d-1}g^{d-1}+\cdots+g^i_0}{P_d^{i-(d-1)}}. \tag{3.15}$$

Notice that $\deg g^i\leqslant i(q-1)$ for all i. Now in order to compute the product

$$F\times G=\frac{F_{d-1}g^{d-1}+\cdots+F_0}{P_d^{k_F}}\times\frac{G_{d-1}g^{d-1}+\cdots+G_0}{P_d^{k_G}},$$

we first rewrite the product as

$$F\times G=\frac{1}{P_d^{k_F+k_G}}\sum_{i=0}^{2d-2}\left(\sum_j F_jG_{i-j}\right)g^i.$$

Next, we substitute g^i by the right hand side of (3.15) for $d \leqslant i \leqslant 2d-2$. Notice that

$$\begin{cases} \deg F \times G & \leqslant & \deg F + \deg G; \\ \mu_{F \times G} & \leqslant & \mu_F + \mu_G. \end{cases} \tag{3.16}$$

The asymptotic cost of the multiplication $F \times G$ is

$$T_{F \times G} = O(dM(\deg F + \deg G) + qd^2(\deg F + \deg G)),$$

since the polynomials g_j^i are fixed.

Let $f = f_0 + \cdots + f_{p-1} z^{p-1}$ be a polynomial. Then the bounds (3.14) and (3.16) yield the following bounds for its right composition $f \circ g = f_0 + \cdots + f_{p-1} g^{p-1}$ with g:

$$\begin{cases} \deg f \circ g & \leqslant & (p-1)(q-1)+1; \\ \mu_{f \circ g} & \leqslant & p-1. \end{cases} \tag{3.17}$$

In particular, $k_{f \circ g} \leqslant \max(p-d, 0)$.

The algorithm

In the truncated context, the polynomials F_i in (3.13) are replaced by truncated power series in $\mathsf{TPS}(\mathsf{C}, n + k_F v)$. This will enable us to extract the first n coefficients of F, when considered as a power series. We will denote by $\mathsf{Algebraic\_TPS}(\mathsf{C}, g)$ and $\mathsf{Algebraic\_TPS}(\mathsf{C}, g, n)$ the algebraic analogues of the classes $\mathsf{TPS}(\mathsf{C})$ resp. $\mathsf{TPS}(\mathsf{C}, n)$.

Now assume that we want to compute the composition of a polynomial $f = f_0 + \cdots + f_{p-1} z^{p-1}$ with the series g up till order n. Then we do the following

- We precompute $g^i : \mathsf{Algebraic\_TPS}(\mathsf{C}, g, \min(i(q-1)+1-\max(i+1-d, 0)v, n))$ for all i of the form $\lfloor p/2^k \rfloor$ or $\lceil p/2^k \rceil$ with $k > 0$. Recall that $\mu_{g^i} \leqslant i$ for all i.
- We precompute $P_d^i : \mathsf{Algebraic\_TPS}(\mathsf{C}, g, \min(i(q-1)+1-\max(i+1-d, 0)v, n))$ for all i of the form $\lfloor p/2^k \rfloor$ or $\lceil p/2^k \rceil$ with $k > 0$.
- We apply the analogue **compose_alg** of **compose_pol** below.
- We convert the result in $\mathsf{Algebraic\_TPS}(\mathsf{C}, g, n)$ back to a truncated series in $\mathsf{TPS}(\mathsf{C}, n)$. This can be done in time $O(M(N))$ using fast division and the linear recurrence relation for the coefficients of g (see section 3.3).

Algorithm **compose_alg**(f, H, P, n)

Input: $f : \mathsf{TPS}(\mathsf{C}, p)$, hashtables H, P and an integer n;
$H[i]$ contains $g^i_{0 \cdots \min(i(q-1)+1-\max(i+1-d,0)v, n)}$ for all $i \in \lfloor p/2^{\mathbb{N}^*} \rfloor \cup \lceil p/2^{\mathbb{N}^*} \rceil$.
$P[i]$ contains $(P_d^i)_{0 \cdots \min(i(q-1)+1-\max(i+1-d,0)v, n)}$ for all $i \in \lfloor p/2^{\mathbb{N}^*} \rfloor \cup \lceil p/2^{\mathbb{N}^*} \rceil$.

Output: $h = f_{0 \cdots n} \circ g \in \mathsf{Algebraic\_TPS}(\mathsf{C}, g, l)$,
with $l = \min((p-1)(q-1)+1-\max(p-d, 0)v, n)$.

P1. [Start]
if $p = 0$ **then return** $f_{0 \cdots 1}$

P2. [Divide and conquer]
$l := \min((p-1)(q-1)+1-\max(p-d, 0)v, n)$
$h_* :=$ **compose_alg**$(f_{0 \cdots \lfloor p/2 \rfloor}, H, n)$
$h^* :=$ **compose_alg**$(f_{\lfloor p/2 \rfloor \cdots p}, H, n)$
return $(h_*)_{0 \cdots l} + (h^* \star H[\lfloor p/2 \rfloor])_{0 \cdots l}$

REMARK. The hashtable P is used in the final addition, in order to rewrite the left hand and right hand fractions, such that they have a common denominator.

THEOREM 3.3. *Let* $f : \mathsf{TPS}(\mathsf{C}, p)$*,* g *as above and* $n \geqslant 0$*. Then there exists an algorithm to compute the first* n *coefficients of* $f \circ g$ *in time* $O(qd^2 \frac{p(q-v)}{n}) M(n+pv) \log n)$ *and space* $O(d(pv + n \log \frac{p(q-v)}{n}))$.

PROOF. The proof is analogous to the proof of theorem 3.2. In this case, using that

$$\begin{aligned} k_{h_*}, k_{h^*}, k_{H[p/2]} &= O(p); \\ \deg h_*, \deg h^*, \deg H[p/2] &= O(\min(pq, pv)), \end{aligned}$$

the main inequalities (3.10) and (3.11) become

$$\begin{aligned} T(n,p,q) &\leqslant 2T(n, \tfrac{p}{2}, q) + O(qd^2 M(\min(pq, n+pv))); \\ S(n,p,q) &\leqslant S(n, \tfrac{p}{2}, q) + O(d \min(pq, n+pv)). \end{aligned}$$

□

REMARK. Notice that $v = 0$, if g is a rational function, and the time and space complexity bounds become $O(d^2 q^2 M(n) \log n)$ resp. $O(dn \log q)$ for $p = n$. The composition algorithm may also be simplified in this particular, but important case.

3.4.3. GENERAL COMPOSITION FOR DIVISIBLE RINGS C

If C is a divisible ring, then Brent and Kung's fast algorithm (Brent and Kung, 1978) can be used in order to compute the composition $f \circ g$ of formal power series f and g up to order n. Their method relies on decomposing $g = g_* + g^* = g_{0 \cdots q} + g_{q \cdots n}$ with $q = \lfloor \sqrt{n / \log n} \rfloor$ and using the Taylor series expansion at order $r = \lceil n/q \rceil$

$$f \circ g = f \circ g_* + (f' \circ g_*) g^* + \cdots + \tfrac{1}{(r-1)!} (f^{(r-1)} \circ g_*)(g^*)^{r-1} + O(z^n). \tag{3.18}$$

Assuming that $(g'_*)_0$ is invertible in C, $f^{(i)} \circ g_*$ can then easily be computed as a function of $f^{(i-1)} \circ g_*$, since

$$f^{(i)} \circ g_* = (f^{(i-1)} \circ g_*)' / g'_*.$$

Inversely, if $(g'_*)_0$ is not necessarily invertible in C, we may write

$$\tfrac{1}{(i-1)!} f^{(i-1)} \circ g_* = f_{i-1} + i \left(\int (\tfrac{1}{i!} f^{(i)} \circ g_*) g'_* \right).$$

This leads to the following algorithm:

ALGORITHM **compose**(f, g)

INPUT: $f, g \in \mathsf{TPS}(\mathsf{C}, n)$ with $g_0 = 0$.
OUTPUT: $f \circ g$.

C1. [Polynomial Composition]
$q := \lfloor \sqrt{n / \log n} \rfloor$
$r := \lceil n/q \rceil$

$g_* := (g_{0\cdots q})_{0\cdots n}$
$g^* := g - g_*$
Compute $H[i] := (g_*^i)_{0\cdots \min((q-1)i+1,n)}$ for all $i = \lfloor n/2^k \rfloor$ and $i = \lceil n/2^k \rceil$ with $k > 0$
$D :=$ **compose_pol**$(f^{(r-1)}, H, n+1-r)/(r-1)!$

C2. [Taylor expansion]
$S := D_{0\cdots \max(0,n+q-rq)}$
for $i := r-1$ **downto** 1 **do**

$D := (f_{i-1})_{0\cdots n+1-i} + \int((iD) \times (g_*')_{0\cdots n-i})$
$T := (S \times (g^* \gg q)_{0\cdots \max(0,n-iq)}) \ll q$
$S := D_{0\cdots \max(0,n+q-iq)} + T_{0\cdots \max(0,n+q-iq)}$

return S

THEOREM 3.4. *Let f and g be power series truncated at order n. Assuming that $g_0 = 0$, there exists an algorithm to compute the power series expansion of $f \circ g$ up till order n in time $O(\sqrt{n \log n} M(n))$ and space $O(n \log n)$.*

PROOF. Step 1 takes a time $O(\sqrt{n \log n} M(n))$ and space $O(n \log n)$, by theorem 3.2. Since the loop in the second step requires only $r = O(\sqrt{n \log n})$ iterations, the second step requires a time $O(\sqrt{n \log n} M(n))$ and space $O(n)$. □

REMARK. The above algorithm also applies if C is an overring of $\mathbb{Z}$, such that the equation $nx = y$ can be solved effectively in C for $n \in \mathbb{Z}^*$ and $y \in \mathsf{C}$ (i.e. we can test whether the equation admits a solution and, if so, compute it). Indeed, in this case, we can do the computations in the effective partial quotient ring of C in which the non zero integers are invertible.

3.4.4. General composition for rings C of finite characteristic

Assume now that $\mathbb{Z}$ can no longer be embedded in C, i.e. the canonical ring homomorphism $\mathbb{Z} \in \mathsf{C}$ has a non trivial kernel $r\mathbb{Z}$ with $r > 0$. Bernstein recently gave a fast composition algorithm for such C (Bernstein, 1998). The idea is to consider subsequently the cases when r is prime, a prime power and general.

$r = p$ is prime

We have $(a+b)^p = a^p + b^p$ for all $a, b \in \mathsf{C}$ and $g(z)^p = g_0^p + g_1^p z^p + g_2^p z^{2p} + \cdots = g^{[p]}(z^p)$ for power series $g(z) = g_0 + g_1 z + g_2 z^2 + \cdots$. Hence we may use the following formula to compute the composition of two power series f and g:

$$f \circ g = \sum_{i=0}^{p-1} (f_i + f_{i+p} z + f_{i+2p} z^2 + \cdots) \circ g^{[p]}(z^p) g^i. \tag{3.19}$$

Assuming that we have an algorithm **Horner**(P, h) to compute $P(h)$ by Horner's method for $P \in \mathsf{TPS}(\mathsf{C}, n)[X]$ and $h \in \mathsf{TPS}(\mathsf{C}, n)$, this leads to the following recursive algorithm of time complexity $O((p/\log p) M(n) \log n)$ and linear space complexity (Bernstein, 1998):

ALGORITHM **prime_compose**(f, g)

INPUT: $f, g : \mathsf{TPS}(\mathsf{C}, n)$ with $g_0 = 0$.
We assume that C has prime characteristic p.

OUTPUT: $f \circ g$.

$m := \lceil n/p \rceil$
for $i := 0$ **to** $p - 1$ **do**
  $L := f_i + \cdots + f_{i+p(m-1)} z^{m-1}$
  $R := g_0^p + \cdots + g_{m-1}^p z^{m-1}$
  $h_i := (\mathbf{prime\_compose}(L, R) \circ z^p)_{0 \cdots n}$

return Horner$(h_0 + \cdots + h_{p-1} X^{p-1}, g)$

REMARK. The algorithm can be optimized by using the algorithm **compose** from the previous section for small n. Indeed, it suffices that $1, 2, \ldots, \lceil n/\lfloor \sqrt{n/\log n} \rfloor \rceil$ are invertible in C.

$r = p^k$ IS A PRIME POWER

In this case, the composition algorithm is based on the fact that we still have $\delta = g(z)^p - g^{[p]}(z^p) \in p\mathsf{C}$. Hence, (3.19) becomes

$$f \circ g = \sum_{i=0}^{p-1} (f_i + f_{i+p} z + f_{i+2p} z^2 + \cdots) \circ (g^{[p]}(z^p) + \delta) g^i. \tag{3.20}$$

This leads to the more general problem of composing f with $g + \varepsilon$, where ε is an infinitesimal formal parameter with $\varepsilon^k = p\varepsilon^{k-1} = \cdots = p^{k-1}\varepsilon = 0$. The analogue relation of (3.20) then again yields a recursive formula and we obtain the following algorithm of time complexity $O((k^3 p/\log p) M(n) \log n)$ and space complexity $O(kn)$ (Bernstein, 1998):

ALGORITHM **prime_power_compose**(f, g)

INPUT: $f, g : \mathsf{TPS}(\mathsf{C}, n)$ with $g_0 = 0$.
We assume that C has prime power characteristic p^k.

OUTPUT: $f \circ (g + \varepsilon) : \mathsf{TPS}(\mathsf{C}[\varepsilon]/(\varepsilon^k, p\varepsilon^{k-1}, \ldots, p^{k-1}), n)$.

$m := \lceil n/p \rceil$
$\varphi := (g + \varepsilon)^p - (g_0^p + \cdots + g_m^p z^{mp})_{0 \cdots n}$
for $i := 0$ **to** $p - 1$ **do**
  $L := f_i + \cdots + f_{i+p(m-1)} z^{m-1}$
  $R := g_0^p + \cdots + g_{m-1}^p z^{m-1}$
  $\psi := (\mathbf{prime\_power\_compose}(L, R) \circ z_p)_{0 \cdots n}$
  Write $\psi = \psi_0 + \psi_1 \varepsilon + \cdots + \psi_{k-1} \varepsilon^{k-1}$
  $h_i := \mathbf{Horner}(\psi_0 + \cdots + \psi_{k-1} X^{k-1}, \varphi, n)$

return Horner$(h_0 + \cdots + h_{p-1} X^{p-1}, g, n)$

$r = q_1 \cdots q_l$ IS A NON TRIVIAL PRODUCT OF PRIME POWERS

This case is a standard application of the Chinese remainder theorem. More precisely, using the Chinese remainder theorem, we first compute integers $i_1, \ldots, i_l$ with

$$i_1 \frac{q_1 \cdots q_l}{q_1} + \cdots + i_l \frac{q_1 \cdots q_l}{q_l} = 1 \mod q_1 \cdots q_l.$$

We next compute the compositions of the projections of f and g in $\mathsf{C}/(q_j)[[z]]$. More precisely, for each j, elements in $\mathsf{C}/(q_j)$ are *redundantly* represented by elements in C (we do not require a zero test) and we use the previous algorithm. We thus obtain a truncated series $h_j \in \mathsf{C}[[z]]$ with $h_j - g \circ f \in q_j \mathsf{C} + O(z^n)$. Then we have $i_1 h_1 + \cdots + i_l h_l$ is equal to $f \circ g$ up till n terms.

THEOREM 3.5. *Let* C *be a ring of positive characteristic* $r > 0$ *and let* $f, g : \mathsf{TPS}(\mathsf{C}, n)$ *be such that* $g_0 = 0$*. Then* n *terms of* $f \circ g$ *can be computed in time* $O((r/\log r) M(n) \log n)$ *and space* $O(n \log r)$*.*

PROOF. By what precedes and since $k^3 p / \log p = O(p^k / (k \log p))$, the theorem holds for prime power characteristic. In general, we have

$$\frac{q_1}{\log q_1} + \cdots + \frac{q_l}{\log q_l} = O\left(\frac{r}{\log r}\right)$$

and $\log q_1 + \cdots + \log q_r = \log r$, so we can perform the composition modulo each q_i in the required time and space. Gluing these partial results together using the Chinese remainder theorem takes linear time and space. □

4. Relaxed multiplication

4.1. NAIVE RELAXED MULTIPLICATION

The lazy, or naive relaxed multiplication algorithm for formal power series f and g in z just computes the coefficient of z^n in fg using the convolution sum $(fg)_n = \sum_{i=0}^n f_i g_{n-i}$. In order to implement this method, we define the class

CLASS $\mathsf{Product1\_Series\_Rep}(\mathsf{C}) \rhd \mathsf{Series\_Rep}(\mathsf{C})$

$\quad f, g \ : \mathsf{Series}(\mathsf{C})$

The constructor takes two series on input which are stored in f and g. We compute the n-th coefficient of fg as follows:

METHOD $\mathsf{Product1\_Series\_Rep}(\mathsf{C}).\mathbf{next}()$
ACTION: The next coefficient $(fg)_n$.

return $\sum_{i=0}^n f_i g_{n-i}$

The actual function for multiplication is given by

ALGORITHM $(f : \mathsf{Series}(\mathsf{C})) \times (g : \mathsf{Series}(\mathsf{C}))$
INPUT: Two series f and g.
OUTPUT: Their product fg.

return new $\mathsf{Product1\_Series\_Rep}(\mathsf{C})(f, g)$

$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
g_7	7	8	9	10	11	12	13	14	$\cdots$
g_6	6	7	8	9	10	11	12	13	$\cdots$
g_5	5	6	7	8	9	10	11	12	$\cdots$
g_4	4	5	6	7	8	9	10	11	$\cdots$
g_3	3	4	5	6	7	8	9	10	$\cdots$
g_2	2	3	4	5	6	7	8	9	$\cdots$
g_1	1	2	3	4	5	6	7	8	$\cdots$
g_0	0	1	2	3	4	5	6	7	$\cdots$
$\times$	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	$\cdots$

Figure 1. Relaxed multiplication by the naive algorithm

Obviously, the naive multiplication algorithm has $O(n^2)$ resp. $O(n)$ time and space complexities. The computation of the successive coefficients of fg by the naive algorithm is illustrated in figure 1: each box corresponds to the contribution of a product $f_i g_j$ to the sum $(fg)_{i+j} = \sum_{k=0}^{i+j} f_k g_{i+j-k}$. The number of the box corresponds to the stage when this contribution is computed. Indeed, the naive algorithm only computes $f_i g_j$ at the moment that $(fg)_{i+j}$ is needed, that is, at stage $i + j$.

4.2. Relaxation of DAC-multiplication

The relaxed multiplication algorithm we present in this section is based on the observation that DAC-multiplication is *essentially relaxed*. Hereby we mean that, if we apply the algorithm to compute the product of two power series f and g with symbolic coefficients, then the computed formula for $(fg)_i$ only depends on the coefficients $f_0, \ldots, f_i$ and $g_0, \ldots, g_i$. In order to transform this observation into an actual relaxed multiplication algorithm, the main problem is to design suitable data structures, which correspond to partial executions of the divide and conquer algorithm. Roughly speaking, the whole computation will be stored in memory, but information which is no longer needed at a given stage is freed again.

4.2.1. Relaxed multiplication of polynomials

Let $f = f_0 + f_1 z + \cdots + f_{N-1} z^{N-1}$ and $g = g_0 + g_1 z + \cdots + g_{N-1} z^{N-1}$ be two polynomials of degrees $< N$, represented as truncated series at order $O(z^N)$. In this section, we show how to compute the coefficients of their product fg in a relaxed way. For the application we have in mind, we will suppose that N is a power of two. The representation class which corresponds to the relaxed computation of fg is given by

Class $\mathsf{DAC\_Rep(C)}$ $\triangleright$ $\mathsf{Series\_Rep(C)}$

N : $\mathsf{Integer}$
f, g : $\mathsf{Series(C)}$
lo, mid, hi : $\mathsf{DAC(C)}$

The pointers lo, mid and hi correspond to the relaxed computations of $f_* g_*$, $(f_* + f^*)(g_* +$

g^*) and $f^* g^*$ (with $f_* = f_{0\cdots N/2}, f^* = f_{N/2\cdots n}, g_* = g_{0\cdots N/2}$ and $g^* = g_{N/2\cdots N}$). The constructor for $\mathsf{DAC\_Rep(C)}$ is given by

CONSTRUCTOR $\mathsf{DAC\_Rep(C)}(\overline{f}, \overline{g}, \overline{N})$

INPUT: Two series $\overline{f}, \overline{g}$ and an order $\mathsf{TPS}(\mathsf{C}, \overline{N})$.

$N := \overline{N}$, $f := \overline{f}$, $g := \overline{g}$
$lo := mid := hi :=$ **null**
$\varphi := 0_{0\cdots 2N-1}$

The computation of the coefficients now goes in three stages. At the first stage, when $0 \leqslant n < \frac{N}{2}$, we only compute the product $f_* g_*$; the pointer lo becomes non null at this stage. At the second stage, when $\frac{n}{2} \leqslant n < N$, we also start the computations of $(f_* + f^*)(g_* + g^*)$ and $f^* g^*$; the pointers mid and hi also become non null at this stage. At the third, and last stage, when $n \geqslant N$, the computation of fg is completed and the pointers lo, mid and hi are freed. For small $N \leqslant \mathit{Threshold}_{\mathsf{C}}$, where $\mathit{Threshold}_{\mathsf{C}}$ is a power of two, we compute fg using the lazy multiplication algorithm.

METHOD $\mathsf{DAC\_Rep(C)}.\mathbf{next}()$

OUTPUT: The next coefficient $(f_{0\cdots N} \star g_{0\cdots N})_n$.

D0. [Small N]
if $N > \mathit{Threshold}_{\mathsf{C}}$ **then go to D1**
if $n < N$ **then return** $\sum_{i=0}^{n} f_i g_{n-i}$
else return $\sum_{i=n-(N-1)}^{N-1} f_i g_{n-i}$

D1. [First stage]
if $n \geqslant \frac{N}{2}$ **then go to D2**
if $n = 0$ **then** $lo :=$ **new** $\mathsf{DAC\_Rep}(f, g, \frac{N}{2})$
return lo_n

D2. [Second stage]
if $n \geqslant N$ **then go to D3**
if $n = \frac{N}{2}$ **then**
  $mid :=$ **new** $\mathsf{DAC\_Rep}(f + (f \gg \frac{N}{2}), g + (g \gg \frac{N}{2}), \frac{N}{2})$
  $hi :=$ **new** $\mathsf{DAC\_Rep}(f \gg \frac{N}{2}, g \gg \frac{N}{2}, \frac{N}{2})$
return $lo_n + mid_{n-N/2} - lo_{n-N/2} - hi_{n-N/2}$

D3. [Third stage]
if $n \geqslant 2N - 1$ **then return** 0
if $n > N$ **then return** φ_n
$\varphi_{N\cdots 2N-1} := hi_{0\cdots N-1}$
$\varphi_{N\cdots \frac{3N}{2}-1} \mathrel{+}= mid_{\frac{N}{2}\cdots N-1} - lo_{\frac{N}{2}\cdots N-1} - hi_{\frac{N}{2}\cdots N-1}$
$lo := mid := hi :=$ **null**
return φ_n

$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
g_7	7	7	7	7	7	7	7	7	$\cdots$
g_6	6	7	6	7	6	7	6	7	$\cdots$
g_5	5	5	7	7	5	5	7	7	$\cdots$
g_4	4	5	6	7	4	5	6	7	$\cdots$
g_3	3	3	3	3	7	7	7	7	$\cdots$
g_2	2	3	2	3	6	7	6	7	$\cdots$
g_1	1	1	3	3	5	5	7	7	$\cdots$
g_0	0	1	2	3	4	5	6	7	$\cdots$
$\times$	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	$\cdots$

Figure 2. Relaxed multiplication by the divide and conquer algorithm

4.2.2. Complexity analysis

Up to some extra operations related to the storage of partial auxiliary products, the main control structure of the relaxed DAC-multiplication algorithm is the same as in the classical algorithm. Hence, their respective time complexities only differ up to a constant factor.

As to the memory storage $S(N)$ needed by the relaxed algorithm, we claim that

$$S(N) \leqslant 2S(N/2) + O(N). \tag{4.1}$$

Indeed, as long as less than $N/2$ coefficients of f and g are known, f^* and g^* are not needed at all. As soon as $N/2$ coefficients are known, f_* and g_* are entirely determined, whence the computation of $f_* g_*$ is completed, and the result takes $O(N)$ memory storage. Furthermore, $f_* + f^*$ and $g_* + g^*$ require another $O(N)$ memory storage, while the computations of $(f_* + f^*)(g_* + g^*)$ and $f^* g^*$ require $2S(N/2)$ memory storage, by induction. From (4.1), we deduce that

$$S(N) = O(N \log N).$$

4.2.3. General relaxed DAC-multiplication

Let us finally treat the case, when we want to compute fg up to any order, and not merely up to order $O(z^N)$. In this case, we use the algorithm from above between successive powers of two. Each time we cross a power of two, we let the old f and g play the rôles of f_* and g_* for the new f and g. More precisely, we introduce the class

Class $\mathsf{Product2\_Series\_Rep(C)} \triangleright \mathsf{Series\_Rep(C)}$

  f, g : $\mathsf{Series(C)}$
  h : $\mathsf{DAC(C)}$

The constructor takes two series on input, which are stored in f and g; h is initialized with **new** $\mathsf{DAC\_Rep(C)}(f, g, \mathit{Threshold}_{\mathsf{C}})$. The member function **next** is now given by

METHOD $\mathsf{Product2\_Series\_Rep(C)}.\mathbf{next}(n)$
OUTPUT: The next coefficient $(fg)_n$.

if $n \geqslant \mathit{Threshold}_{\mathsf{C}}$ **and** $n \in 2^{\mathbb{N}}$ **then** $h :=$ **new** $\mathsf{DAC\_Rep(C)}(f, g, h, 2n)$
return h_n.

Here we use a second constructor for $\mathsf{DAC\_Rep(C)}$ in order to extend previous computations:
CONSTRUCTOR $\mathsf{DAC\_Rep(C)}(\overline{f}, \overline{g}, \overline{g}, \overline{N})$
INPUT: Series $\overline{f}, \overline{g}, \overline{h_*}$ and an order $\mathsf{TPS}(\mathsf{C}, \overline{N})$.

$N := \overline{N}$, $f := \overline{f}$, $g := \overline{g}$
$lo := h_*, mid := hi :=$ **null**
$\varphi := (h_*)_{0\cdots 2N-1}, n := N/2$

Clearly, the time and space complexities of this algorithm are again $O(n^{\log 3/\log 2})$ and $O(n \log n)$. In figure 2, we schematically represented the computation process of the successive coefficients of fg by the relaxed multiplication algorithm.

4.3. FAST RELAXED MULTIPLICATION

Although relaxed DAC-multiplication is significantly faster than the naive algorithm, it still is not as fast as the fastest zealous multiplication algorithms based on the fast Fourier transform. In this section, we give a fast relaxed multiplication algorithm, in which the fast Fourier transform may be exploited.

For each $i, j, p \in \mathbb{N}$, let us denote

$$\begin{aligned} \Pi_{i,j,p} &= (f_{i2^p-1} z^{i2^p-1} + \cdots + f_{(i+1)2^p-1} z^{(i+1)2^p-1}) \times \\ &\quad (g_{j2^p-1} z^{j2^p-1} + \cdots + g_{(j+1)2^p-1} z^{(j+1)2^p-1}) \end{aligned}$$

The fast multiplication algorithm is based on the observation that, as soon as the first $2^{p+1}-1$ coefficients of f and g are known, then the contribution of $\Pi_{1,1,p}$ to fg can be computed prematurely by *any* fast zealous multiplication algorithm. More generally, as soon as the first $n = k2^p - 1$ coefficients of f and g are known, with odd $k \geqslant 3$ and $p \geqslant 1$, then we can compute the contributions of $\Pi_{1,k-1,p}$ and $\Pi_{k-1,1,p}$.

4.3.1. FAST RELAXED MULTIPLICATION ALGORITHM

The representation class $\mathsf{Product3\_Series\_Rep(C)}$ and its constructor are taken to be the same as for $\mathsf{Product1\_Series\_Rep(C)}$:

CLASS $\mathsf{Product3\_Series\_Rep(C)} \triangleright \mathsf{Series\_Rep(C)}$
  f, g : $\mathsf{Series(C)}$

The coefficients of fg are computed as follows:

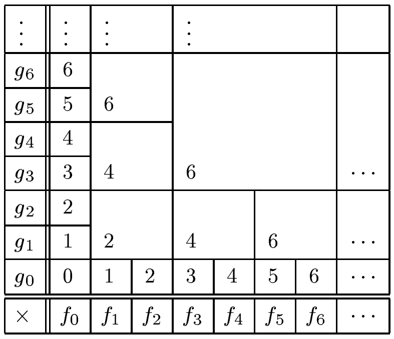

Figure 3. Fast relaxed multiplication

METHOD Product3_Series_Rep(C).**next**()
OUTPUT: The next coefficient $(fg)_n$.

F1. [Enlarge φ]
Let $k \in 2^{\mathbb{N}}$ be minimal with $k \geqslant 2n$.
if $\sharp\varphi < k$ **then** $\varphi := \varphi_{0\cdots k}$

F2. [Accumulate]
$k := 2(n+2), p := -1$
while $(k \textbf{ mod } 2) = 0$

$k := k/2, p := p+1$
$\varphi_{k2^p-2\cdots(k+2)2^p-3} \mathrel{+}= f_{2^p-1\cdots 2^{p+1}-1} \star g_{(k-1)2^p-1\cdots k2^p-1}$
if $k = 2$ **then return** φ_n
$\varphi_{k2^p-2\cdots(k+2)2^p-3} \mathrel{+}= f_{(k-1)2^p-1\cdots k2^p-1} \star g_{2^p-1\cdots 2^{p+1}-1}$

return φ_n

The computation process is schematically represented in figure 3. From this figure, it is easily seen that the contribution of each $f_i g_j$ to $(fg)_{i+j}$ is computed exactly once and before the coefficient $(fg)_{i+j}$ is output. This proves the correctness of our algorithm.

4.3.2. Complexity analysis

THEOREM 4.1. *There exists a relaxed multiplication algorithm for formal power series f and g with coefficients in* C*, which computes the first n terms of fg in time $O(M(n)\log n)$ and space $O(n)$.*

PROOF. Since the time complexity of the algorithm from the previous section is an increasing function in n, it suffices to consider the case when $n = 2^p - 1$ for some $p > 0$. Then looking at figure 3, we observe that the algorithm performs $2(n+1) - 3$ constant multiplications, $(n+1) - 3$ multiplications of polynomials with 2 terms, $\frac{1}{2}(n+1) - 3$ multiplications of polynomials with 4 terms and so on. Hence, the overall time complexity is bounded by

$$2\sum_{k=0}^{p-1}\frac{n}{2^k}M(2^k)+O(n)=O(M(n)\log n).$$

The space complexity is clearly bounded by $O(n)$. □

4.4. Remarks and optimizations

Although the relaxed multiplication algorithms from sections 4.2 and 4.3 are both asymptotically faster than lazy multiplication, they both have drawbacks for certain applications: the relaxed DAC-multiplication algorithm is more cumbersome to implement (whence a large overhead) and it has an additional logarithmic space overhead. On the other hand, fast relaxed multiplication has a good space complexity, is asymptotically fast and easy to implement, but the algorithm outperforms the relaxed divide and conquer algorithm only for large values of n, especially when multiplications in C are expensive, so that the extra overhead needed by the divide and conquer strategy can be neglected. Finally, if we know beforehand that we wish to compute only n coefficients of a power series, then both methods have the drawback that they anticipate the computation of the next n coefficients.

Consequently, it is interesting to search for algorithms which overcome these problems and we will make some suggestions in this section. In table 3 we have compared the respective complexities of different methods, by counting the number of constant multiplications they use as a function of n. For the fast relaxed algorithm and the variant from section 4.4.1 below, we both considered the cases in which we use

I. DAC-multiplication.
II. A linear algorithm with $M(n)=2n-1$

for zealous multiplication.

As a conclusion, it seems that there is no overall best relaxed multiplication method. The implementer should choose the algorithm as a function of the application he has in mind and in particular as a function of the cost of constant multiplications, the expansion order n, the space complexity he is willing to pay, the desired degree of laziness and the time he wishes to spend on his implementation. We refer to section 7 for a further discussion of this issue.

4.4.1. An alternative fast relaxed multiplication algorithm

It is possible to slightly improve constant factor in the theoretical complexity of the algorithm from section 4.3.1, by using the trick (3.1) in order to compute the contributions of $\Pi_{1,k-1,p}$ and $\Pi_{k-1,1,p}$ simultaneously. Unfortunately, this makes the algorithm more complex, since this supposes that we have $\Pi_{1,1,p}$ and $\Pi_{k-1,k-1,p}$ in memory. Nevertheless, working the idea out carefully leads to the slightly more efficient algorithm below, which uses approximately twice as much memory. In this algorithm, the "diagonal products" $\Pi_{i,i,p}$ are retrieved from the truncated series ψ.

In figure 4 we illustrated the corresponding computation process. In table 3 we compared its theoretical efficiency with the algorithm from section 4.3.1. However, it should be noticed that, in practice, for certain constant rings C, the operands for which we apply the trick (3.1) usually have very different sizes, so that the mean cost of multiplications in C may be higher for the alternative algorithm.

n	1	2	3	4	5	6	7	8	9	10	100	1000	10000
Naive	1	3	6	10	15	21	28	36	45	55	5050	500500	50005000
DAC	1	3	5	9	11	15	19	27	29	33	1251	52137	1844937
Fast-I	1	3	8	10	18	20	37	39	47	49	2938	103693	4458055
Fast-II	1	3	8	10	18	20	35	37	45	47	1602	27408	411963
Variant-I	1	3	5	8	14	16	22	24	33	35	1904	66515	2535836
Variant-II	1	3	5	8	14	16	22	24	33	35	1176	20311	300794

Table 3. The number of needed constant multiplications at order n for different relaxed multiplication algorithms.

CLASS Product4_Series_Rep(C) $\triangleright$ Series_Rep(C)

f, g : Series(C)
ψ : TPS(C)

METHOD Product4_Series_Rep(C).**next**()

OUTPUT: The next coefficient $(fg)_n$.

V1. [Enlarge φ and ψ]
Let $k \in 2^{\mathbb{N}}$ be minimal with $k \geqslant 2n$.
if $\sharp\varphi < k$ **then** $\varphi := \varphi_{0\cdots k}$
if $\sharp\psi < k$ **then** $\psi := \psi_{0\cdots k}$

V2. [Accumulate]
$c := f_n g_n$
φ_{2n} += c
ψ_{2n} += c
if $n + 2 = 5 \times 2^p \quad (p \in \mathbb{N})$ **then accumulate**$(2 \times 2^p - 1, 3 \times 2^p - 1, 2^p, \mathbf{true})$

$k := 2(n+2), p := -1$
while $(k \,\mathbf{mod}\, 2) = 0$ **and** $k \neq 4$
  $k := k/2, p := p + 1$
  if $p > 0$ **then accumulate**$((2k-1)2^{p-1} - 1, (2k-2)2^{p-1} - 1, 2^{p-1}, \mathbf{true})$
  accumulate$(2^p - 1, (k-1)2^p - 1, 2^p, \mathbf{false})$

return φ_n

METHOD Product4_Series_Rep(C).**accumulate**$(i, j, k, flag)$

INPUT: Indices i, j, k and a flag $flag$.
ACTION: $\xi = f_{i\cdots i+k} g_{j\cdots j+k} + f_{j\cdots j+k} g_{i\cdots i+k}$ is added to $\varphi_{i+j\cdots i+j+2k-1}$.
If $flag$ holds, then ξ is also added to $\psi_{i+j\cdots i+j+2k-1}$.

$\xi := (f_{i\cdots i+k} + f_{j\cdots j+k}) \star (g_{i\cdots i+k} + g_{i\cdots i+k}) - \psi_{2i\cdots 2i+2k-1} - \psi_{2j\cdots 2j+2k-1}$
$\varphi_{i+j\cdots i+j+2k-1}$ += ξ
if $flag$ **then** $\psi_{i+j\cdots i+j+2k-1}$ += ξ

<table>
<tr><td>⋮</td><td>⋮</td><td colspan="2">⋮</td><td colspan="4">⋮</td><td></td></tr>
<tr><td>g_6</td><td>6</td><td colspan="2" rowspan="2">6</td><td colspan="2" rowspan="2">8</td><td>6</td><td>6</td><td rowspan="4">$\cdots$</td></tr>
<tr><td>g_5</td><td>5</td><td>5</td><td>6</td></tr>
<tr><td>g_4</td><td>4</td><td colspan="2" rowspan="2">4</td><td>4</td><td>4</td><td colspan="2" rowspan="2">8</td></tr>
<tr><td>g_3</td><td>3</td><td>3</td><td>4</td></tr>
<tr><td>g_2</td><td>2</td><td>3</td><td>2</td><td colspan="2" rowspan="2">4</td><td colspan="2" rowspan="2">6</td><td rowspan="2">$\cdots$</td></tr>
<tr><td>g_1</td><td>1</td><td>1</td><td>3</td></tr>
<tr><td>g_0</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>$\cdots$</td></tr>
<tr><td>$\times$</td><td>f_0</td><td>f_1</td><td>f_2</td><td>f_3</td><td>f_4</td><td>f_5</td><td>f_6</td><td>$\cdots$</td></tr>
</table>

Figure 4. A variant of fast relaxed multiplication

4.4.2. Truncation

Assume that we want to compute the first n terms of a power series and that we know that we do not need any more terms. Then the relaxed algorithms from the previous sections have the disadvantage that they do more computations than needed, since the computations of the next n coefficients are already anticipated. There are two approaches to this problem.

In the first approach, we implement a class of "truncated product series". Such a series has a field ν which contains the truncation order and no computations beyond this order are allowed and anticipated. Furthermore, such a series contains an additional method to increase the truncation order and to perform the non effectuated premature computations. When applied to fast relaxed multiplication, we have illustrated in figure 5 the truncated computation at order 12.

In the second approach, we do not have an additional method to increase ν. Instead, we adopt the convention that, as soon as we wish to compute the n-th term of the product series, then we increase ν to n if necessary. This approach has the advantage that the user interface does not change. However, one should be aware that a sequential computation of the first n terms of the product will have the same complexity as in the case of naive lazy multiplication. Therefore, if the user knows beforehand that he needs n terms, then he should first compute the last term, before retrieving the others.

4.4.3. Inlining

For applications in numerical analysis, it is interesting to consider the case when C is a ring of floating point numbers of low precisions and when the expansion order is small. Then one would like to use truncated relaxed DAC-multiplication, since this method has a good complexity for small orders. However, the overhead of the method becomes much to high in this case, due to recursive function calls and memory allocations. Nevertheless, the overhead can significantly be reduced by "unrolling" the whole process. This means that a buffer is allocated at the start for all premature and temporary results and that the computations at each stage are performed "inline".

Let us give an example of how to do program this for order 8. In practice, the program should rather be generated automatically as a function of the (maximal) order. The product class is given by

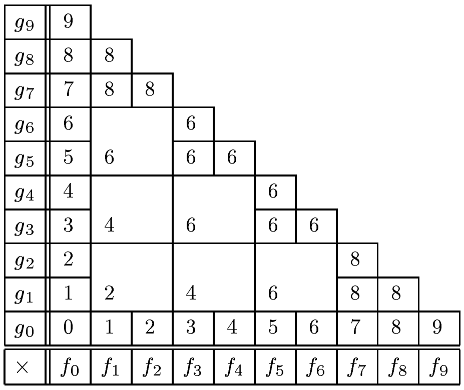

Figure 5. Truncated fast relaxed multiplication at order 10

CLASS $\mathsf{Product5\_Series\_Rep(C)} \triangleright \mathsf{Series\_Rep(C)}$

f, g : $\mathsf{Series(C)}$
ψ : $\mathsf{TPS(C, 5)}$

The constructor takes the two multiplicands on input and stores them in f and g. We also set $\varphi := 0_{0 \cdots 8}$, $\psi := 0_{0 \cdots 5}$.

METHOD $\mathsf{Product5\_Series\_Rep(C)}.\mathbf{next}()$
OUTPUT: The next coefficient $(fg)_n$ assuming that $n < 8$.

I*. [Separate cases]
if $n = 0$ **then go to I0**
$\vdots$
if $n = 7$ **then go to I7**
error "n too high"

I0. return $f_0 g_0$

I1. $\varphi_2 := f_1 g_1$
return $(f_0 + f_1)(g_0 + g_1) - \varphi_0 - \varphi_2$

I2. $\psi_0 := \varphi_2$
$\varphi_4 := f_2 g_2$
$\psi_1 := f_0 + f_2$
$\psi_2 := g_0 + g_2$
return $\varphi_2 + \psi_1 \psi_2 - \varphi_0 - \varphi_4$

I3. $\varphi_6 := f_3 g_3$
$\varphi_5 := (f_2 + f_3)(g_2 + g_3) - \varphi_4 - \varphi_6$
$\psi_3 := f_1 + f_3$
$\psi_4 := f_2 + f_4$
$\varphi_4 := \psi_3 \psi_4 - \psi_0 - \varphi_6$
return $(\psi_1 + \psi_3)(\psi_2 + \psi_4) - \varphi_1 - \varphi_5$

I4. $\psi_0 := f_0 g_4$
$\psi_1 := f_4 g_0$
return $\varphi_4 + \psi_0 + \psi_1$

I5. $\psi_2 := f_1 g_5$
$\psi_3 := f_5 g_1$
return $\varphi_5 + (f_0 + f_1)(g_4 + g_5) + (f_4 + f_5)(g_0 + g_1) - \psi_0 - \psi_1 - \psi_2 - \psi_3$

I6. **return** $\varphi_6 + \psi_2 + \psi_3 + f_0 g_6 + f_2 g_4 + f_4 g_2 + f_6 g_0$

I7. **return** $f_0 g_7 + f_1 g_6 + f_2 g_5 + f_3 g_4 + f_4 g_3 + f_5 g_2 + f_6 g_1 + f_7 g_0$

Although the size of inline programs tends to grow rapidly as a function of the order, they should remain acceptable due to the fact that we only consider small orders. In the case when multiplication in C is really fast with respect to addition (for instance, when using "machine doubles"), it is possible to adapt the strategy, so that the trick (3.1) is only applied for $2^p \times 2^p$ multiplications with sufficiently large p. Numerical experiments by A. Norman tend to show that inline relaxed multiplication becomes more efficient for orders $\geqslant 32$.

4.5. Applications

4.5.1. Implicit series

The main application of the relaxed multiplication algorithm is the efficient expansion of power series solutions to certain functional equations, mainly ordinary and partial differential equations. Therefore, it is convenient to introduce the class $\mathsf{Implicit\_Series(C)}$, whose instances are pointers to the representation class

Class $\mathsf{Implicit\_Series\_Rep(C)} \triangleright \mathsf{Series\_Rep(C)}$

I : $\mathsf{TPS(C)}$
eq : $\mathsf{Series(C)}$

Here I contains the initial conditions ($\sharp I$ in number) and eq the implicit equation which yields the remaining coefficients. The constructor sets $I := 0_{0 \cdots 0}$ and $eq :=$ **null**. The n-th coefficient is computed as follows:

Method $\mathsf{Implicit\_Series\_Rep(C)}$.**next**()
Output: The next, n-th coefficient of the series.

if $eq =$ **null then error** "equation not set"
if $n < \sharp I$ **then return** I_n
else return eq_n

Remark. We notice that in low level languages, implicit series have to be treated with care from a memory management point of view. When using a reference counting technique for the copying of series, one needs to reset eq to **null** after using the implicit series; otherwise, cyclic dependencies might fool the reference counter. In high level computer algebra systems this problem usually does not occur, because the garbage collector is sufficiently powerful to recover non used memory automatically.

4.5.2. Ordinary differential equations

The use of the class Implicit_Series(C) is well illustrated by an example. Consider the system of ordinary differential equations

$$\begin{cases} f' = fg; \\ g' = f + g, \end{cases}$$

with initial conditions $f(0) = g(0) = 1$. Then the following piece of code computes the n-th coefficient of f:

$f :=$ **new** Implicit_Series(C)
$g :=$ **new** Implicit_Series(C)
$f.I := 1_{0\cdots 1}$
$g.I := 1_{0\cdots 1}$
$f.eq := \int f \times g$
$g.eq := \int f + g$
$c := f_n$
$f.eq :=$ **null**
$g.eq :=$ **null**

In a similar fashion, relaxed multiplication can for instance be used to solve systems of algebraic differential equations, by rewriting the equations in integral form like in (3.5). Although we loose a factor $\log n$ in the asymptotic complexity with respect to Brent and Kung's zealous algorithm, the relaxed approach has two advantages:

- We may directly treat systems of o.d.e.'s.
- The constant factor in the asymptotic complexity depends linearly on the size of the equation, when rewritten in its integral form.

As to the second advantage, we notice that Brent and Kung's algorithm is exponential in the order r of the equation. Therefore, our algorithm is more efficient in practice except for particularly low orders.

4.5.3. Other functional equations

The relaxed multiplication algorithm can also be used to solve more general functional equations, such as

$$s(z) = 1 + z\frac{s(z)^3 + 2s(z^3)}{3}. \tag{4.2}$$

The generating function $s(z)$ which satisfies this equation enumerates the number of stereoisomeres of alcohols of the form $C_nH_{2n+1}OH$ (Pólya, 1937). Theorem 4.1 implies that the asymptotic complexity to compute the first n coefficients of $s(z)$ is $O(n \log^2 n)$, which is much better than the previously best known bound $O(n^2)$. Many other differential difference equations arising in combinatorics and the analysis of algorithms are similar to 4.2 (Flajolet and Sedgewick, 1996); in particular, we mention binary splitting algorithms and differential q-difference equations. In section 5, we will consider even more general equations.

4.5.4. Partial differential equations

Fast relaxed multiplication can also be used to solve non linear partial differential equations, by considering d-dimensional power series as power series with $(d-1)$-dimensional power series as coefficients. Consider for instance the equation

$$\begin{aligned} \frac{\partial f}{\partial y} &= \left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial^2 f}{\partial x^2}\right)^2; \\ f(x,0) &= e^x. \end{aligned}$$

We can compute the coefficient of $x^n y^m$ in $f(x,y)$ using the following piece of code:

$f :=$ **new** $\mathsf{Implicit\_Series}(\mathsf{Series}(\mathsf{C}))$
$f.I :=$ **exp**(x)
$f.eq := \int_y ((\partial_x f)^2 + (\partial_x \partial_x f)^2)$
$c := (f_m)_n$
$f.eq :=$ **null**

Here $x : \mathsf{Series}(\mathsf{C})$, $\int_y = \int$ and ∂_x is implemented trivially. Now we notice that the computation of $x^n y^m$ involves expansion of f_m up till $n+1$ terms, f_{m-1} up till $n+3$ terms and so on until f_0, which is expanded up till $n+2m+1$ terms. Using fast relaxed multiplication in y, the complexity of this computation is therefore bounded by $O(M((n+m)m)\log m)$. Actually, this almost linear theoretical complexity is a general situation and the following theorem is proved similarly:

Theorem 4.2. *Let the classes A_d be defined inductively by*

- $A_0 = \mathsf{C}$.
- *A_d is the class of power series $f \in \mathsf{C}[z_1, \ldots, z_d]$, which satisfy an algebraic differential equation of the form*

$$P\left(\left(\frac{\partial^{k_1+\cdots+k_d} f}{\partial z_1^{k_1} \cdots z_d^{k_d}}\right)_{k_1,\cdots,k_d}\right),$$

with initial conditions in A_{d-1}, and such that the separant in z_d

$$S_d = \frac{\partial P}{\partial \frac{\partial^r f}{\partial z_d^r}} \left(\left(\frac{\partial^{k_1+\cdots+k_d} f}{\partial z_1^{k_1} \cdots z_d^{k_d}}\right)_{k_1,\cdots,k_d}\right),$$

(where r is highest such that $\frac{\partial^r f}{\partial z_d^r}$ occurs in P) evaluates to an invertible series in $\mathsf{C}[[z_1, \ldots, z_{d-1}]]$ when setting $z_d = 0$.

Given a series f in A_d and integers $n_1, \ldots, n_d \geqslant 0$, the coefficients of $z_1^{k_1} \cdots z_d^{k_d}$ in f with $k_1 < n_1, \ldots, k_n < n_d$ can be evaluated in time $O(M((n_1 + \cdots + n_d) \cdots (n_{d-1} + n_d) n_d) \log(n_1 + \cdots + n_d))$ and space $O(n_1 \cdots n_d)$. □

Multiplication	10	20	50	100	200	500	1000	2000	1h
Zealous	0.161	0.985	7.202	27.017	92.36	361.19	1135.4	3403	2135
Naive	0.048	0.282	2.533	11.474	48.86	317.00	1283.8		1670
DAC	0.079	0.309	1.428	4.384	13.19	61.35	1887.4		1025
Fast	0.061	0.331	2.162	7.583	25.10	96.20	307.2	959	4095
Variant	0.077	0.347	1.874	5.938	18.34	67.27	193.8	494	*
Truncated	0.047	0.274	1.838	6.782	21.70	98.21	307.5	947	4408

Table 4. Time in seconds to expand $\exp(z \exp z)$ at various orders, using different algorithms and 10000 bits floating point coefficients.

4.6. Benchmarks

We have implemented the zealous multiplication algorithm and several relaxed multiplication algorithms in C++, using integer, rational and floating point arithmetic from GMP (the GNU multiprecision library). Our benchmarks were obtained on a PC running under Linux, with an AMD processor of 200MHz and 64Mb of memory. In our tables, all timings are done in seconds. We aborted the computations after one hour; the maximal number of coefficients which could be computed in this time are shown in the last columns of the tables.

We compared the following multiplication algorithms:

- Zealous: The purely zealous algorithms from section 3.
- Naive: The naive lazy algorithm from section 4.1.
- DAC: The relaxed DAC-multiplication algorithm from section 4.2.
- Fast: Fast relaxed multiplication from section 4.3.
- Variant: The variant of fast relaxed multiplication from section 4.4.1.
- Truncated: Fast truncated relaxed multiplication, as sketched in section 4.4.2.

In table 4, we considered the expansion of $\exp(z \exp z)$, using high precision floating point numbers, so that multiplication in C has a high, but fixed cost. Not surprisingly, all relaxed algorithms do asymptotically better than lazy multiplication (except for DAC-multiplication, which starts swapping for high orders). The threshold for FFT-multiplication being high, we observe a sesquilinear asymptotic complexity. In the future, when GMP will support FFT-multiplication, even higher gains should be achievable (see section 6.1). We also noticed another advantage of fast relaxed multiplication: when sufficient memory is not available, little time is spent on swapping, since most of the computations are done on large blocks of consecutive coefficients in memory.

In table 5, we have computed the expansion of $\exp(z \exp z)$, using rational coefficients. Although the naive algorithm turns out to be the fastest in this case, the results are "fooled" by the fact that rational number arithmetic is not implemented optimally in GMP. Indeed, although DAC-multiplication is used for integers, the gcd-algorithm has a quadratic complexity... Therefore, most time is spent on computing gcd's. Notice also that both in tables 4 and 5, the zealous algorithm is slower than the fast relaxed algorithms, despite its better asymptotic complexity.

In table 6, we considered the expansion of the solution to equation (4.2), using integer coefficients modulo the prime number 1234577. In this case, multiplication in C has a fixed low cost. The threshold for FFT-multiplication is between 2048 and 4096 coefficients, which explains a better asymptotic performance of the fast relaxed algorithms than

Multiplication	10	20	50	100	200	500	1h
Zealous	0.052	0.187	1.294	6.916	50.09	1085.87	686
Naive	0.025	0.072	0.417	2.194	16.62	446.34	845
DAC	0.029	0.101	0.641	3.614	30.45	918.78	758
Fast	0.038	0.125	0.800	4.190	30.96	430.92	767
Variant	0.047	0.155	0.995	5.658	48.78	888.92	703
Truncated	0.026	0.082	0.485	2.308	15.52	342.05	944

Table 5. Time in seconds to expand $\exp(z \exp z)$ at various orders, using different algorithms and rational coefficients.

Multiplication	500	1000	2000	5000	10000	20000	50000	100000	200000	1h
Naive	0.948	2.897	9.541	52.09	198.46	786.5				43312
DAC	0.992	2.603	6.860	24.70	70.24	204.4	873	2624		121561
Fast	0.863	2.101	5.407	20.93	56.25	147.4	547	1355	3370	217087
Variant	0.918	2.055	4.997	16.28	42.10	108.7	411	1014	2480	275967
Truncated	0.766	2.022	5.151	19.03	52.60	145.3	539	1392	3529	203767

Table 6. Time in seconds to compute the number of stereoisomeres of $C_n H_{2n+1} OH$ modulo 1234577 for various n, using different algorithms.

$O(n^{3/2})$. Although our implementation of FFT-multiplication may still be improved, it becomes clear that important gains are already achieved.

In table 7, we again considered the expansion of the solution to equation (4.2), but using integer coefficients. In this case, the sizes of the coefficients in C grow linearly with the expansion order, which explains the rapid growth of the computation times. For suggestions about additional speedups, we refer to sections 6.1 and 6.3.

5. Relaxed composition

A dependency analysis of the composition algorithms from section 3.4 shows that they are, or are almost essentially relaxed, just like DAC-multiplication. Therefore, they admit relaxed analogues with the same asymptotic time complexities (when using a relaxed multiplication algorithm). We will specify these analogues in more detail in this section.

5.1. Fast relaxed composition with polynomials

In this section, we specify the relaxed version of the algorithm from section 3.4.1. Actually, we will compute partial compositions $f_{i \cdots i+p} \circ g$ using the algorithm from sec-

Multiplication	10	20	50	100	200	500	1000	2000	5000	1h
Naive	0.012	0.026	0.087	0.249	0.850	5.485	32.56	297.57		4018
DAC	0.013	0.032	0.113	0.308	0.922	5.635	30.72	235.50		3583
Fast	0.015	0.037	0.131	0.375	1.185	4.853	21.33	134.54	2611	5759
Variant	0.017	0.043	0.151	0.407	1.221	5.558	29.59	215.59	3519	5119
Truncated	0.012	0.028	0.098	0.276	0.871	4.496	19.95	129.23	2295	5862

Table 7. Time in seconds to compute the number of stereoisomeres of $C_n H_{2n+1} OH$ for various n, using different algorithms (and integer coefficients).

tion 3.4.1 and "glue" these partial computations together into a global algorithm as we did in section 4.2.3 for DAC-multiplication.

5.1.1. Partial series

For convenience, we first introduce the class

Class Partial_Series_Rep(C) $\triangleright$ Series_Rep(C)

eq : Series(C)
N : Integer

This class implements series, whose first N coefficients are given by eq and whose other coefficients vanish. Moreover, as soon as the first N coefficients have been computed, eq is released.

More precisely, the constructor of Partial_Series_Rep(C) takes a series and an integer on input, which are assigned to eq and N. We also implement a function

$$\textbf{partial} : \mathsf{Series}(\mathsf{C}) \times \mathsf{Integer} \rightarrow \mathsf{Series}(\mathsf{C}),$$

which takes eq and N on input and returns **new** Partial_Series_Rep(C)(eq, N). The coefficients of a partial series are computed as follows:

Method Partial_Series_Rep(C).**next**()
Output: The next coefficient of the partial series.

if n is a power of two **then** $\varphi := \varphi_{0\cdots\max(2n,\sharp\varphi)}$
if $n = N$ **then** $eq :=$ **null**
if $n \geqslant N$ **then return** 0
return eq_n

5.1.2. Partial right composition with polynomials

In what follows, p will always be a power of two. The algorithm **partial_compose_pol** below computes the partial composition of $f_{i\cdots i+p}$ with g. We assume that the powers $g, g^2, g^4, \ldots, g^{p/2}$ have been computed elsewhere and stored in a hashtable H.

Algorithm **partial_compose_pol**(f, i, p, q, H)
Input: f : Series(C), integers i, p and a hashtable H,
such that $H[i]$ contains g^i for $i = 1, 2, 4, \ldots, p/2$.
Output: The right composition of $f_{i\cdots i+p}$ with g.

if $p = 1$ **then return** f_i
$h_* :=$ **partial_compose_pol**$(f, i, p/2, q, H)$
$h^* :=$ **partial_compose_pol**$(f, i + p/2, p/2, q, H)$
$h := h_* + ((h^* \times (H[p/2] \gg (p/2))) \ll (p/2))$
return partial$(h, (p-1)(q-1)+1)$

It is also convenient to have the following variant of **partial_compose_pol** in order to extend previous computations:

ALGORITHM **partial_compose_pol**(h_*, p, q, H)

INPUT: A previous partial composition $h_* = f_{0\cdots p/2} \circ g$ and the hashtable H with the powers of g.

OUTPUT: The right composition of $f_{0\cdots p}$ with g.

$h^* :=$ **partial_compose_pol**$(f, p/2, p/2, q, H)$
$h := h_* + ((h^* \times (H[p/2] \gg (p/2))) \ll (p/2))$
return partial$(h, (p-1)(q-1)+1)$

5.1.3. Right composition with polynomials

The representation class Compose_Polynomial_Rep(C) corresponds to the total composition of f with a polynomial g:

CLASS Compose_Polynomial_Rep(C)

f, h : Series(C)
H : Hash_Table(Integer, Series(C))
q : Integer

The constructor takes $\overline{f}, g$: Series(C) and an integer $\overline{q}$ as arguments and initializes $f := \overline{f}, H[1] := g, q := \overline{q}$ and $h :=$ **partial_compose_pol**$(f, 0, 1, q, H)$; the other entries of H are undefined at initialization.

In order to compute the coefficients of $f \circ g$, we use the partial composition algorithm, but we double the order p each time when n becomes a power of two.

METHOD Compose_Polynomial_Rep(C).**next**()

OUTPUT: The next coefficient $(f \circ g)_n$.

if n is a power of two **then**

  if $n > 1$ **then** $H[n] := H[n/2] \times H[n/2]$
  $h :=$ **partial_compose_pol**$(h, 2n, q, H)$

return h_n

5.1.4. Complexity analysis

THEOREM 5.1. *There exists a relaxed right composition algorithm for formal power series f by polynomials g, which computes n terms of $f \circ g$ in time $O(qM^*(n) \log n)$ and space $O(nq \log q)$.*

PROOF. We may assume without loss of generality that n is a power of two. Then the estimation for the time complexity is clear, since we perform the same constant operations as in the zealous case.

In order to determine the space complexity, we have to estimate the space which is occupied by the instances $h_{i,p}$ of Partial_Series_Rep(C), which correspond to the compositions $f_{i\cdots i+p} \circ g$ (here p is a power of two and i a multiple of p). These instances can be organized in a binary tree with root $h_{0,n}$ and such that the children of $h_{i,p}$ are $h_{i,p/2}$ and $h_{i+p/2,p/2}$ (for $p > 1$).

We distinguish the following types of instances $h = h_{i,p}$:

I. Active instances: $h.eq \neq$ **null** and $h.n > 0$.
II. Latent instances: $h.eq \neq$ **null** but $h.n = 0$.
III. Completed instances: $h.eq =$ **null**.

We observe that each latent instance occupies $O(1)$ memory space. In total, they therefore occupy $O(n)$ memory space. Each remaining instance occupies $O(\min(pq, n))$ memory. Furthermore, the parent of a completed instance is necessarily active, so that the completed instances do not occupy more than twice as much memory as the active ones.

Now for given $p|n$, consider the instances $h_{0,p}, \ldots, h_{n-p,p}$. Each instance $h_{i,p}$ contributes to the coefficients of $f \circ g$ between i and $i + (p-1)(q-1) + 1$. Hence, if the instance $h_{i,p}$ is active at stage n, then $i \leqslant n < i + pq$. The number of such instances is therefore bounded by q. Hence, the total amount of memory occupied by the active instances is bounded by

$$O\left(\sum_{p|n} q \min(n, pq)\right) = O(nq \log q),$$

as desired. □

5.2. Fast relaxed composition with algebraic functions

5.2.1. The class $\mathsf{Algebraic\_Series}(\mathsf{C})$

Let g be as in section 3.4.2. The algorithm for relaxed right composition with polynomials is easily adapted to the case of right composition with g, by introducing a suitable relaxed analogue $\mathsf{Algebraic\_Series}(g, \mathsf{C})$ of the class $\mathsf{Algebraic\_TPS}(g, \mathsf{C})$ from section 3.4.2.

An instance of $\mathsf{Algebraic\_Series}(g, \mathsf{C})$ consists of d series $F_0, \ldots, F_{d-1}$ and an integer k_F. The following functions are easily implemented:

- An binary powering algorithm to compute and remember $P_d^{k_F}$ in a hashtable P_{C}.
- An addition algorithm for $\mathsf{Algebraic\_Series}(g, \mathsf{C})$.
- A multiplication algorithm for $\mathsf{Algebraic\_Series}(g, \mathsf{C})$.
- The analogue of the algorithm **partial** from section 5.1.1 for $\mathsf{Algebraic\_Series}(g, \mathsf{C})$.
- A function **convert** which converts an instance of $\mathsf{Algebraic\_Series}(g, \mathsf{C})$ back to a series in $\mathsf{Series}(\mathsf{C})$.

5.2.2. Partial right composition

The analogues of the algorithms **partial_compose_alg** from section 5.1.2 are given by

Algorithm **partial_compose_alg**(f, i, p, q, H)
Input: $f : \mathsf{Series}(\mathsf{C})$, integers i, p and a hashtable H,
such that $H[i]$ contains g^i for $i = 1, 2, 4, \ldots, p/2$.
Output: The right composition of $f_{i \cdots i+p}$ with g.

if $p = 1$ **then return** f_i
$h_* :=$ **partial_compose_alg**$(f, i, p/2, q, H)$
$h^* :=$ **partial_compose_alg**$(f, i + p/2, p/2, q, H)$
$h := h_* + ((h^* \times (H[p/2] \gg (p/2))) \ll (p/2))$
return partial$(h, (p-1)(q-1) + 1 - \max(p-d, 0)v)$

ALGORITHM **partial_compose_alg**(h_*, p, q, H)
INPUT: A previous partial composition $h_* = f_{0\cdots p/2} \circ g$ and the hashtable H with the powers of g.
OUTPUT: The right composition of $f_{0\cdots p}$ with g.

$h^* :=$ **partial_compose_alg**$(f, p/2, p/2, q, H)$
$h := h_* + ((h^* \times (H[p/2] \gg (p/2))) \ll (p/2))$
return partial$(h, (p-1)(q-1) + 1 - \max(p-d, 0)v)$

5.2.3. Right composition with algebraic functions

The analogue of the class Compose_Polynomial_Rep(C) is given by

CLASS Compose_Algebraic_Rep(g, C)
f, h : Series(C)
h^{alg} : Algebraic_Series(g, C)
H : Hash_Table(Integer, Algebraic_Series(g, C))
q : Integer

The constructor takes $\overline{f}$ and an integer $\overline{q}$ as arguments and initializes $f := \overline{f}, H[1] := g, q := \overline{q}$, $h^{alg} :=$ **partial_compose_alg**$(f, 0, 1, q, H)$ and $h :=$ **convert**(h^{alg}); the other entries of H are undefined at initialization.

METHOD Compose_Algebraic_Rep(g, C).**next**()
OUTPUT: The next coefficient $(f \circ g)_n$.

if n is a power of two **then**

  if $n > 1$ **then** $H[n] := H[n/2] \times H[n/2]$
  $h^{alg} :=$ **partial_compose_alg**$(h^{alg}, 2n, q, H)$
  $h :=$ **convert**(h^{alg})

return h_n

The following theorem is proved in a similar way as theorem 5.1:

THEOREM 5.2. *Let g be as in section 3.4.2. There exists a relaxed right composition algorithm for formal power series f by g, which computes n terms of $f \circ g$ in time $O(qd^2(q-v)M^*((1+v)n))$ and space $O(qdn(v + \log(q-v))))$.* □

5.3. Fast relaxed composition when C is a divisible ring

Assume that C is a divisible ring. The representation class Compose_Rep(C) below corresponds to the composition of two arbitrary power series f, g : Series(C).

Class Compose_Rep(C) ▷ Series_Rep(C)

f, g, h : Series(C)

The constructor initializes f and g with the arguments and $h :=$ **null**.

The "relaxation" of Brent and Kung's algorithm from section 3.4.3 gives rise to a new problem: we would like to use the relaxed algorithm for right composition with polynomials in order to compute $f \circ g_*$ in (3.18). But as n increases, the value of g_* changes very often, and each time this happens, we have to start over the relaxed computation of $f \circ g_*$.

Therefore, we should neither change g_* to often, so that we make efficient use of the polynomial right composition algorithm, nor to little, so that the power series expansion of $f \circ (g_* + g^*)$ can still be done quickly. A good compromise is to let $q = 2^{p+1}$ be the largest power of two with $p4^{p-1} \leqslant n$.

Method Compose_Rep(C).**next**()

Output: The next coefficient of $(f \circ g)_n$.

C1. [n small]
  if $n = 0$ **then return** f_0
  if $n = 1$ **then return** $f_1 g_1$
  if $n = 2$ **then return** $f_2 g_1^2 + f_1 g_2$
  if $n = 3$ **then return** $(f_3 g_1^2 + 2 f_2 g_2) g_1 + f_1 g_3$

C2. [Compute q and r]
  $p := \max\{p \in \mathbb{N} | p4^{p-1} \leqslant n\}$
  if $n \neq 4$ **and** $p4^{p-1} \leqslant n - 1$ **then return** h_n $n' := (p+1)4^p$
  $q := 2^{p+1}$
  $r := \lceil n'/q \rceil$
  while $r > n$ **do** $q := 2q, r := \lceil n'/q \rceil$

C3. [Adjust h]
  $g_* := g_{0 \cdots q}$
  $g^* := g - g_*$
  $D :=$ **new** Compose_Polynomial_Rep$(f^{(r-1)}, g_*)$
  $h := D := D/(r-1)!$
  for $i := r - 1$ **downto** 1 **do**
    $D := f_{i-1} + \int((iD) \times g_*')$
    $h := D + ((h \times (g^* \gg q)) \ll q)$
  return h_n

Theorem 5.3. *There exists a relaxed composition algorithm for formal power series f, g, which computes n terms of $f \circ g$ in time $O(M^*(n)\sqrt{n \log n})$ and space $O(n\sqrt{n} \log n)$.*

Proof. For n between two successive changes of q, the time and space complexities of the computation of $f \circ g_*$ are $O(M^*(n)\sqrt{n \log n})$ resp. $O(n\sqrt{n} \log n)$, by theorem 5.1.

The Taylor expansion of $f \circ g$ contains $O(\sqrt{n \log n})$ terms. Hence, the complexity of its evaluation (which requires only additions, derivations, multiplications and divisions, which are all performed in time $O(M^*(n))$) is again $O(M^*(n)\sqrt{n \log n})$. The Taylor expansion requires $O(n\sqrt{n \log n})$ space.

Now observe that q changes at most once for n between a given number n_0 and $2n_0$. Hence, for general values of n, the time complexity is bounded by $O(M^*(n)\sqrt{n \log n} + M^*(n/2)\sqrt{(n \log n)/2} + \cdots + M^*(1)) = O(M^*(n)\sqrt{n \log n})$ and the space complexity by $O(n\sqrt{n} \log n)$. □

Remark. In section 4.4.2, we have shown how to gain a constant factor on the time and space complexities for relaxed multiplication if an *a priori* bound for the expansion order has been specified. A similar optimization can be carried out here: if the maximal order is known beforehand, then we may choose q and r as in section 3.4.3, thereby avoiding certain recomputations.

5.4. Fast relaxed composition for rings C of finite characteristic

The formulas (3.19) and (3.20), combined with the Chinese remainder theorem, yield straightforward relaxed composition algorithms when C has finite characteristic. We will just treat the case when the characteristic p of C is prime; the general case is left as an exercise.

The following functions are easily implemented

- **compose_p** : $\mathsf{Series}(\mathsf{C}) \to \mathsf{Series}(\mathsf{C}); f \mapsto f \circ z^p$.
- **power_p** : $\mathsf{Series}(\mathsf{C}) \to \mathsf{Series}(\mathsf{C}); f \mapsto f_0^p + f_1^p z + f_2^p z^2 + \cdots$.
- **progression_p** : $\mathsf{Series}(\mathsf{C}) \times \mathsf{Integer} \to \mathsf{Series}(\mathsf{C}); (f, i) \mapsto f_i + f_{i+p} z + f_{i+2p} z^2 + \cdots$.

Now the class Compose_Rep(C) corresponds to the composition of f and g:

Class Compose_prime_Rep(C) ▷ Series_Rep(C)

  f, g, h : Series(C)

The constructor initializes f and g with the arguments and $h :=$ **null**.

Method Compose_prime_Rep(C).**next**()
Output: The next coefficient of $(f \circ g)_n$.

C1. [Easy case]
  if $n = 0$ **then return** f_0
  if $n > 1$ **then return** h_n

C2. [Set up equation]
  $h := 0$
  for $i := p - 1$ **downto** 0 **do**
    $t :=$ **compose_p**(**compose**(**progression_p**(f, i), **power_p**(g)))
    $h := ((h \times (g \gg 1)) \ll 1) + t$
  return h_n

THEOREM 5.4. *Assume that* C *has prime characteristic* p. *Then there exists a relaxed composition algorithm for formal power series* f, g, *which computes* n *terms of* $f \circ g$ *in time* $O((p/\log p)M^*(n)\log n)$ *and space* $O((p/\log p)n\log n)$.

PROOF. The time and space complexities $T(n)$ resp. $S(n)$ satisfy

$$\begin{aligned} T(n) &= pT(n/p) + O(pM^*(n)); \\ S(n) &= pS(n/p) + O(pn). \end{aligned}$$

The complexity bounds follow from these relation. □

As in the zealous case, the above algorithm can be optimized by using the lazy version of Brent and Kung's algorithm for small values of n. For rings C of more general characteristic, the following can be proved:

THEOREM 5.5.

a. *Assume that* C *has prime power characteristic* p^k. *Then there exists a relaxed composition algorithm for formal power series* f, g, *which computes* n *terms of* $f \circ g$ *in time* $O((k^3 p/\log p)M^*(n)\log n)$ *and space* $O((kp/\log p)n\log n)$.
b. *Assume that* C *has general characteristic* r. *Then there exists a relaxed composition algorithm for formal power series* f, g, *which computes* n *terms of* $f \circ g$ *in time* $O((r/\log r)M^*(n)\log n)$ *and space* $O((r/\log r)n\log n)$. □

5.5. APPLICATIONS

5.5.1. FINITE DIFFERENCE EQUATIONS

One of the most interesting applications of theorem 5.2 is that all linear or non linear finite difference equations at infinity (assuming that they have been put in some normal form) can be solved in essentially linear space and time. Consider for instance the equation

$$f(x) = \frac{1}{x}(1 + f(x+1) + f'(x)^2), \tag{5.1}$$

which admits a unique power series solution in $1/x$:

$$f(x) = \frac{1}{x} + \frac{1}{x^2} - \frac{1}{x^4} - \frac{3}{x^6} + O\left(\frac{1}{x^7}\right)$$

Putting $x = 1/z$ and $f(x) = f(1/z) = g(z)$, the equation (5.1) becomes

$$g(z) = z\left(1 + g\left(\frac{z}{1+z}\right) - z^4 g'(z)^2\right). \tag{5.2}$$

Using the initial condition $g(0) = 0$, the first n terms of $g(z)$ can be computed in time $O(n\log^3 n\log\log n)$ by theorem 5.2, when using FFT-multiplication.

5.5.2. Combinatorics

In combinatorics, one also sometimes encounters functional equations, which involve right composition with polynomials or algebraic functions. An example of such an equation is

$$f(z) = z + f(z^2 + z^3).$$

The generating function f counts the number of so called 2-3-trees (Odlyzko, 1982). The coefficients can again be computed in essentially linear time.

5.5.3. General functional equations

Theorem 5.3 can be used to solve any kind of functional equation involving differentiation and composition up till n terms in time $O(n^{3/2} \log^{5/2} n \log \log n)$. An example of such an equation is given by

$$f(z) = z + f(z f(z) + z^2 f'(z)) + z^4 \exp(z f''(z)).$$

Notice that power series reversion is another example.

5.6. Benchmarks

Using the same conventions and multiplication algorithms as in section 4.6, we have tested four composition algorithms:

- Naive: The naive relaxed composition algorithm, using Horner's rule.
- Brent&Kung: The relaxed version of Brent and Kung's algorithm.
- Fast: The almost linear algorithms in case of right composition with polynomials or algebraic functions.
- Bernstein: The relaxed version of Bernstein's algorithm, for coefficient rings of characteristic $p > 0$.

We also implemented truncated versions of these algorithms, which were used each time we used truncated relaxed multiplication.

In table 8, we have considered the expansion of the solution to equation (5.2), where we took the ring of integers modulo a large number p as our coefficient ring. Actually, these timings do not depend on p, whence they can be compared to those from table 9, where p is a small prime number, and where we use the relaxed version of Bernstein's composition algorithm. In table 10, we considered the same equation, using integer coefficients.

6. Suggestions for specific coefficient rings

In the previous sections, we have given asymptotically fast algorithms for the manipulation of formal power series over a "generic ring" C. In practice, C is usually the ring of integers, rational numbers, floating point numbers, etc. or constructed from one these, by considering polynomial rings, rings of formal power series, or quotients.

On the one hand, this makes it possible to exploit the special nature of C in order to gain additional constant factors on the complexities of the relaxed algorithms. These factors may be considerable, but they rely on a clever use of the FFT-transform, as explained in section 6.1. In particular, it is time consuming to write good implementations.

Composition	Multiplication	100	200	500	1000	2000	5000	10000	20000	1h
Naive	Naive	0.537	3.213	43.80	337.1	2647.2				2216
	Fast	1.113	6.187	69.15	459.5	3152.5				2093
	Truncated	0.592	2.857	28.53	169.0	1022.7				3119
Brent&Kung	Naive	0.561	2.065	23.68	96.4	871.1				4148
	Fast	1.067	3.549	34.90	120.0	960.2				4188
	Truncated	0.809	2.650	23.18	75.0	573.8	2905			5628
Fast	Naive	0.406	1.448	8.21	34.2	144.9	1111			8560
	Fast	0.713	2.151	8.13	25.8	83.8	499	1611		16385
	Truncated	0.445	1.366	5.89	19.2	62.9	333	1070	3341	20253

Table 8. Time in seconds to expand the solution to (5.2) at various orders, using different algorithms and integer coefficients modulo 1234577.

p	Multiplication	100	200	500	1000	2000	5000	10000	20000	50000	1h
3	Naive	0.182	0.557	2.535	8.704	31.72	187.4	738	2923		22198
	Fast	0.326	0.894	3.444	10.597	34.23	176.7	617	2214		25809
	Truncated	0.231	0.565	1.955	5.287	14.76	56.8	162	489	3254	50000
11	Naive	0.264	0.855	4.275	15.876	60.77	370.4	1470			15668
	Fast	0.431	1.400	5.901	19.281	66.00	358.4	1287			17025
	Truncated	0.279	0.819	2.940	8.351	24.86	96.1	287	868		31946
37	Naive	0.483	1.678	9.988	39.336	158.56	994.5				9492
	Fast	0.867	2.876	13.866	48.200	173.21	983.8				9990
	Truncated	0.501	1.412	6.181	18.791	58.95	226.0	682	2132		24601

Table 9. Time in seconds to expand the solution to (5.2) at various orders, using the relaxed version of Bernstein's algorithm and integer coefficients modulo p.

On the other hand, for most of the constant fields used in practice, the size of the n-th coefficient of a series tends to grow with n. Analogously, in numerical analysis, the precision of the n-th coefficient of a series tends to decrease with n. Unfortunately, the relaxed algorithms, more than the naive ones, tend to add coefficients of different sizes resp. precisions, which leads to a loss of efficiency or numerical instability. This issue will be treated in more detail in sections 6.2 and 6.3 and some approaches will be suggested.

This section is mainly included to give some hints about how to adapt the theoretical algorithms from the previous sections to particular, frequently used constant rings C. Our presentation will be informal and our suggestions have still to be tried out in practice.

Composition	Multiplication	10	20	50	100	200	500	1000	2000	1h
Naive	Naive	0.018	0.055	0.433	3.129	24.942	456.09			618
	Fast	0.024	0.104	0.979	6.495	47.541	1037.63			514
	Truncated	0.018	0.066	0.547	3.307	22.258	344.23			596
Brent&Kung	Naive	0.026	0.077	0.848	2.881	12.805	195.72			830
	Fast	0.039	0.135	1.764	5.781	23.679	298.51			894
	Truncated	0.028	0.091	1.342	4.747	18.764	220.00			800
Fast	Naive	0.031	0.095	0.431	1.782	7.636	55.15	359.41	3487	2012
	Fast	0.057	0.202	0.955	3.575	13.124	59.69	274.84	1722	2049
	Truncated	0.033	0.116	0.572	2.206	8.439	47.24	229.90	1572	2307

Table 10. Time in seconds to expand the solution to (5.2) at various orders, using different algorithms and integer coefficients.

6.1. Generalizing the fast Fourier transmorm

In most of the actual computer algebra systems, polynomials, vectors, matrices, etc. over a base ring C are implemented in a generic way. Unfortunately, this approach make it hard to fully exploit the fast Fourier transformation.

Consider for instance polynomials A and B with large integer coefficients, so that the coefficients are multiplied using the FFT. Then in order to compute AB, we may first transform the coefficients of A and B, next multiply the *transformed* polynomials and finally transform back. In this approach we only have to compute the FFT of each coefficient of A and B once, so that we gain with respect to the generic polynomial multiplication algorithm. Moreover, this optimization can be used recursively for multivariate polynomials over the integers, and each time we increase the number of variables, we gain a constant factor with respect to the generic approach.

This example shows that we have to rethink the basic arithmetic operations for the most elementary generic computer algebra types, in order to obtain maximal efficiency for large input sizes. For this purpose, let us reformulate FFT-multiplication in an abstract way for elements A and B in a ring C.

- We first have to "transform" the ring C into $\hat{\mathsf{C}}_{A,B}$.
- We next compute the fast Fourier transforms $\hat{A}, \hat{B} : \hat{\mathsf{C}}_{A,B}$ of A and B.
- We multiply $\hat{A}$ and $\hat{B}$ in $\hat{\mathsf{C}}_{A,B}$, yielding $\hat{C}$.
- We transform $\hat{C}$ back into the product C of A and B.

The "transformed ring" $\hat{\mathsf{C}}_{A,B}$ is depends on certain characteristics of A and B, such as size or degree. In the algorithm from section 3.1.2, we would have $\mathsf{C}[x]\widehat{/(x^n+1)}_{A,B} = \mathsf{C}[y]/(y^m+1)$. The multiplication is represented schematically by the following diagram:

$$\begin{tikzcd} A, B : \mathsf{C} \arrow[r, "\mathrm{FFT}"] \arrow[d, "\times_{\mathsf{C}}"] & \hat{A}, \hat{B} : \hat{\mathsf{C}}_{A,B} \arrow[d, "\times_{\hat{\mathsf{C}}_{A,B}}"] \\ AB : \mathsf{C} & \hat{A}\hat{B} : \hat{\mathsf{C}}_{A,B} \arrow[l, "\mathrm{FFT}^{-1}"'] \end{tikzcd}$$

It is not hard to see how such an abstract FFT-transform might be implemented for elementary computer algebra types such as integers, floating point numbers, polynomials, matrices, etc. However, there are three different approaches in the case of dense polynomials, which are detailed below. The choice of the fastest approach may depend on the system. A calibration function should be implemented to find the optimal one for a given input size.

6.1.1. Usual multiplication with transformed coefficients

Let C be a constant ring for which an abstract FFT-transform has been implemented and consider the ring $\mathsf{C}[x]$ of dense polynomials over C. Given such a polynomial $A = A_d x^d + \cdots + A_0$, we may transform it using

$$\hat{A} = \widehat{A_d} x^d + \cdots + \widehat{A_0} : \widehat{\mathsf{C}[x]} = \hat{\mathsf{C}}[x]$$

and use a generic multiplication algorithm in $\hat{\mathsf{C}}[x]$. Of course, the precise ring $\hat{\mathsf{C}}$ depends on the sizes of the polynomials one wishes to multiply.

For small degrees, this approach yields the best results. For instance, the cost of multiplying two polynomials of degree 1 is strictly less than three constant multiplications (when the coefficients have approximately the same sizes). For other small degrees n, two polynomials can be multiplied using $2n+1$ constant operations using Toom-Cook's algorithm (Toom, 1963b; Cook, 1966; Knuth, 1981). However, the overhead of this algorithm grows rapidly, which makes this approach less interesting for higher degrees.

6.1.2. Reduction to the base ring

For rings C of characteristic zero, another approach is to take

$$\hat{A} = \widehat{A(2^N)} : \widehat{\mathsf{C}[x]} = \hat{\mathsf{C}},$$

for a sufficiently large N. For example, in base 10, this corresponds to multiplying polynomials as follows:

$$\begin{tikzcd}
(101x+213) \times (219x+173) \arrow[r, "\mathrm{FFT}"] \arrow[d] & 101000212 \times 219000173 \arrow[d, "\text{FFT-multiply}"] \\
22119x^2 + 64120 + 36849 & 22119064120036849 \arrow[l, "\mathrm{FFT}^{-1}"']
\end{tikzcd}$$

If A and B are polynomials of degree n, whose coefficients are very large and of approximately the same size s, then the sizes of $A(2^N)$ and $B(2^N)$ are both approximately $(2n+1)s$. Hence, AB can be computed in roughly the same time as $2n+1$ coefficient multiplications. Moreover, contrary to the method from the previous section, the additional overhead is low, even for large n.

Another advantage of the present method is that it "smoothes" the graph with the computation time as a function of the input size. Indeed, when using FFT-multiplication, each time that extra roots of unity are needed (i.e. when doubling the input size), a sudden increase in the computation time is observed. The present method reduces this phenomenon.

Remark. Notice the interesting philosophy behind the method: usually, complex problems (such as multiplying polynomials) are reduced to many small simple problems (multiplication of coefficients). Here, we rather reduce the complex problem to a huge, but simple problem, and we make use of the fact that we have an asymptotically efficient method for the huge simple problem.

6.1.3. Multivariate fast Fourier transforms

Still another method is based on the observation that in order to multiply polynomials in $\hat{\mathsf{C}}[x]$, the ring $\hat{\mathsf{C}}$ already has many 2^N-th roots of unity. Hence, after a first transformation $A_d x^d + \cdots + A_0 \to \widehat{A_d} x^d + \cdots + \widehat{A_0}$ as in section 6.1.1, FFT-multiplication becomes interesting much earlier than for a generic polynomial ring.

Although the multiplication scheme based on this method is slightly slower for small degrees (for instance, we need 4 "constant multiplications" in order to multiply two first degree polynomials), the method is virtually linear from then on. Especially when multiplication in $\hat{\mathsf{C}}$ becomes expensive with respect to the fast Fourier transformation, this method may be an interesting alternative for moderate degrees n.

Remark. We also suggest to use this method for integer multiplication itself. Indeed, Schönhage-Strassen's algorithm (Schönhage and Strassen, 1971) reduces the multiplication problem for integers modulo $2^{2^N}+1$ to the problem of multiplying polynomials of degrees $\leqslant n$ in $\mathbb{Z}/(2^{2^n}+1)\mathbb{Z}[x]$, where $n \approx N/2$. However, for large N (that is $N \gtrapprox 20$ on actual machines), the modular multiplication step of numbers modulo $2^{2^n}+1$ becomes far more expensive than the transformation step. For such N, we therefore suggest to use polynomials in $\mathbb{Z}/(2^{2^n}+1)\mathbb{Z}[x,y]$ of degrees $\leqslant n$ in x and y instead, where $n \approx N/3$.

6.2. On the numerical instability of relaxed algorithms

In this section, we study the numerical stability of the different relaxed multiplication algorithms for power series with floating point coefficients. For this purpose, it is important two distinguish two types of applications.

For applications to numerical analysis, such as the analytic continuation of holomorphic functions, the coefficients are usually known with a high precision, that is, a precision which is linearly dependent on the required expansion order. This enables us to evaluate the series close to the origin up till a number of digits which is linearly dependent on the expansion order. For such applications, a sublinear or even a small linear precision loss will not change the asymptotic complexity of the evaluation of the series up till n digits.

For other applications, such as the random generation of combinatorial structures (Flajolet *et al.*, 1994; Denise *et al.*, 1998; Denise and Zimmermann, 1999), we are interested in the coefficients themselves and we require a given, small number of digits after the decimal point. On the one hand, the fact that we want many terms using a low precision makes this application vulnerable for numerical instability. On the other hand, the coefficients of the series are often all positive with nice asymptotic properties in this case. Under additional hypotheses, we may therefore hope to estimate the precision loss.

6.2.1. Sources of numerical instability

There are two main sources of numerical instability when multiplying formal power series. The first source is "massive cancellation" of coefficients like the increase of the radius of convergence. An example is given by

$$\sqrt{1-z}\sqrt{1-z} = 1-z.$$

This source is intrinsic and no particular numerical multiplication method will be able to avoid it.

The second source of numerical instability is encountered, when coefficients of different magnitudes are added up in order to speed up the product computation. Consider for instance the computation of

$$P = (1.000\cdot 10^0 + 1.000\cdot 10^{-5}z)\times(1.000\cdot 10^0 + 1.000\cdot 10^{-5}z)$$

using DAC-multiplication:

$$\begin{aligned}
1.000\cdot 10^0 &= (1.000\cdot 10^0)\times(1.000\cdot 10^0)\\
1.000\cdot 10^{-10} &= (1.000\cdot 10^{-5})\times(1.000\cdot 10^{-5})\\
1.000\cdot 10^0 &= (1.000\cdot 10^0 + 1.000\cdot 10^{-5})\times(1.000\cdot 10^0 + 1.000\cdot 10^{-5}).
\end{aligned}$$

We obtain

$$P = 1.000\cdot 10^0 + 0.000\cdot 10^0 z + 1.000\cdot 10^{-10}z^2.$$

Hence, the addition $1.000\cdot 10^0 + 1.000\cdot 10^{-5}$ is responsible for the precision loss.

6.2.2. Increasing the numerical stability

In the frequent case when we multiply convergent power series f and g, we can often avoid this problem by "normalizing" blocks $f_{i\cdots i'}$ and $g_{j\cdots j'}$ (with $l = i' - i = j' - j$) of successive coefficients before multiplying them. Indeed, in the convergent case, the exponents of the coefficients $f_i, \ldots, f_{i'-1}$ resp. $g_j, \ldots, g_{j'-1}$ approximately form an arithmetic progression; at least, they are bounded by such a progression.

Hence, by looking at these exponents, we determine the "least approximate minimal radius of convergence" $\tilde{r}$ of f and g: for certain constants F and G we have

$$\begin{aligned} f_k &\leqslant F/\tilde{r}^k \quad (i \leqslant k < i'); \\ g_k &\leqslant G/\tilde{r}^k \quad (j \leqslant k < j'), \end{aligned}$$

where the inequalities are (approximate) equalities for at least one k and l, and for at least two k or l. Now we compute

$$h_0 + \cdots + h_{2l-1} = (f_i + \cdots + f_{i'-1}\tilde{r}^{l-1}z^{l-1}) \times (g_j + \cdots + g_{j'-1}\tilde{r}^{l-1}z^{l-1})$$

using any fast multiplication algorithm for polynomials. Then

$$f_{i\cdots i'}g_{j\cdots j'} = h_0 + \cdots + \frac{h_{2l-1}}{\tilde{r}^{2l-1}}z^{2l-1}.$$

This way of computing $f_{i\cdots i'}g_{j\cdots j'}$ increases the numerical stability. For instance, in the example from the previous section, we get $\tilde{r} = 1.000 \cdot 10^5$ and

$$\begin{aligned} h &= 1.000 \cdot 10^0 + 2.000 \cdot 10^0 z + 1.000 \cdot 10^0 z^2; \\ fg &= 1.000 \cdot 10^0 + 2.000 \cdot 10^{-5} z + 1.000 \cdot 10^0 z^{-10}. \end{aligned}$$

Remark. Considering a finite number of coefficients $f_{i\cdots i'}$, the "approximate radius of convergence" of a series f may for instance be computed in linear time, by "traversing" the convex envelope of the logarithms of these coefficients and retaining the longest segment. A slower, but more stable method is obtained by maximizing the quantity

$$\sum_{k=i}^{i'-1} C|f_k/\tilde{r}^k|$$

among all $C \geqslant 0$ and $\tilde{r} > 0$ with

$$\max_{k=i}^{i'-1} C|f_k/\tilde{r}^k| = 1.$$

It would be interesting to find a fast and stable compromise between these two extremes.

6.2.3. Series with positive coefficients and error estimations

In the case when all series we consider have positive coefficients, which is frequently the case in combinatorics and the analysis of algorithms, it is often possible to obtain precise error estimations for the various relaxed algorithms for multiplication and composition.

Let B be the number of significant bits with which we compute. In what follows, when approximating a real number $\tilde{x}$ by a floating point number $x = M \cdot 2^E$ (with $\frac{1}{2} \leqslant M < 1$), we will denote by δ_x the "normalized relative error" we commit, so that

$$\tilde{x} - \delta_x 2^{E-B} \leqslant x \leqslant \tilde{x} + \delta_x 2^{E-B}.$$

For small errors (that is $\delta_x \leqslant 2^{B/2}$), we then have

$$\delta_{x+y} \leqslant \max(\delta_x, \delta_y) + 2; \quad (6.1)$$
$$\delta_{xy} \leqslant \delta_x + \delta_y + 2, \quad (6.2)$$

for positive floating point numbers x and y.

Naive lazy algorithms

Let us first consider the case of a system of differential equations, which has been put into integral form

$$\begin{pmatrix} f_1(z) \\ \vdots \\ f_r(z) \end{pmatrix} = \int \begin{pmatrix} P_1(f_1, \ldots, f_r) \\ \vdots \\ P_r(f_1, \ldots, f_r) \end{pmatrix}, \quad (6.3)$$

where $P_1, \ldots, P_r$ are polynomials with positive coefficients. Then we can expand the solutions using the lazy power series technique. Let $f_{i,n}$ denote the n-th coefficient of f_i. Then the equations (6.1) and (6.2) yield

$$\delta_{f_{i,n}} \leqslant \max_{n_1+\cdots+n_r=n-1} \sum_{j=1}^{r} \delta_{f_{j,n_j}} + O(n), \quad (6.4)$$

for each i, since coefficients of the P_i and the initial conditions $f_{i,0}$ have bounded normalized relative errors. Consequently, putting $E_n = \max_{1\leqslant i\leqslant r} \delta_{f_{i,n}}$, we have

$$E_n \leqslant \max_{n_1+\cdots+n_r=n-1} \sum_{j=1}^{r} E_j + O(n). \quad (6.5)$$

It follows that $E_n = O(n^2)$. This shows that number of erroneous bits in $f_{i,n}$ grows only logarithmically with n.

For functional equations which involve composition, we get similar bounds. For instance, if we compute $(f \circ (zg))_n$ using Horner's rule:

$$(f \circ (zg))_n = f_0 + zg(f_1 + zg(f_2 + \cdots + zg(f_n)\cdots)),$$

we obtain

$$\delta_{(f\circ(zg))_n} \leqslant \max_{i+j_1+\cdots+j_k=n} \delta_{f_i}\delta_{g_{j_1}} + \cdots + \delta_{g_{j_k}} + O(n^2).$$

Hence, (6.4) would now become

$$\delta_{f_{i,n}} \leqslant \max_{\sum n_{j,k}=n-1} \sum_{j,k} \delta_{f_{j,n_{j,k}}} + O(n^2),$$

and we would rather get $E_n = O(n^3)$, which still ensures a logarithmic growth of the number of erroneous bits in the result. For the fast relaxed composition algorithms, a similar growth of the error can be proved, since the symbolic application of the algorithm yields the n-th coefficient of $f \circ g$ as an expression in $f_0, \ldots, f_n, g_1, \ldots, g_n$ and positive rational numbers, using sums and products only.

Fast relaxed multiplication

Let f and g be convergent power series with positive coefficients. Denoting by r_f and r_g the convergence radii of f and g, we define

$$\begin{aligned}\varepsilon_f(n) &= \tfrac{1}{n}\log_2 f_n + \log_2 r_f;\\ \varepsilon_g(n) &= \tfrac{1}{n}\log_2 g_n + \log_2 r_g.\end{aligned}$$

Let us make the "convexity hypothesis" that the sequences $\varepsilon_f(n)$ and $\varepsilon_g(n)$ are convex or concave for sufficiently large n. This is in particular the case if the coefficients f_n and g_n admit asymptotic equivalents in a Hardy fields, such as

$$f_n \sim C(\log n)^\alpha n^\beta r_f^{-n}.$$

We will study the numerical stability of the fast relaxed multiplication algorithm, assuming that we use the normalization procedure from section 6.2.2.

Let us first assume that $r_f = r_g$ and consider the multiplication $f_{i\cdots i'} \times g_{j\cdots j'}$ with the notations from section 6.2.2. The convexity hypothesis implies that the exponents of the normalized coefficients are dominated by $O(|\varepsilon_f(i')| + |\varepsilon_g(j')|)$. Consequently, we loose $O(|\varepsilon_f(i')| + |\varepsilon_g(j')|)$ extra bits of precision in the multiplication $f_{i\cdots i'} \times g_{j\cdots j'}$ with respect to the naive method. This leads to the error estimation

$$\delta_{(fg)_n} \leqslant \max_{i+j=n} \delta_{f_i} + \delta_{g_j} + 2^{O(|\varepsilon_f(n)|+|\varepsilon_g(n)|)} + O(n)$$

for the coefficients of the product fg. This estimation remains valid in the case when $r_f < r_g$ (or $r_f > r_g$), because the extra precision loss in the multiplications is compensated by the exponential decrease of the coefficients of g with respect to those of f (see also the next paragraph).

Now reconsider the system of differential equations (6.3). Assume that the convexity hypothesis is verified for all series g encountered in the relaxed expansion process and let $\varepsilon(n)$ be the sum of the corresponding $|\varepsilon_g(n)|$. Then the equations (6.4) become

$$\delta_{f_{i,n}} \leqslant \max_{n_1+\cdots+n_r=n-1} \sum_{j=1}^{r} \delta_{f_{j,n_j}} + 2^{\varepsilon(n)} + O(n) \tag{6.6}$$

and we obtain

$$E_n = O(2^{\varepsilon(n)} + n^2). \tag{6.7}$$

Indeed ε is ultimately monotonic (by the convexity hypothesis), so that either $2^{\varepsilon(n)} = O(n)$, or $2^{\varepsilon(n)}$ increases towards infinity.

Intuitively speaking, (6.7) means that the precision loss is proportional to the asymptotic behaviour near the most violent dominant singularity encountered in the expansion process. In particular, if all these singularities are algebraic (such as in the example (4.2)), then the precision loss remains logarithmic. This result generalizes to the case of more general functional equations, as in the case of naive multiplication. Finally, a similar growth of the error may be expected in the general case when the coefficients are no longer positive. Indeed, the main obstruction to such a behaviour is massive cancellation of coefficients, which occurs only in very specific situations.

Unequal radii of convergence

Assume that we want to multiply two series f and g with unequal radii $r_f < r_g$ of convergence, which satisfy the convexity hypothesis. Then f_n/g_n decreases exponentially. We will indicate how to use this observation in order to obtain a multiplication algorithm for f with g of time complexity $O(n)$.

During the expansion process of f (resp. g), we heuristically compute its approximate convergence radius $\tilde{r}_f$, based on the knowledge of the first n coefficients. This may for instance be done efficiently by updating $\tilde{r}_f$, each time that n becomes a power of two, by applying a similar algorithm as in section 6.2.2 on the coefficients $f_{n/2}, \ldots, f_{n-1}$. Simultaneously, at each stage n, we update a bound C_f (resp. C_g), such that $f_i \leqslant C_f \tilde{r}_f^{-i}$ for all $i < n$. By the convexity hypothesis, $\tilde{r}_f$ will tend to the convergence radius r_f of f for large n.

When computing the n-th coefficient of fg (say by the naive algorithm for simplicity), we now sum $f_{n-i}g_i$ for i running from 0 to n, where we stop the summation process as soon as

$$\sum_{j \geqslant i} f_{n-j} g_j \leqslant C_f C_g \tilde{r}_f^n \frac{(\tilde{r}_f/\tilde{r}_g)^i}{1 - (\tilde{r}_f/\tilde{r}_g)}$$

is smaller than $2^{-B}(f_n g_0 + \cdots + f_{n-(i-1)} g^{i-1})$. Assuming the convexity hypothesis, the summation process stops for $i = O(1)$, when n tends to infinity. Consequently, we obtain a linear time algorithm. Modulo some care, the trick may be adapted to relaxed multiplication.

6.3. Power series in several variables

6.3.1. Representation of power series in several variables

In principle, multivariate power series can be implemented recursively as univariate power series or univariate power series with multivariate power series coefficients. Unfortunately, this way of doing has two disadvantages. First, the generalized fast Fourier transformation from section 6.1 can not be fully exploited. Secondly, the truncation orders of the coefficients of, say, a bivariate power series are not necessarily constant. This may lead to a phenomenon which is analogous to the "precision loss" phenomenon from section 6.2.1: adding numbers with different orders of growth is equally harmful as adding power series with different truncation orders.

Actually, multivariate power series $f(z_1, \ldots, z_d)$ can be truncated in many ways. For applications in numerical analysis, we are usually interested in evaluating f. Hence, we need the coefficients $f_{n_1, \ldots, n_d}$ with

$$|\alpha_1 n_1 + \cdots + \alpha_d n_d| \leqslant N,$$

where N is proportional to the required precision and $\alpha_1, \ldots, \alpha_d > 0$ depend on the evaluation point and the domain of convergence of f. For applications in combinatorics and the analysis of algorithms, we are often interested in certain specific coefficients of f only. Nevertheless, the computation of such a coefficient $f_{n_1, \ldots, n_d}$ usually amounts to the computation of all "previous" coefficients $f_{k_1, \ldots, k_d}$ with $k_1 \leqslant n_1, \ldots, k_d \leqslant n_d$.

We therefore suggest to implement multivariate power series by an abstract class $\mathsf{Multivariate\_Series(C)}$ whose representation class is given by

CLASS Multivariate_Series_Rep(C)
φ : Multivariate_TPS(C)
I : Multivariate_Dense_Set
virtual compute : Array(Integer) $\to$ C

Here

- Instances of Multivariate_Dense_Set are subsets of $\mathbb{N}^d$ with "a dense flavour": in our case, I will always be the initial segment of already computed coefficients. I.e. if $(n_1, \ldots, n_d) \in I$ and $0 \leqslant m_1 \leqslant n_1, \ldots, 0 \leqslant m_d \leqslant n_d$, then $(m_1, \ldots, m_d) \in I$.
- Instances of Multivariate_TPS(C) are "multivariate truncated power series". The analogue of $\sharp\varphi$ is an instance $\Phi \supseteq I$ of Multivariate_Dense_Set, called the *domain* of φ. Then φ associates a coefficient in C to each element of Φ.
- The (private) method **compute** computes the $(n_1, \ldots, n_d)$-th coefficient of the series, while assuming that all previous coefficients have already been computed. As in the univariate case, the corresponding public method makes sure that all previous coefficients are computed.

REMARK. As we will see in section 6.3.4, it is convenient *not* to assume that instances of Multivariate_Dense_Set are necessarily initial segments.

6.3.2. TRUNCATED MULTIPLICATION

The zealous multivariate truncated multiplication problem can be stated as follows: given two multivariate truncated series f, g : Multivariate_TPS(C) and a dense subset H : Multivariate_Dense_Set of $\mathbb{N}^d$, how to compute the restriction $h = f \times_H g$ of $f \times g$ to H efficiently? That is, how to compute the coefficients $(f \times g)_n$, with $n \in H$?

Let F and G denote the domains of f and g. The naive approach consists of computing h using the formula

$$h = \sum_{(n,m) \in F \times_H G} f_n g_m,$$

where

$$F \times_H G = \{(n,m) \in F \times G | n + m \in H\}.$$

For most domains F, G and H, the time complexity of this computation is bounded by $O(|F \times_H G|)$, where $|F \times_H G|$ denotes the cardinality of $F \times_H G$. The worst case time complexity of a sufficiently clever implementation of the naive algorithm is bounded by

$$O(\min(|F|\,|G|, |F|\,|H|, |G|\,|H|) + |H|).$$

The "fully dense" approach consists of changing f and g into $\overline{f}$ and $\overline{g}$ by inserting zero terms, so that the enlarged domains $\overline{F}$ and $\overline{G}$ of $\overline{f}$ and $\overline{g}$ are *blocks* of the form

$$(a_1 \cdots b_1) \times \cdots \times (a_d \cdots b_d).$$

Here $i \cdots j$ denotes $\{i, \ldots, j-1\}$. Next, we apply an asymptotically fast dense algorithm as described in section 6.1 for the multiplication $\overline{f} \times \overline{g}$ and we truncate the result.

Unfortunately, the fully dense approach is extremely inefficient for certain domains

F, G and H in the multivariate case, because the ratio

$$\rho = \frac{|\overline{F}|\,|\overline{G}|}{|F \times_H G|} \tag{6.8}$$

may become more important than the gain we obtain by using FFT-multiplication. Consider for instance the important special case when

$$F = G = H = \{(n_1, \ldots, n_d) \in \mathbb{N}^d | n_1 + \cdots + n_d \leqslant N\},$$

for some $N > 0$. Let $\rho_{d,N}$ denote the ratio (6.8) for given d and N. For large N, it can be checked that $\rho_{d,N}$ tends to a constant ρ_d given by

$$\begin{aligned} \rho_2 &= 24; \\ \rho_3 &= 1080; \\ \rho_d &\sim \frac{4^d d!^2}{2\sqrt{\pi d}}, \text{ for } d \to \infty. \end{aligned}$$

Hence, even for $d = 2$, we loose a very important factor with respect to the naive algorithm, for small values of N.

6.3.3. A compromise between naive and fully dense multiplication

In this section, we sketch a truncated multivariate multiplication algorithm, which is a compromise between the naive and the fully dense algorithms from the previous section. Our algorithm will never be more than a fixed small constant factor slower than the naive algorithm, but it will fully exploit FFT-multiplication if F, G and H are blocks.

Our algorithm will be recursive on the dimension d. We decompose the truncated series f and g as follows:

$$\begin{aligned} f &= \overline{f} z_d^{q_d} &= (\overline{f}_0 + \cdots + \overline{f}_{\overline{k}-1} z_d^{(\overline{k}-1)p_d}) z_d^{q_d}; \\ g &= \overline{g} z_d^{r_d} &= (\overline{g}_0 + \cdots + \overline{g}_{\overline{l}-1} z_d^{(\overline{l}-1)p_d}) z_d^{r_d}, \end{aligned}$$

where $p_d \geqslant 1, q_d$ and r_d will be chosen heuristically and where $\overline{f}$ and $\overline{g}$ are series in $z_1, \ldots, z_{d-1}, \overline{z_d} = z_d^{q_d}$, whose coefficients are polynomials of degrees $< p_d$ in z_d.

Assuming that we have computed p_d, q_d and r_d, our truncated multiplication algorithm now consists of the following steps:

1. Compute $\overline{f}$ and $\overline{g}$ with domains $\overline{F}$ and $\overline{G}$.
2. Compute the "closure" $\overline{H}$ of H.
3. Compute the truncated product $\overline{h} = \overline{f} \times_{\overline{H}} \overline{g}$ using

$$\overline{h}_{\overline{n}} = \sum_{\overline{\imath}+\overline{\jmath}=\overline{n}} \overline{f}_{\overline{\imath}} \times_{\overline{H}_{\overline{n}}} \overline{g}_{\overline{\jmath}},$$

   where $\overline{H}_{\overline{n}} = \{(n_1, \ldots, n_{d-1}) \in \mathbb{N}^{d-1} | (n_1, \ldots, n_{d-1}, \overline{n}) \in \overline{H}\}$.
4. Recover the product $h = f \times_H g$ from $\overline{h}$.

Step 1 is easy. For instance, the domain of $\overline{f}$ is determined by $(n_1, \ldots, n_{d-1}, \overline{n}_d) \in \overline{F}$, if and only if there exists an n_d with $p_d \overline{n_d} \leqslant n_d - q_d < p_d \overline{n_d} + p_d$ and $(n_1, \ldots, n_d) \in F$. As to $\overline{H}$, we take $(n_1, \ldots, n_{d-1}, \overline{n}_d) \in \overline{H}$ if and only if there exists an n_d with $p_d \overline{n_d} \leqslant$

$n_d - q_d - r_d < p_d\overline{n_d} + 2p_d - 1$ and $(n_1, \ldots, n_d) \in H$. Indeed, the degrees of the coefficients of $\overline{h}$ in z_d are strictly bounded by $2p_d - 1$ and not merely by p_d. In order to recover $h_{n_1,\ldots,n_d}$, we therefore should add up $(\overline{h}_{n_1,\ldots,n_{d-1},\overline{n}_d})_i$ and $(\overline{h}_{n_1,\ldots,n_{d-1},\overline{n}_d-1})_{i+p_d}$, where $\overline{n}_d$ and $0 \leqslant i < p_d$ satisfy $n_d - q_d - r_d = p_d\overline{n}_d + i$.

Let us now show how to compute p_d and q_d. We assume that given p_d, we have an algorithm which rapidly estimates the running time of the multiplication algorithm. The computation of such an estimation will take much time if p_d is small and little time when p_d is large. The idea is now to take compare the estimated running times for decreasing values of p_d and to stop the research of an optimal p_d as soon as smaller values of p_d yield larger estimated running times. More precisely, we start with $p_d = \min(\operatorname{span} F, \operatorname{span} G, \operatorname{span} H)$, where

$$\operatorname{span} S = \max_{(n_1,\ldots,n_d)\in S} n_d - \min_{(n_1,\ldots,n_d)\in S} n_d + 1.$$

Next, we decrease p_d by factors of two ($p_d := \lceil p_d/2 \rceil$). Finally, q_d and r_d are chosen such that $\operatorname{span} \overline{F}, \operatorname{span} \overline{G}$ and $\operatorname{span} \overline{H}$ are as small as possible.

6.3.4. Relaxed multiplication of multivariate power series

Let us now sketch the multivariate analogue of the fast truncated relaxed multiplication algorithm from section 4.4.2. Let f and g denote the series we want to multiply and let h their product. We will first assume that we have fixed upper bounds F, G, H : $\mathsf{Multivariate\_Dense\_Set}$ for the coefficients of f, g and h that we want to compute. These upper bounds coincide with the domains of $f.\varphi, g.\varphi$ and $h.\varphi$.

We observe that, essentially, the fast univariate relaxed algorithm from section 4.3.1 is based on a partition

$$\mathbb{N}^2 = \coprod_{n\in\mathbb{N}} S_n = \coprod_{n\in\mathbb{N}} \coprod_{\alpha\in A_n} S_{n,\alpha}, \tag{6.9}$$

where the $S_{n,\alpha}$ are square blocks of the form $(i \cdots i + l) \times (j \cdots j + l)$:

$$\begin{array}{rcl} S_1 & = & (0\cdots 1)\times(0\cdots 1); \\ S_2 & = & (0\cdots 1)\times(1\cdots 2)\amalg(1\cdots 2)\times(0\cdots 1); \\ S_3 & = & (0\cdots 1)\times(2\cdots 3)\amalg(2\cdots 3)\times(0\cdots 1)\amalg \\ & & (1\cdots 3)\times(1\cdots 3); \\ S_4 & = & (0\cdots 1)\times(3\cdots 4)\amalg(3\cdots 4)\times(0\cdots 1); \\ S_5 & = & (0\cdots 1)\times(4\cdots 5)\amalg(4\cdots 5)\times(0\cdots 1)\amalg \\ & & (1\cdots 3)\times(3\cdots 5)\amalg(3\cdots 5)\times(1\cdots 3); \\ & \vdots & \end{array}$$

Now at the n-th stage, the algorithm consists of computing the contribution

$$\sum_{\alpha\in A_n} \sum_{(i,j)\in S_{n,\alpha}} f_i g_j z^{i+j}$$

of all blocks $S_{n,\alpha}$ to fg.

In the multivariate case, we do a similar thing: we partition $(\mathbb{N}^d)^2$ by

$$\begin{aligned}(\mathbb{N}^d)^2 &= \coprod_{(n_1,\ldots,n_d)\in\mathbb{N}^d} S_{(n_1,\ldots,n_d)}\\ &= \coprod_{(n_1,\ldots,n_d)\in\mathbb{N}^d} \coprod_{(\alpha_1,\ldots,\alpha_d)\in A_{(n_1,\ldots,n_d)}} S_{(n_1,\ldots,n_d),(\alpha_1,\ldots,\alpha_d)}\\ &= \coprod_{(n_1,\ldots,n_d)\in\mathbb{N}^d} \coprod_{(\alpha_1,\ldots,\alpha_d)\in A_{n_1}\times\cdots\times A_{n_d}} \varphi(S_{n_1,\alpha_1}\times\cdots\times S_{n_d,\alpha_d}),\end{aligned}$$

where φ is the natural isomorphism from $(\mathbb{N}^2)^d$ onto $(\mathbb{N}^d)^2$. Then each $S_{(n_1,\ldots,n_d),(\alpha_1,\ldots,\alpha_d)}$ is the product of two d-dimensional blocks

$$S_{(n_1,\ldots,n_d),(\alpha_1,\ldots,\alpha_d)} = B_{(n_1,\ldots,n_d),(\alpha_1,\ldots,\alpha_d)} \times C_{(n_1,\ldots,n_d),(\alpha_1,\ldots,\alpha_d)}.$$

When we ask for the $(n_1,\ldots,n_d)$-th coefficient of h, the multivariate truncated relaxed multiplication algorithm now computes all truncated products

$$(B_{(n_1,\ldots,n_d)\cap F),(\alpha_1,\ldots,\alpha_d)} \times_H (C_{(n_1,\ldots,n_d)\cap G),(\alpha_1,\ldots,\alpha_d)}$$

by the zealous algorithm from the previous section and adds these contributions to $h.\varphi$.

Until now, we assumed that F, G and H remained fixed throughout the execution, as is often the case in practice. Sometimes however, these domains have to be increased dynamically, say into $\hat{F}$, $\hat{G}$ and $\hat{H}$. Whenever this happens, it suffices to add all products

$$\begin{aligned}(B_{(n_1,\ldots,n_d),(\alpha_1,\ldots,\alpha_d)}\cap(\hat{F}\backslash F)) &\times_H & (C_{(n_1,\ldots,n_d),(\alpha_1,\ldots,\alpha_d)}\cap G),\\ (B_{(n_1,\ldots,n_d),(\alpha_1,\ldots,\alpha_d)}\cap\hat{F}) &\times_H & (C_{(n_1,\ldots,n_d),(\alpha_1,\ldots,\alpha_d)}\cap(\hat{G}\backslash G)) \text{ and}\\ (B_{(n_1,\ldots,n_d),(\alpha_1,\ldots,\alpha_d)}\cap\hat{F}) &\times_{\hat{H}\backslash H} & (C_{(n_1,\ldots,n_d),(\alpha_1,\ldots,\alpha_d)}\cap\hat{G})\end{aligned}$$

to $h.\varphi$, where $(n_1,\ldots,n_d)$ runs over all already computed coefficients in $h.I$. Notice that $\hat{F}\backslash F$, $\hat{G}\backslash G$ and $\hat{H}\backslash H$ are not necessarily initial segments; this explains why it is convenient to allow the instances of Multivariate_Dense_Set to be general subsets of $\mathbb{N}^d$.

7. Conclusion

In this paper, we have shown that all classical fast zealous algorithms for manipulating formal power series admit relaxed analogues of the same asymptotic complexity up to a factor $O(\log n)$. Theoretically speaking, this allows us to expand power series solutions to (partial) differential equations with almost linear time complexities and solutions to differential-composition equations with an almost sesquilinear complexity.

We have also pointed out that it is hard to conceive implementations in actual computer algebra systems, which adequately reflect these asymptotic time complexities in actual computer algebra systems. This is mainly due to the absense of fast arithmetic in such systems, such as DAC- and FFT-multiplication. An interesting, but perverse consequence of the lack of such arithmetic, is that compairisons between certain algorithms on the hand of benchmarks may be misleading (e.g. see our remarks about table 5).

Another difficulty for actual implementations is that there seems not to be a best overall relaxed multiplication algorithm (see section 4.4). Nevertheless, for applications where the expansion order is known in advance, i.e. when computations need not be resumed, the fast truncated relaxed algorithm (see sections 4.4.2 and 6.3.4) often turns out to be the fastest. In general, we expect that the best performance is obtained by a hybrid algorithm, which selects between different expansion methods as a function of the origin of the series (general, algebraic, holonomic, etc.), the constant field, the expansion order and the possibility to resume computations. Of course, such a hybrid algorithm is also the longest one to implement.

Despite the above drawbacks of the relaxed approach, our benchmarks show that for large expansion orders, we systematically gain with respect to the lazy approach. In certain cases (see tables 8, 9 and 10) these gains become very important and may exceed a factor of 100. In the future, these factors are expected to increase more and more, since processor speed and memory capacity tend to increase proportionally and powerful implementations of the FFT-transform might eventually show up. We also notice that the relaxed algorithms tend to be faster than Brent and Kung's algorithm for exponentiation and the resolution of differential equations (see table 4).

Having summarized the advantages and disadvantages of the relaxed approach, we will conclude this section by a discussion of its fitness for different types of applications, with some suggestions for those who want to implement a power series library into a computer algebra system, and some final general remarks.

7.1. Applications

Symbolic computation.

The pertinence of the relaxed approach for general applications in symbolic computation depends strongly on the problem. On the one hand, multiplication of large symbolic expressions will tend to be slow (which favorites the relaxed approach). On the other hand, often only little terms are required (which favorites the lazy approach).

Combinatorics and the analysis of algorithms.

In combinatorics, the analysis of algorithms and for the random generation of combinatorial objects, one usually needs to expand generating functions up to a high order. Therefore, this is the ideal application for relaxed power series (Flajolet *et al.*, 1990). In this context, multivariate power series correspond to the study of parameters in enumeration problems or the analysis of a certain algorithm (Soria, 1990).

Numerical analysis.

In numerical analysis, power series are mainly computed in order to be evaluated. The required number of terms usually depends linearly on the required precision of the evaluation. Usually, in absence of numerical instability only a few terms suffice and constant multiplications will be very efficient. Therefore, only small spee-ups can possibly achieved using the relaxed approach, at the price of massive inlining.

On the other hand, near singularities, analytic continuation algorithms may become numerically instable and higher precisions and expansion orders might be required. The numerical resolution of partial differential equations might be another future application of the relaxed approach.

7.2. Suggestions for implementors

The choice which algorithms to implement in a power series package should mainly depend on the applications one has in mind and the time one is willing to spend. Roughly speaking, we would like to distinguish three choices:

A simple quickly implemented package.

If you have little time and not interested in applications where high expansions orders are needed (such as combinatorics and the analysis of algorithms), you are probably best off with a quickly implemented lazy power series package.

Boosting your simple package.

If you have some more time and you want to boost the performance of a lazy power series package for large expansion orders, then you may replace your multiplication procedure with the algorithm from section 4.3.1. You may also implement one or more relaxed composition algorithm from section 5 and a holonomic function package. On the other hand, it seems not necessary to implement the fast zealous algorithms from section 3, since the relaxed algorithms are almost as fast and offer the possibility of solving virtually all functional equations.

Developing an optimal package.

If you really want optimal speed and/or genericity, then we suggest first to implement a package for really fast dense arithmetic based on the FFT-transform (as described in section 6.1). Next, we suggest you to carefully implement hybrid relaxed truncated multiplication and composition algorithms, which are both efficient for small sizes (due to massive inlining) and larger sizes (due to the asymptotically fast zealous arithmetic).

7.3. Final remarks

Other infinite structures.

In principle, the relaxed approach may be applied to other valuation rings with fast zealous arithmetic, such as the p-adic numbers (Bernardin, 1998).

Generalized series and transseries.

The lazy approach also applies in the case of power series with generalized exponents (Salvy, 1991). In general, the relaxed approach does not lead to faster algorithms, because there no fast arithmetic for sparse polynomials. Nevertheless, if the exponents are *grid-based* (i.e. they belong to a set of the form $a + b_1\mathbb{N} + \cdots + b_n\mathbb{N}$, where $a \in \mathbb{R}$ and $b_1, \ldots, b_n \in \mathbb{R}_*^+$), then we are essentially handling power series in several variables, so we can gain on the complexity. For applications, see (Richardson *et al.*, 1996; van der Hoeven, 1997a).

Computing specific terms.

For certain very particular power series, it is possible to compute given coefficients without computing the previous ones, usually by using Lagrange's inversion formula (Brent and Kung, 1978).

MODULAR ARITHMETIC.

In computer algebra, modular arithmetic is often used to speed up computations with integers. For our application, modular algorithms may be interesting for parallelization purposes and in order to reduce the memory requirements if we are merely interested in a particular coefficient of the series. Notice that modular arithmetic enters in the general scheme for fast arithmetic as described in section 6.1.

PARALLELISM.

Except for the zealous algorithms from section 3, lazy and relaxed algorithms have the disadvantage of being essentially sequential. Nevertheless, the fast relaxed multiplication algorithm is closer to being parallel, since the zealous multiplications might be done in parallel modulo a proper synchronization. We also notice that it is often possible to parallelize the ring operations for C.

MIXING ZEALOUS AND RELAXED MULTIPLICATION.

Consider the multiplication $h = f \times g$ of two relaxed power series. Sometimes, the arguments f and g do not depend on h. In this case, a zealous algorithm may be used for the multiplication. It can also happen that f depends on h, but not g. In this case, it is possible to improve the constant factor for the relaxed multiplication by choosing a more appropriate partition of $\mathbb{N}^2$ instead of (6.9).

OTHER OPERATIONS ON FORMAL POWER SERIES.

Some other operation on formal power series may be considered, such as

$$f(z) \mapsto \sum_{k \geqslant 1} \frac{\phi(k)}{k} \log \frac{1}{1 - f(z^k)},$$

which corresponds to taking cycles of combinatorial structures (Flajolet and Soria, 1991). Clearly, the relaxed approach still applies to the above and other, similar operations.

ACKNOWLEDGMENT. The author expresses his thanks to D. Saunders for suggesting the name "laid-back power series" and I apologize for my zealotry in searching an equivalent name.

References

Bernardin, L. (1998). On bivariate Hensel lifting and its parallelization. In Gloor, O., editor, *Proc. ISSAC '98*, pages 96–100, Rostock, Germany.

Bernstein, D. (1998). Composing power series over a finite ring in essentially linear time. *J.S.C.*, **26**(3):339–341.

Brent, R., Kung, H. (1975). $O((n \log n)^{3/2})$ algorithms for composition and reversion of power series. In Traub, J., editor, *Analytic Computational Complexity*. Proc. of a symposium on analytic computational complexity held by Carnegie-Mellon University.

Brent, R., Kung, H. (1977). Fast algorithms for composition and reversion of multivariate power series. In *Proc. Conf. Th. Comp. Sc.*, pages 149–158, Waterloo, Ontario, Canada. University of Waterloo.

Brent, R., Kung, H. (1978). Fast algorithms for manipulating formal power series. *Journal of the ACM*, **25**:581–595.

Cantor, D., Kaltofen, E. (1991). On fast multiplication of polynomials over arbitrary algebras. *Acta Informatica*, **28**:693–701.

Chudnovsky, D., Chudnovsky, G. (1990). Computer algebra in the service of mathematical physics and number theory (computers in mathematics, stanford, ca, 1986). In *Lect. Notes in Pure and Applied Math.*, volume 125, pages 109–232, New-York. Dekker.

Cook, S. (1966). *On the minimum computation time of functions.* PhD thesis, Harvard University.

Cooley, J., Tukey, J. (1965). An algorithm for the machine calculation of complex fourier series. *Math. Computat.*, **19**:297–301.

Denise, A., Dutour, I., Zimmermann, P. (1998). Cs: a MuPAD package for counting and randomly generating combinatorial structures. In *Proceedings of FPSAC'98*, pages 195–204. Sofware Demonstration.

Denise, A., Zimmermann, P. (1999). Uniform random generation of decomposable structures using floating-point arithmetic. *Theoretical Computer Science*, **218**(2):219–232.

Flajolet, P., Salvy, B., Zimmermann, P. (1990). Automatic average-case analysis of algorithms. *T.C.S.*, **79**(1):37–109.

Flajolet, P., Sedgewick, R. (1996). *An introduction to the analysis of algorithms.* Addison Wesley, Reading, Massachusetts.

Flajolet, P., Soria, M. (1991). The cycle construction. *SIAM Journal on Discrete Mathematics*, **4**:48–60.

Flajolet, P., Zimmerman, P., Van Cutsem, B. (1994). A calculus for the random generation of labelled combinatorial structures. *Theoretical Computer Science*, **132**(1-2):1–35.

Heideman, M., Johnson, D., Burrus, C. (1984). Gauss and the history of the fft. *IEEE Acoustics, Speech and Signal Processing Magazine*, **1**:14–21.

Karatsuba, A., Ofman, J. (1962). Умножение многозначных чисел на автоматах. *Doklady Akad. Nauk SSSR*, **7**:293–294. English translation in (Karatsuba and Ofman, 1963).

Karatsuba, A., Ofman, J. (1963). Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, **7**:595–596.

Knuth, D. (1981). *The art of computer programming*, volume 2: Seminumerical algorithms. Addison-Wesley, 2-nd edition.

Kung, H., Traub, J. (1978). All algebraic functions can be computed fast. *Journal of the ACM*, **25**(2):245–260.

Lipshitz, L. (1989). D-finite power series. *Journal of Algebra*, **122**:353–373.

Norman, A. (1975). Computing with formal power series. *ACM Trans. of Math. Software*, **1**(4):346–356.

Norman, A., Fitch, J. (1997). Cabal: Polynomial and power series algebra on a parallel computer. In Hitz, M., Kaltofen, E., editors, *Proc. PASCO '97*, pages 196–203, Maui, Hawaii.

Nussbaumer, H. (1981). *Fast Fourier Transforms and Convolution Algorithms.* Springer-Verlag.

Odlyzko, A. (1982). Periodic oscillations of coefficients of power series that satisfy functional equations. *Adv. in Math.*, **44**:180–205.

Pólya, G. (1937). Kombinatorische Anzahlbestimmungen für Gruppen, Graphen und chemische Verbindungen. *Acta Mathematica*, **68**:145–254.

Richardson, D., Salvy, B., Shackell, J., van der Hoeven, J. (1996). Expansions of exp-log functions. In Lakhsman, Y., editor, *Proc. ISSAC '96*, pages 309–313, Zürich, Switzerland.

Salvy, B. (1991). *Asymptotique automatique et fonctions génératrices.* PhD thesis, École Polytechnique, France.

Salvy, B., Zimmermann, P. (1994). Gfun: a Maple package for the manipulation of generating and holonomic functions in one variable. *ACM Trans. on Math. Software*, **20**(2):163–177.

Schönhage, A., Strassen, V. (1971). Schnelle Multiplikation grosser Zahlen. *Computing 7*, **7**:281–292.

Sieveking, M. (1972). An algorithm for division of power series. *Computing*, **10**:153–156.

Soria, M. (1990). *Méthodes d'analyse pour les constructions combinatoires et les algorithmes.* PhD thesis, Univ. Paris-Sud, Orsay, France.

Stanley, R. (1980). Differentially finite power series. *European J. Combin.*, **1**:175–188. MR # 81m:05012.

Stroustrup, B. (1995). *The C++ programming language.* Addison-Wesley, 2-nd edition.

Toom, A. (1963a). The complexity of a scheme of functional elements realizing the multiplication of integers. *Soviet Mathematics*, **4**(2):714–716.

Toom, A. (1963b). О сложности схемы из функциональных элементов, реализирующей умножение целых чисел. *Doklady Akad. Nauk SSSR*, **150**:496–498. English translation in (Toom, 1963a).

van der Hoeven, J. (1997a). *Automatic asymptotics.* PhD thesis, École polytechnique, France.

van der Hoeven, J. (1997b). Lazy multiplication of formal power series. In Küchlin, W. W., editor, *Proc. ISSAC '97*, pages 17–20, Maui, Hawaii.

von Zurgathen, J., Gerhard, J. (1999). *Modern Computer Algebra.* Cambridge University Press.

Zeilberger, D. (1990). A holonomic systems approach to special functions identities. *Journal of Comp. and Appl. Math.*, **32**:321–368.