The Truncated Fourier Transform and Applications

Joris van der Hoeven
Département de Mathématiques (bât. 425)
Université Paris-Sud
91405 Orsay Cedex
France
joris@texmacs.org

ABSTRACT

In this paper, we present a truncated version of the classical Fast Fourier Transform. When applied to polynomial multiplication, this algorithm has the nice property of eliminating the "jumps" in the complexity at powers of two. When applied to the multiplication of multivariate polynomials or truncated multivariate power series, we gain a logarithmic factor with respect to the best previously known algorithms.

Categories and Subject Descriptors

F.2.1 [**Theory of Computation**]: Analysis of algorithms and problem complexity—*Numerical algorithms and problems*

General Terms

Algorithms

Keywords

Fast Fourier Transform, jump phenomenon, truncated multiplication, FFT-multiplication, multivariate polynomials, multivariate power series.

1. INTRODUCTION

Let $\mathcal{R} \ni 1/2$ be an effective ring of constants (i.e. the usual arithmetic operations $+$, $-$ and $\times$ can be carried out by algorithm). If $\mathcal{R}$ has a primitive n-th root of unity with $n = 2^p$, then the product of two polynomials $P, Q \in \mathcal{R}[X]$ with $\deg PQ < n$ can be computed in time $O(n \log n)$ using the Fast Fourier Transform or FFT [4]. If $\mathcal{R}$ does not admit a primitive n-th root of unity, then one needs an additional overhead of $O(\log \log n)$ in order to carry out the multiplication, by artificially adding new root of unity [11, 3].

Besides the fact that the asymptotic complexity of the FFT involves a large constant factor, another classical drawback is that the complexity function admits important jumps at each power of two. These jumps can be reduced by using $(k2^p)$-th roots of unity for small k. They can also be smoothened by decomposing $(n+\delta) \times (n+\delta)$-multiplications as $n \times n$-, $n \times \delta$- and $(n+\delta) \times \delta$-multiplications. However, these tricks are not very elegant, cumbersome to implement, and they do not allow to completely eliminate the jump problem.

In section 3, we present a new kind of "Truncated Fourier Transform" or TFT, which allows for the fast evaluation of a polynomial $P \in \mathcal{R}[X]$ in any number n of well-chosen roots of unity. This algorithm coincides with the usual FFT if n is a power of two, but it behaves smoothly for intermediate values. In section 4, we also show that the inverse operation of interpolation can be carried out with the same complexity (modulo a few additional shifts).

The TFT permits to speed up the multiplication of univariate polynomials with a constant factor between 1 and 2. In the case of multivariate polynomials, the repeated gain of such a constant factor leads to the gain of a non-trivial asymptotic factor. More precisely, assuming that $\mathcal{R}$ admits sufficiently 2^p-th roots of unity, we will show in section 5 that the product of two multivariate polynomials $P, Q \in \mathcal{R}[z_1, \ldots, z_d]$ can be computed in time $O(s \log s)$, where $s = \binom{r+d-1}{d}$ and $r = \deg PQ + 1$. The best previously known algorithm [2], based on sparse polynomial multiplication, has time complexity $O(s \log^2 s)$.

In section 6 we finally give an algorithm for the multiplication of truncated multivariate power series. This algorithm, which has time complexity $O(s \log^2 s)$, again improves the best previously known algorithm [8] by a factor of $O(\log s)$. Moreover, both in the cases of multivariate polynomials and power series, we expect the corresponding constant factor to be better.

2. THE FAST FOURIER TRANSFORM

Let $\mathcal{R}$ be an effective ring of constants, $n = 2^p$ with $p \in \mathbb{N}$ and $\omega \in \mathcal{R}$ a primitive n-th root of unity (i.e. $\omega^{n/2} = -1$). The discrete Fast Fourier Transform (FFT) of an n-tuple $(a_0, \ldots, a_{n-1}) \in \mathcal{R}^n$ (with respect to ω) is the n-tuple $(\hat{a}_0, \ldots, \hat{a}_{n-1}) = \mathrm{FFT}_\omega(a) \in \mathcal{R}^n$ with

$$\hat{a}_i = \sum_{j=0}^{n-1} a_j \omega^{ij}.$$

In other words, $\hat{a}_i = A(\omega^i)$, where $A \in \mathcal{R}[X]$ denotes the polynomial $A = a_0 + a_1 X + \cdots + a_{n-1} X^{n-1}$.

The F.F.T can be computed efficiently using binary splitting: writing

$$(a_0, \ldots, a_{n-1}) = (b_0, c_0, \ldots, b_{n/2-1}, c_{n/2-1}),$$

we recursively compute the Fourier transforms of the sequences $(b_0, \ldots, b_{n/2-1})$ and $(c_0, \ldots, c_{n/2-1})$ by

$$\begin{aligned} \mathrm{FFT}_{\omega^2}(b_0, \ldots, b_{n/2-1}) &= (\hat{b}_0, \ldots, \hat{b}_{n/2-1})\,; \\ \mathrm{FFT}_{\omega^2}(c_0, \ldots, c_{n/2-1}) &= (\hat{c}_0, \ldots, \hat{c}_{n/2-1}). \end{aligned}$$

Then we have

$$\begin{aligned} \mathrm{FFT}_{\omega}(a_0, \ldots, a_{n-1}) = {} & (\hat{b}_0 + \hat{c}_0, \ldots, \hat{b}_{n/2-1} + \hat{c}_{n/2-1}\omega^{n/2-1} \\ & \hat{b}_0 - \hat{c}_0, \ldots, \hat{b}_{n/2-1} - \hat{c}_{n/2-1}\omega^{n/2-1}). \end{aligned}$$

This algorithm requires $np = n \log_2 n$ multiplications with powers of ω and $2np$ additions (or subtractions).

In practice, it is most efficient to implement an in-place variant of the above algorithm. We will denote by $[i]_p$ the bitwise mirror of i at length p (for instance, $[3]_5 = 24$ and $[11]_5 = 26$). At step 0, we start with the vector

$$x_0 = (x_{0,0}, \ldots, x_{0,n-1}) = (a_0, \ldots, a_{n-1}).$$

At step $s \in \{1, \ldots, p\}$, we set

$$\begin{pmatrix} x_{s,im_s+j} \\ x_{s,(i+1)m_s+j} \end{pmatrix} = \begin{pmatrix} 1 & \omega^{[i]_s m_s} \\ 1 & -\omega^{[i]_s m_s} \end{pmatrix} \begin{pmatrix} x_{s-1,im_s+j} \\ x_{s-1,(i+1)m_s+j} \end{pmatrix}. \quad (1)$$

for all $i \in \{0, 2, \ldots, n/m_s - 2\}$ and $j \in \{0, \ldots, m_s - 1\}$, where $m_s = 2^{p-s}$. Using induction over s, it can easily be seen that

$$x_{s,im_s+j} = (\mathrm{FFT}_{\omega^{m_s}}(a_j, a_{m_s+j}, \ldots, a_{n-m_s+j}))_{[i]_s},$$

for all $i \in \{0, \ldots, n/m_s - 1\}$ and $j \in \{0, \ldots, m_s - 1\}$. In particular,

$$\begin{aligned} x_{p,i} &= \hat{a}_{[i]_p} \\ \hat{a}_i &= x_{p,[i]_p} \end{aligned}$$

for all $i \in \{0, \ldots, n-1\}$. This algorithm of "repeated crossings" is illustrated in figure 1.

A classical application of the FFT is the multiplication of polynomials $A = a_0 + \cdots + a_{n-1}X^{n-1}$ and $B = b_0 + \cdots + b_{n-1}X^{n-1}$. Assuming that $\deg AB < n$, we first evaluate A and B in $1, \omega, \ldots, \omega^{n-1}$ using the FFT:

$$\begin{aligned} (A(1), \ldots, A(\omega^{n-1})) &= \mathrm{FFT}_{\omega}(a_0, \ldots, a_{n-1}) \\ (B(1), \ldots, B(\omega^{n-1})) &= \mathrm{FFT}_{\omega}(b_0, \ldots, b_{n-1}) \end{aligned}$$

We next compute the evaluations

$$(A(1)B(1), \ldots, A(\omega^{n-1})B(\omega^{n-1})))$$

of AB at $1, \ldots, \omega^{n-1}$. We finally have to recover AB from these values using the inverse FFT. But the inverse FFT with respect to ω is nothing else as $1/n$ times the direct FFT with respect to ω^{-1}. Indeed, for all $(a_0, \ldots, a_{n-1}) \in \mathcal{R}^n$ and all $i \in \{0, \ldots, n-1\}$, we have

$$\mathrm{FFT}_{\omega^{-1}}(\mathrm{FFT}_{\omega}(a))_i = \sum_{k=0}^{n-1} \sum_{j=0}^{n-1} a_i \omega^{(i-k)j} = n a_i, \qquad (2)$$

since

$$\sum_{j=0}^{n-1} \omega^{(i-k)j} = 0$$

whenever $i \neq k$. This yields a multiplication algorithm of time complexity $O(n \log n)$ in $\mathcal{R}[X]$, when assuming that $\mathcal{R}$ admits enough primitive 2^p-th roots of unity. In the case that $\mathcal{R}$ does not, then new roots of unity can be added artificially [11, 3, 13] so as to yield an algorithm of time complexity $O(n \log n \log \log n)$.

3. THE TRUNCATED FOURIER TRANSFORM

The algorithm from the previous section has the disadvantage that n needs to be a power of two. If we want to multiply two polynomials $A, B \in \mathcal{R}[X]$ such that $\deg AB + 1 = n + \delta$, then we need to carry out the FFT at precision $2n$, thereby losing a factor of 2. This factor can be reduced using several tricks. For instance, one may decompose the $(n+\delta) \times (n+\delta)$-product into an $n \times n$ product, an $n \times \delta$-product and an $(n+\delta) \times \delta$-product. This is efficient for small δ, but not very good if $\delta \approx n/2$. In the latter case, one may also use an FFT at precision $3n/2$, by using 3×3-matrices at one step of the FFT computation. However, all these tricks of the trade require a large amount of hacking and one always continues to lose a non-trivial factor between 1 and 2.

The idea behind the Truncated Fourier Transform is to provide an efficient algorithm for the evaluation of polynomials in any number of distinct points. Moreover, the inverse operation of interpolation can be carried out with the same complexity (modulo a few additional shifts). This technique will eliminate the "jumps" in the complexity of FFT multiplication.

So let $n = 2^p$, $l \leqslant n$ (usually, $l > n/2$) and let ω be a primitive n-th root of unity. Given an l-tuple $(a_0, \ldots, a_{l-1})$, we will evaluate the corresponding polynomial $A = a_0 + \cdots + a_{l-1}X^{l-1}$ in $\omega^{[0]_p}, \omega^{[1]_p}, \ldots, \omega^{[l-1]_p}$. We call

$$(A(\omega^{[0]_p}), \ldots, A(\omega^{[l-1]_p}))$$

the *Truncated Fourier Transform* (TFT) of $(a_0, \ldots, a_{l-1})$. Now consider the completion of the l-tuple $(a_0, \ldots, a_{l-1})$ into an n-tuple $(a_0, \ldots, a_{l-1}, 0, \ldots, 0)$. When using the in-place algorithm from the previous section in order to compute $(A(\omega^{[0]_p}), \ldots, A(\omega^{[l-1]_p}))$, we claim that many of the computations of the $x_{s,i}$ can actually be skipped (see figure 2). Indeed, at stage s, it suffices to compute the vector $(x_{s,0}, \ldots, x_{s,(\lfloor (l-1)/m_s \rfloor + 1)m_s - 1})$. Besides $x_{s,0}, \ldots, x_{s,l-1}$, we therefore compute at most $m_s = 2^{p-s}$ additional values. In total, we therefore compute at most $pl + 2^{p-1} + 2^{p-2} + \cdots + 1 < pl + n$ values $x_{s,i}$. This proves the following result:

THEOREM 1. *Let $n = 2^p$, $l \leqslant n$ and let $\omega \in \mathcal{R}$ be a primitive n-th root of unity in $\mathcal{R}$. Then the Truncated Fourier Transform of an l-tuple $(a_0, \ldots, a_{l-1})$ w.r.t. ω can be computed using at most $lp + n$ additions (or subtractions) and $\lceil (lp + n)/2 \rceil$ multiplications with powers of ω.*

REMARK 1. Assume that $\mathcal{R}$ admits a privileged primitive n-th root of unity ω_n for every $n \in 2^{\mathbb{N}}$, such that $\omega_{2n}^2 = \omega_n$ for all n. Then the TFT $(\hat{a}_0, \ldots, \hat{a}_{l-1})$ of an l-tuple $(a_0, \ldots, a_{l-1})$ w.r.t. ω_n with $n \geqslant l$ does not depend on the choice of n. We call $(\hat{a}_0, \ldots, \hat{a}_{l-1})$ the TFT of $(a_0, \ldots, a_{l-1})$ w.r.t. the privileged sequence $(\omega_1, \omega_2, \omega_4, \ldots)$ of roots of unity.

REMARK 2. Since the only operations we need for computing the TFT are additions, subtractions and multiplications by powers of ω, the algorithm naturally combines

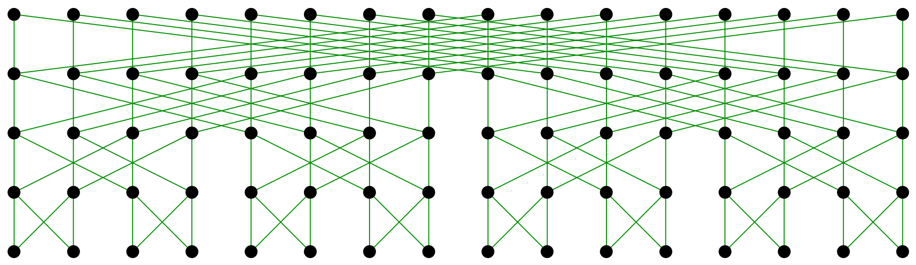

Figure 1: Schematic representation of a Fast Fourier Transform for $n = 16$. The black dots correspond to the $x_{s,i}$, the upper row being $(x_{0,0}, \ldots, x_{0,15}) = (a_0, \ldots, a_{15})$ and the lower row $(x_{4,0}, \ldots, x_{4,15}) = (\hat{a}_0, \hat{a}_8, \hat{a}_4, \hat{a}_{12}, \ldots, \hat{a}_{15})$.

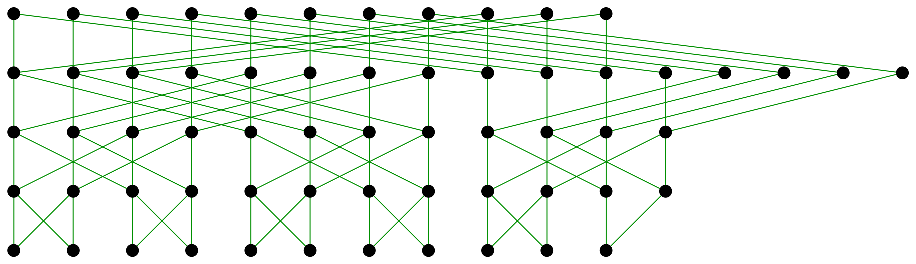

Figure 2: Schematic representation of a TFT for $n = 16$ and $l = 11$.

with Schönhage-Strassen's algorithm when ω is a symbolically added root of unity.

REMARK 3. If $f_0 = \cdots = f_{l'-1}$, then the Truncated Fourier Transform of $(f_0, \ldots, f_{l-1})$ can be computed using $O((l-l')p+2n)$ ring operations using a similar algorithm as above. More generally, this allows the rapid transformation of "unions of segments".

4. INVERTING THE TRUNCATED FOURIER TRANSFORM

Unfortunately, the inverse TFT cannot be computed using a similar formula as (2). Indeed, starting with the $x_{l,i}$, we need to compute an increasing number of $x_{s,i}$ when s decreases. Therefore we will rather invert the algorithm which computes the TFT, but with this difference that we will sometimes need $x_{s',i'}$ with $s' < s$ in order to compute $x_{s,i}$. We will use the fact that whenever one value among

$$x_{s,im_s+j}, x_{s-1,im_s+j}$$

and one value among

$$x_{s,(i+1)m_s+j}, x_{s-1,(i+1)m_s+j}$$

are known in the cross relation (1), then we can deduce the others from them using one multiplication by a power of ω and two "shifted" additions or subtractions (i.e. the results may have to be divided by 2).

More precisely, let us denote $k_s = \lfloor (l-1)/m_s \rfloor m_s$ and $l_s = k_s + m_s$ at each stage s. We use a recursive algorithm which takes the values $x_{p,k_s}, \ldots, x_{p,l-1}$ and $x_{s,l}, \ldots, x_{s,l_s}$ on input, and which computes $x_{s,k_s}, \ldots, x_{s,l-1}$. If $s = p$, then we have nothing to do. Otherwise, we distinguish two cases:

- If $l_s = l_{s+1}$, then we first compute $x_{s,k_s}, \ldots, x_{s,k_{s+1}-1}$ from $x_{p,k_s}, \ldots, x_{p,k_{s+1}-1}$ using repeated crossings. We next deduce $x_{s,i}$ and $x_{s+1,i+m_s/2}$ from $x_{s+1,i}$ and $x_{s,i+m_s/2}$ for all $i \in \{l-m_s/2, \ldots, k_{s+1}-1\}$. Invoking our algorithm recursively, we now obtain $x_{s+1,k_{s+1}}, \ldots, x_{s+1,l-1}$. We finally compute $x_{s,i}$ and $x_{s,i+m_s/2}$ from $x_{s+1,i}$ and $x_{s+1,i+m_s/2}$ for $i \in \{k_s, \ldots, l-m_s/2-1\}$.

- If $l_s > l_{s+1}$, then we first compute $x_{s+1,i}$ from $x_{s,i}$ and $x_{s,i+m_s/2}$ for $i \in \{l, \ldots, l_{s+1}-1\}$. Invoking our algorithm recursively, we next compute $x_{s+1,k_{s+1}}, \ldots, x_{s+1,l-1}$. For each $i \in \{k_s, \ldots, l-1\}$, we finally deduce $x_{s,i}$ from $x_{s+1,i}$ and $x_{s,i+m_s/2}$.

The two cases are illustrated in figures 3 resp. 4. Since $x_{0,l} = \cdots = x_{0,n-1} = 0$, the application of our algorithm for $s = 0$ computes the inverse TFT. We notice that the values $x_{s,i}$ with $i < l$ are computed in decreasing order (for s) and the values $x_{s,i}$ with $i \geqslant l$ in increasing order. In other words, the algorithm may be designed in such a way to remain in place. We have proved:

THEOREM 2. *Let $n = 2^p$, $l \leqslant n$ and let $\omega \in \mathcal{R}$ be a primitive n-th root of unity in $\mathcal{R}$. Then the l-tuple $(a_0, \ldots, a_{l-1})$ can be recovered from its Truncated Fourier Transform w.r.t. ω using at most $lp+n$ shifted additions (or subtractions) and $\lceil (lp+n)/2 \rceil$ multiplications with powers of ω.*

REMARK 4. Besides $O(n)$ shifted additions, subtractions or multiplications by powers of ω, the algorithm essentially computes inverse FFT-transforms of sizes $2^{q_1}, \ldots, 2^{q_r}$ with $n = 2^{q_1} + \cdots + 2^{q_r}$. Using (2), it is therefore possible to replace all but $O(n)$ shifted additions and subtractions by normal additions and subtractions.

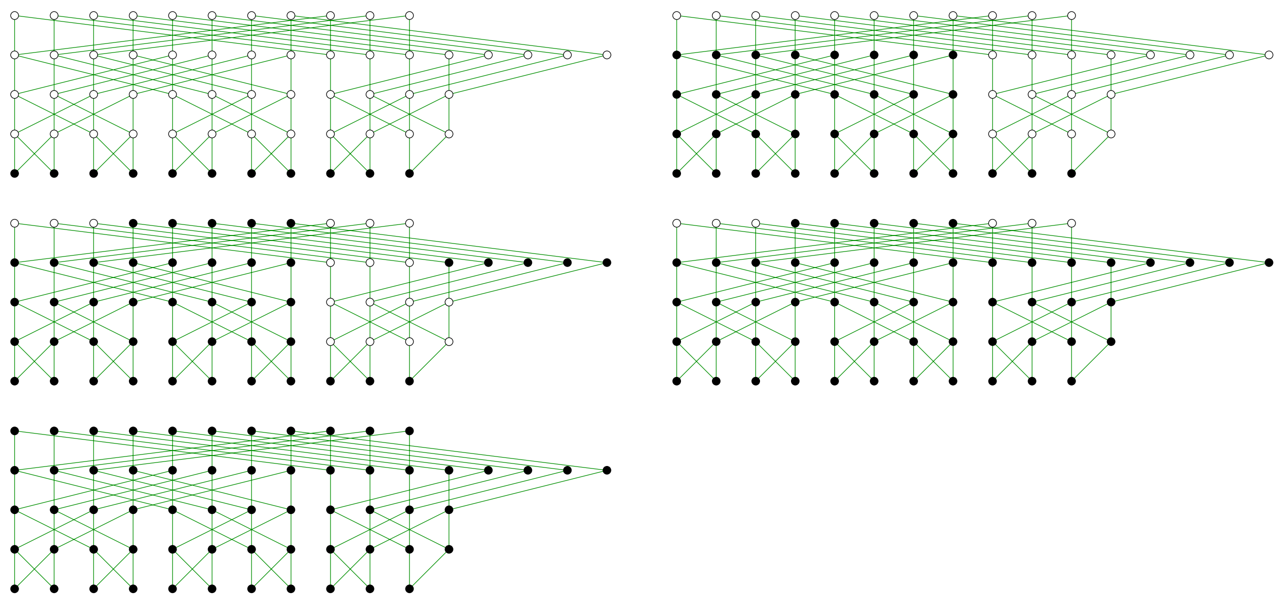

Figure 3: Schematic representation of the recursive computation of the inverse TFT for $n = 16$, $l = 11$. The different images show the progression of the known values $x_{i,j}$ (the black dots) during the different computations at stage $s = 0$. Since $l_0 = l_1 = 16$, we fall into the first case of our algorithm and the recursive invocation of the algorithm is done between the third and the fourth image.

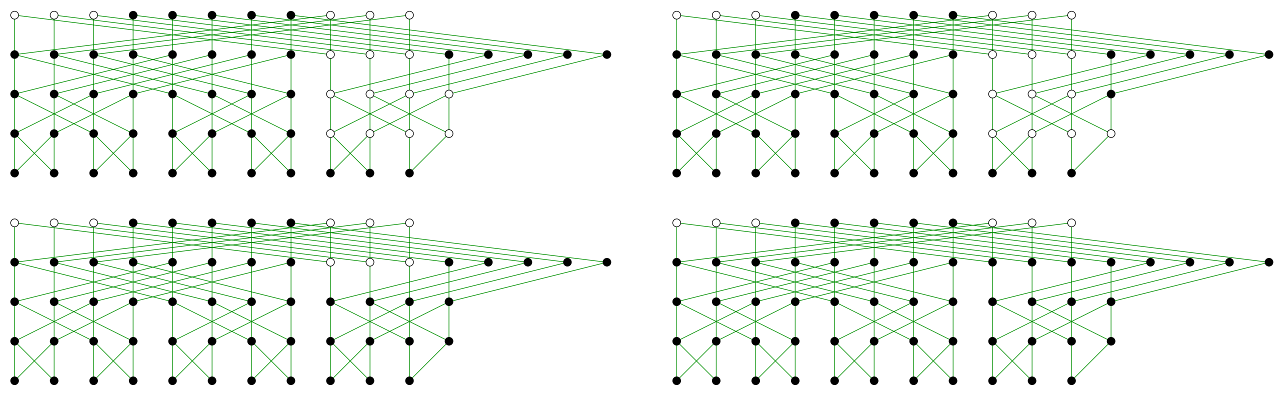

Figure 4: Schematic representation of the recursive computation of the inverse TFT for $n = 16$, $l = 11$ at stage $s = 1$. Since $l_1 = 16$ and $l_2 = 12$, we now fall into the second case of our algorithm and the recursive invocation of the algorithm is done between the third and the fourth image.

5. MULTIPLYING MULTIVARIATE POLYNOMIALS

Let $\mathcal{R}$ be a ring with a privileged sequence $(\omega_1, \omega_2, \omega_4, \ldots)$ of roots of unity (see remark 1). Given a non-zero multivariate polynomial

$$f = \sum_{i_1,\ldots,i_d} f_{i_1,\ldots,i_d} z_1^{i_1} \cdots z_d^{i_d} \in \mathcal{R}[z_1, \ldots, z_d],$$

in $d > 1$ variables, we define the *total degree* of f by

$$\deg f = \max\{i_1 + \cdots + i_d : f_{i_1,\ldots,i_d} \neq 0\} \in \mathbb{N}$$

We let $\deg 0 = -1$. Now let $f, g \in \mathcal{R}[z_1, \ldots, z_d]$ be such that $\deg fg < r$. In this section we present an algorithm to compute fg, which has a good complexity in terms of the number

$$s = \binom{r+d-1}{d}$$

of expected coefficients of fg. When computing fg using the classical FFT with respect to each of the variables $z_1, \ldots, z_d$, we need a time $O(d(2r)^d \log r))$ which is much bigger than s, in general. When using multiplication of sparse polynomials [2], we need a time $O(s \log^2 s)$ with a non-trivial constant factor. Our algorithm is based on the TFT w.r.t. all variables and we will show that it has a complexity $O(s \log s)$.

Given $f \in \mathcal{R}[z_1, \ldots, z_d]$ with $\deg f < r$, the TFT of f with respect to one variable z_v at order r is defined by

$$\mathrm{TFT}_{v;r}(f) = \left(f_{i_1,\ldots,i_{v-1},\cdot,i_{v+1},\ldots,i_d}(z_v = \omega_v^{[i_v]_p})\right)_{i_1+\cdots+i_d<r},$$

where $n = 2^p \geqslant r$. We recall that the result does not depend on the choice of n. The TFT with respect to all variables $z_1, \ldots, z_d$ at order r is defined by

$$\mathrm{TFT}_{;r}(f) = \left(f(\omega_n^{[i_1]_p}, \ldots, \omega_n^{[i_d]_p})\right)_{i_1+\cdots+i_d<r},$$

where $n = 2^p \geqslant r$ (see figure 5). We have

$$\mathrm{TFT}_{;r}(f) = \mathrm{TFT}_{d;r}(\cdots\ \mathrm{TFT}_{1;r}(f)\ \cdots).$$

Given $f, g \in \mathcal{R}[z_1, \ldots, z_d]$ with $\deg fg < r$, we will use the formula

$$fg = \mathrm{TFT}_{;r}^{-1}(\mathrm{TFT}_{;r}(f)\, \mathrm{TFT}_{;r}(g))$$

in order to compute the product fg.

In order to compute $\mathrm{TFT}_{v;r}(f)$, say for $v = 1$, we compute the TFT of $(f_{0,i_2,\ldots,i_d}, \ldots, f_{l-1,i_2,\ldots,i_d})$ with $l = r - i_2 - \cdots - i_r$ for all $i_2, \ldots, i_d$ with $i_2 + \cdots + i_n < r - 1$ (if $i_2 + \cdots + i_n = r - 1$, then the TFT of $(f_{0,i_2,\ldots,i_d})$ is given by itself, so we have nothing to do). One such computation takes a time $\leqslant Cl \log l$ for some universal constant C, by using the TFT w.r.t. ω_n with minimal $n = 2^p \geqslant l$ (so n may vary as a function of $i_2, \ldots, i_d$, but not C). The computation of $\mathrm{TFT}_{;r}(f)$ therefore takes a time $T_{d,r}$ with

$$T_{d,r} \leqslant Cd \sum_{l=2}^{r} \binom{r-l+d-2}{d-2} l \log l.$$

Dividing by s, we obtain

$$\begin{aligned} \tfrac{T_{d,r}}{s} &\leqslant Cd^2(d-1) \sum_{l=2}^{r} \tfrac{(r+d-l-2)!}{(r+d-1)!} \tfrac{(r-1)!}{(r-l)!} l \log l \\ &\leqslant Cd^3 \sum_{l=2}^{r} \tfrac{(r+d-l)!}{(r+d-1)!} \tfrac{(r-1)!}{(r-l)!} \tfrac{l \log l}{(r-l+d)(r-l+d-1)} \qquad (3) \end{aligned}$$

If $r \leqslant d$, then the summand rapidly deceases when $l > 2$, so that

$$\frac{T_{d,r}}{s} = O(d^3 \frac{r}{(r+d)^3}) = O(r) = O(\log s).$$

Consequently, $T_{d,r} = O(s \log s)$ and even $T_{d,r} = O(s)$ for fixed r. If $r > d$, then for $d = \varepsilon r$ and $l = \delta r$, Stirling's formula yields

$$\log \frac{(r+d-l)!}{(r+d-1)!} \frac{(r-1)!}{(r-l)!} = -\varepsilon\delta r + \cdots .$$

It follows that only the first $O(r/d) = O(1/\varepsilon)$ terms in (3) contribute to the asymptotic behaviour of $\frac{T_{d,r}}{s}$, so that

$$T_{d,r} = O(d^3 \frac{1}{\varepsilon} \frac{\log(1/\varepsilon)}{\varepsilon r^2}) = O(d \log(r/d)) = O(\log s).$$

Again, we find that $T_{d,r} = O(s \log s)$. We have proved:

THEOREM 3. *Let $\mathcal{R}$ be a ring with a privileged sequence $(\omega_1, \omega_2, \omega_4, \ldots)$ of roots of unity. Let $f, g \in \mathcal{R}[z_1, \ldots, z_d]$ be polynomials with $\deg f + \deg g < r$ and let $s = \binom{r+d-1}{r}$. Then the product fg can be computed using $O(s \log s)$ ring operations in $\mathcal{R}$.*

6. MULTIPLYING MULTIVARIATE POWER SERIES

Since power series have infinitely many terms, implementing an operation on power series really corresponds to implementing the operation for polynomial approximations at all degrees. As usual, multiplication is a particularly important operation. Given $f, g \in \mathcal{R}[z_1, \ldots, z_d]$ with $\deg f < r$ and $\deg g < r$, we will show how to compute the truncated product $h = \sum_{i_1,\ldots,i_d<d} (fg)_{i_1,\ldots,i_d} z_1^{i_1} \cdots z_d^{i_d} \in \mathcal{R}[z_1, \ldots, z_d]$ of f and g.

The first idea [8] is to use homogeneous coordinates instead of the usual ones:

$$\begin{aligned} \tilde{f}(z_1, z_2, \ldots, z_d) &= f(z_1, z_1 z_2, \ldots, z_1 z_d) \\ \tilde{g}(z_1, z_2, \ldots, z_d) &= g(z_1, z_1 z_2, \ldots, z_1 z_d). \end{aligned}$$

This transformation takes no time since it corresponds to some re-indexing. We next compute the TFTs $\hat{f}$ and $\hat{g}$ in $z_2, \ldots, z_d$ at order r:

$$\begin{aligned} \hat{f} &= \mathrm{TFT}_{d;r}(\cdots\ \mathrm{TFT}_{2;r}(\tilde{f})\ \cdots) \\ \hat{g} &= \mathrm{TFT}_{d;r}(\cdots\ \mathrm{TFT}_{2;r}(\tilde{g})\ \cdots). \end{aligned}$$

We next compute the $s' = \binom{r+d-2}{d-1}$ truncated products $\hat{h}_{\cdot,i_2,\ldots,i_d}(z_1)$ of the obtained polynomials $\hat{f}_{\cdot,i_2,\ldots,i_d}(z_1)$ and $\hat{g}_{\cdot,i_2,\ldots,i_d}(z_1)$. After transforming the results of these multiplication back using

$$\tilde{h} = \mathrm{TFT}_{2;r}^{-1}(\cdots\ \mathrm{TFT}_{d;r}^{-1}(\hat{h})\ \cdots),$$

we obtain the truncated product h of f and g by

$$h(z_1, z_2, \ldots, z_d) = \tilde{h}(z_1, z_2/z_1, \ldots, z_d/z_1).$$

The total computation time is bounded by $O(rs' \log s' + rs' \log r)$. Using the fact that $rs' = O(s \log s)$, we have proved the following theorem:

THEOREM 4. *Let $\mathcal{R}$ be a ring with a privileged sequence $(\omega_1, \omega_2, \omega_4, \ldots)$ of roots of unity. Let $f, g \in \mathcal{R}[z_1, \ldots, z_d]$ be polynomials of degrees $< r$ and let $s = \binom{r+d-1}{r}$. Then the truncated product of f and g at degree $< r$ can be computed using $O(s \log^2 s)$ ring operations in $\mathcal{R}$.*

$f_{0,5}$					
$f_{0,4}$	$f_{1,4}$				
$f_{0,3}$	$f_{1,3}$	$f_{2,3}$			
$f_{0,2}$	$f_{1,2}$	$f_{2,2}$	$f_{3,2}$		
$f_{0,1}$	$f_{1,1}$	$f_{2,1}$	$f_{3,1}$	$f_{4,1}$	
$f_{0,0}$	$f_{1,0}$	$f_{2,0}$	$f_{3,0}$	$f_{4,0}$	$f_{5,0}$

$\rightarrow$

$f(1,\omega^3)$					
$f(1,\omega^5)$	$f(\omega^4,\omega^5)$				
$f(1,\omega^6)$	$f(\omega^4,\omega^6)$	$f(\omega^2,\omega^6)$			
$f(1,\omega^2)$	$f(\omega^4,\omega^2)$	$f(\omega^2,\omega^2)$	$f(\omega^6,\omega^2)$		
$f(1,\omega^4)$	$f(\omega^4,\omega^4)$	$f(\omega^2,\omega^4)$	$f(\omega^6,\omega^4)$	$f(\omega^5,\omega^4)$	
$f(1,1)$	$f(\omega^4,1)$	$f(\omega^2,1)$	$f(\omega^6,1)$	$f(\omega^5,1)$	$f(\omega^3,1)$

Figure 5: Illustration of the TFT in two variables ($\omega = \omega_8$).

Remark 5. In practice, if the coefficients $f_{i_1,\ldots,i_d}$ have different growths in $i_1, \ldots, i_d$, then it may be useful to consider truncations along more general degrees of the form

$$\deg_{\boldsymbol{\alpha}} f = \max\{\alpha_1 i_1 + \cdots + \alpha_d i_d : f_{i_1,\ldots,i_d} \neq 0\}.$$

The "slicing technique" from section 6.3.5 in [13] may then be used in order to obtain complexity bounds of the same type.

Remark 6. Using remark 3, the polynomial and truncated multiplication algorithms can be used in combination with the strategy of relaxed evaluation [12, 13, 15] for solving partial differential equations in multivariate power series with an additional overhead of $O(\log r)$. A recent technique [14] allows to reduce this overhead even further and it would be interesting to study more precisely what happens in the multivariate case.

7. FINAL NOTES

The author would like to thank the first referee for his enthusiastic and helpful comments. This referee also implemented the algorithms from sections 3 and 4 and he reports a behaviour which is close to the expected one. In response to some other comments and suggestions, we conclude with the following remarks:

- The results of the paper may be generalized to characteristic 2 and general rings $\mathcal{R}$ along similar lines as in [3]. The crucial remark is that, if $j^3 = 1$ and

$$\begin{pmatrix} a_1 \\ b_1 \\ c_1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & j & j^2 \\ 1 & j^2 & j \end{pmatrix} \begin{pmatrix} a_0 \\ b_0 \\ c_0 \end{pmatrix},$$

  then, for all $\epsilon \in \{0,1\}^3$, we may compute $(a_\epsilon, b_\epsilon, c_\epsilon)$ in terms of $(a_{1-\epsilon}, b_{1-\epsilon}, c_{1-\epsilon})$ by using only additions, subtractions, multiplications by j and divisions by 3.

- Theorem 1 in [2] implies theorem 3 with $O(s \log s)$ replaced by $O(s \log^2 s)$. The technique from [2] is actually more general: let $f, g \in \mathcal{R}[z_1, \ldots, z_d]$ and assume that we know

$$\begin{aligned} \mathcal{S} &= \operatorname{supp} f \operatorname{supp} g \\ &= \{(i_1 + j_1, \ldots, i_d + j_d) : f_{i_1,\ldots,i_d} \neq 0 \wedge g_{j_1,\ldots,j_d}\}. \end{aligned}$$

  If f and g are not "extraordinarily sparse", then fg may be computed in time $O((\#\mathcal{S}) \log^2(\#\mathcal{S}))$. It would be interesting to prove something similar in our context, so as to examine to which extent we need the density hypothesis. Using remark 3 in a recursive way, we expect that there exists an algorithm of complexity $O(\#\mathcal{S}(\log \#\mathcal{S} + \#\partial\mathcal{S}))$, for a suitable definition of $\partial\mathcal{S}$.

- The terminology of privileged sequences may seem to be an overkill. Indeed, in practice, we rather need a sufficiently large root of unity in order to carry out a given computation. Nevertheless, from a theoretical point of view, this paper suggests that it may be interesting to study "fractal FFT-transforms"

$$\begin{aligned} & f_0 + f_1 z + f_2 z^2 + \cdots \\ \rightarrow\ & (f(1), f(\omega_2^{[1]_1}), f(\omega_4^{[2]_2}), f(\omega_4^{[3]_2}), f(\omega_8^{[4]_3}), \ldots) \end{aligned}$$

  of power series with convergence radius > 1 with respect to a privileged sequence $(\omega_1, \omega_2, \omega_4, \ldots)$.

- Two referees pointed us to the on-line paper [1] which also contains the idea of evaluating in l powers of ω_n in order to multiply polynomials f, g with $\deg fg = l < n = 2^p$. However, while we are writing these lines, this paper does not contain a precise algorithm for the inverse transform (cf. section 3), nor any claims about the complexity (cf. theorems 1 and 2).

8. REFERENCES

[1] Bernstein, D. Fast multiplication and its applications. Available from `http://cr.yp.to/papers.html#multapps`. See section 4, page 11.

[2] Canny, J., Kaltofen, E., and Lakshman, Y. Solving systems of non-linear polynomial equations faster. In *Proc. ISSAC '89* (Portland, Oregon, A.C.M., New York, 1989), ACM Press, pp. 121–128.

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

[4] Cooley, J., and Tukey, J. An algorithm for the machine calculation of complex Fourier series. *Math. Computat. 19* (1965), 297–301.

[5] Hanrot, G., Quercia, M., and Zimmermann, P. Speeding up the division and square root of power series. Research Report 3973, INRIA, July 2000.

Available from
`http://www.inria.fr/RRRT/RR-3973.html`.

[6] Hanrot, G., Quercia, M., and Zimmermann, P. The middle product algorithm I. speeding up the division and square root of power series. Accepted for publication in AAECC, 2002.

[7] Hanrot, G., and Zimmermann, P. A long note on Mulders' short product. *JSC 37*, 3 (2004), 391–401.

[8] Lecerf, G., and Schost, E. Fast multivariate power series multiplication in characteristic zero. *SADIO Electronic Journal on Informatics and Operations Research 5*, 1 (September 2003), 1–10.

[9] Mulders, T. On short multiplication and division. *AAECC 11*, 1 (2000), 69–88.

[10] Pan, V. Y. Simple multivariate polynomial multiplication. *JSC 18*, 3 (1994), 183–186.

[11] Schönhage, A., and Strassen, V. Schnelle Multiplikation grosser Zahlen. *Computing 7 7* (1971), 281–292.

[12] van der Hoeven, J. Lazy multiplication of formal power series. In *Proc. ISSAC '97* (Maui, Hawaii, July 1997), W. W. Küchlin, Ed., pp. 17–20.

[13] van der Hoeven, J. Relax, but don't be too lazy. *JSC 34* (2002), 479–542.

[14] van der Hoeven, J. New algorithms for relaxed multiplication. Tech. Rep. 2003-44, Univ. d'Orsay, 2003.

[15] van der Hoeven, J. Relaxed multiplication using the middle product. In *Proc. ISSAC '03* (Philadelphia, USA, August 2003), M. Bronstein, Ed., pp. 143–147.