Tellegen's Principle into Practice

A. Bostan
Laboratoire STIX
FRE CNRS 2341
École polytechnique
91128 Palaiseau, France
bostan@stix.polytechnique.fr

G. Lecerf
Laboratoire de Mathématiques
UMR CNRS 8100
Université de Versailles
St-Quentin-en-Yvelines
45, avenue des États-Unis
78035 Versailles, France
lecerf@math.uvsq.fr

É. Schost
Laboratoire STIX
FRE CNRS 2341
École polytechnique
91128 Palaiseau, France
schost@stix.polytechnique.fr

ABSTRACT

The transposition principle, also called Tellegen's principle, is a set of transformation rules for linear programs. Yet, though well known, it is not used systematically, and few practical implementations rely on it. In this article, we propose explicit transposed versions of polynomial multiplication and division but also new faster algorithms for multipoint evaluation, interpolation and their transposes. We report on their implementation in Shoup's NTL C++ library.

Categories and Subject Descriptors

F.2 [**Theory of Computation**]: Analysis of Algorithms and Problem Complexity; D.4.8 [**Mathematics of Computing**]: Mathematical Software

General Terms

Algorithm, Theory

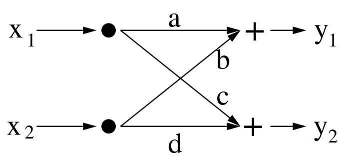
Keywords

Computer algebra, interpolation, polynomial evaluation, Tellegen's principle, transposition principle

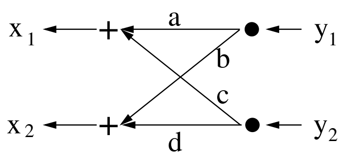
1. INTRODUCTION

The *transposition principle*, sometimes referred to as *Tellegen's principle*, asserts that a linear algorithm that performs a matrix-vector product can be *transposed*, producing an algorithm that computes the transposed matrix-vector product. Further, the transposed algorithm has almost the same complexity as the original one (see Section 3 for precise statements).

The following example illustrates this principle, using the computation graph representation. Taking x_1, x_2 as input, it computes $y_1 = ax_1 + bx_2, y_2 = cx_1 + dx_2$; edges perform multiplications by the constant values a, b, c, d.



Reversing all arrows and exchanging vertices $+$ and $\bullet$ yield the following graph:



Taking y_1, y_2 as input, it computes the transposed map $x_1 = ay_1 + cy_2, x_2 = by_1 + dy_2$ (see [16] for details).

Such transformation techniques originate from linear circuit design and analysis [1, 3, 19, 24] and were introduced in computer algebra in [7, 8, 12, 16]. Since then, there has been a recurrent need for transposed algorithms [2, 4, 6, 10, 13, 14, 21, 22, 23, 25]. Yet, the transposition principle in itself is seldom applied, and specific algorithms were often developed to circumvent its use, with the notable exceptions of [10, 22].

Contributions. In this paper, we detail several linear algorithms for univariate polynomials and their transposes: multiplication, quotient, remainder, evaluation, interpolation and exemplify a systematic use of Tellegen's principle.

Our first contribution concerns univariate polynomial remaindering: we show that this problem is dual to extending linear recurrence sequences with constant coefficients. This clarifies the status of algorithms by Shoup [21, 23], which were designed as alternatives to the transposition principle: they are actually the transposes of well-known algorithms.

Our second contribution is an improvement (by a constant factor) of the complexities of multipoint evaluation and interpolation. This is done by designing a fast algorithm for transposed evaluation, and transposing it backwards. We also improve the complexity of performing several multipoint evaluations at the same set of points: discarding the costs of the precomputations, multipoint evaluation and interpolation now have very similar complexities.

ISSAC'03, August 3–6, 2003, Philadelphia, Pennsylvania, USA.

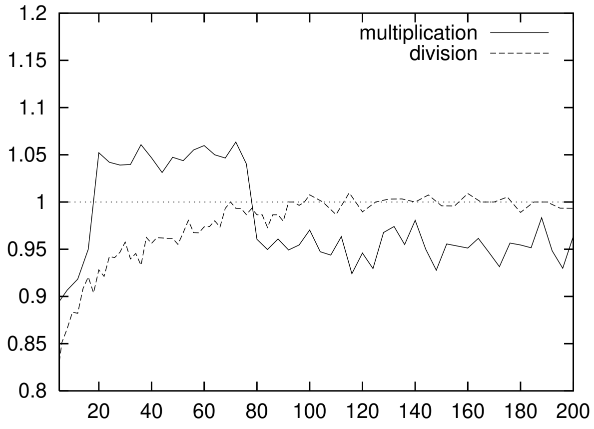


Figure 1: Transposed/direct ratios

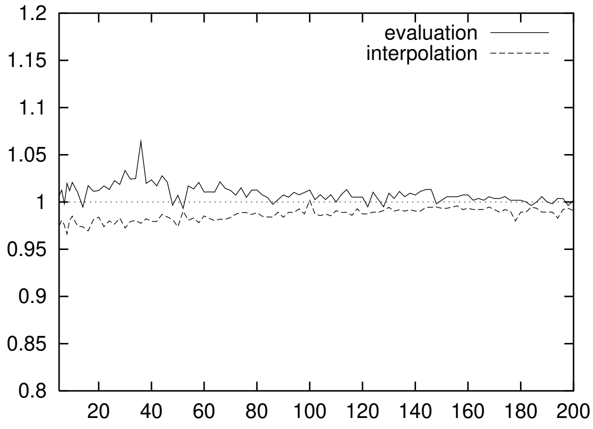


Figure 2: Transposed/direct ratios

Finally, we demonstrate that the transposition principle is quite practical. We propose (still experimental) NTL [20] implementations of all algorithms mentioned here and their transposes. They show that the expected time ratio of 1 between an algorithm and its transpose is well respected in practice. The source code can be downloaded from `http://www.math.uvsq.fr/~lecerf`.

Contents. Section 2 introduces the notation used in the rest of this article. In Section 3, we state the transposition principle in two distinct computation models; we discuss the memory requirement question that was raised in Kaltofen's Open Problem 6 in [13].

In Section 4, we transpose polynomial multiplication. This was already done in [10], where transposed versions of Karatsuba's and Fast Fourier Transform (FFT) multiplications were given. We transpose NTL Karatsuba's multiplication, which was optimized to treat unbalanced situations. We also transpose plain, as well as FFT multiplication.

In Section 5, we transpose the operations of polynomial quotient and remainder and show the duality between polynomial remaindering and extending linear recurrence sequences.

In Section 6, we finally consider polynomial multipoint evaluation and interpolation. We give algorithms for these operations and their transposes that improve previously known algorithms by constant factors.

Implementation. Our algorithms are implemented in the C++ library NTL [20]. Figures 1 and 2 describe the behavior of our implementation; the computations were done over $\mathbb{Z}/p\mathbb{Z}$, with p a prime of 64 bit length, and the times were measured on a 32 bit architecture.

Figure 1 shows the time ratios between direct and transposed algorithms for polynomial multiplication and polynomial remainder, for degrees up to 200.

- For multiplication, the horizontal axis gives the degree m of the input polynomials. Multiplication in NTL uses three different algorithms: plain multiplication for $m \leq 19$, Karatsuba's multiplication for $20 \leq m < 79$ and FFT for larger m. The same thresholds are used for the transposed versions. Note that Karatsuba's multiplication is slightly slower in its transposed version.
- The division deals with polynomials of degrees $2m$ and m. NTL provides the plain remainder algorithm and a faster one based on Sieveking-Kung's algorithm, with further optimizations. The threshold is 90, we used the same value for the transposed algorithms.

Figure 2 presents the ratios between the direct and transposed versions for our new fast multipoint evaluation and interpolation algorithms, with degree on the horizontal axis (Section 6 gives their complexity estimates).

Remark. Transposed algorithms were used for modular multiplication and power series inversion in NTL as soon as [22], yet only for FFT multiplication. More generally the transposed product is used in [10] to speed up some algorithms for power series.

2. DEFINITIONS AND NOTATION

Let R be a commutative ring with unity and $n \geq 0$. By $R[x]_n$ we denote the free R-module of polynomials of degree at most n. We use the monomial basis $1, x, \ldots, x^n$ on $R[x]_n$ to represent polynomials by vectors and linear maps by matrices. It will also be convenient to consider elements of $R[x]$ as infinite sequences with finite support: we will write a polynomial a as $\sum_{i\geq 0} a_i x^i$, where almost all a_i vanish. The degree of a is denoted by $\deg(a)$, with $\deg(0) = -\infty$.

For $l \geq 0$ and $h \geq 0$, we introduce the following maps $\lceil . \rceil^h$, $\lfloor . \rfloor_l$, $\lceil . \rfloor_l^h$ on $R[x]$ and the power series ring $R[[x]]$:

$$\lceil a \rceil^h = \sum_{i=0}^{h-1} a_i x^i, \ \lfloor a \rfloor_l = \sum_{i\geq 0} a_{i+l} x^i, \ \lceil a \rfloor_l^h = \sum_{i=0}^{h-l-1} a_{i+l} x^i.$$

Observe that these maps satisfy the following relations:

$$a = \lceil a \rceil^h + x^h \lfloor a \rfloor_h, \quad \lceil a \rfloor_l^h = \left\lceil \lfloor a \rfloor_l \right\rceil^{h-l} = \left\lfloor \lceil a \rceil^h \right\rfloor_l.$$

We define the *reversal* endomorphism $\operatorname{rev}(n, .)$ of $R[x]_n$ by $\operatorname{rev}(n, a) = \sum_{k=0}^{n} a_{n-k} x^k$, for all $a \in R[x]_n$.

By $\lfloor q \rfloor$ we denote the integer part of a rational q. Finally, we use a block matrix notation: $0_{h,l}$ denotes a $h \times l$ block filled with zeros and 1_h the $h \times h$ identity matrix.

3. TELLEGEN'S PRINCIPLE

Tellegen's principle is usually stated using linear computation graphs [16, 24]. In this section, we first briefly recall it in terms of *linear straight-line programs.* We then define another computational model, and prove Tellegen's principle in that setting. The first model can be thought as taking only time complexity into account, while the second one considers both time and space.

Linear Straight-Line Programs. Linear straight-line programs can be thought as "ordinary" straight-line programs, but with only linear operations, see [5, Chapter 13] for precise definitions. Their complexity is measured by their number of operations, the *size*, which reflects a *time* complexity. In this model, Tellegen's principle can be formulated as:

PROPOSITION 1. [5, Th. 13.20] *Let $\phi : R^n \to R^m$ be a linear map that can be computed by a linear straight-line program of size L and whose matrix in the canonical bases has z_r zero rows and z_c zero columns. Then the transposed map ϕ^t can be computed by a linear straight-line program of size $L - n + m - z_r + z_c$.*

Let us remark that Theorem 13.20 in [5] actually expresses this result in terms of *linear complexity* of ϕ, that is, the *minimum* size of a linear straight-line program computing ϕ. A quick look at the proof shows that the statement in the proposition above is implicitely proved therein. We will use Proposition 1 in Section 6, as the complexity estimates in that section are given in the linear straight-line program model.

Another Computational Model. We now introduce a measure of space complexity. Informally speaking, our model derives from the *random access memory* model by restricting to linear operations. In this context, a program $\mathcal{P}$ is given by:

- A finite set $\mathcal{D}$ of *registers*, with two distinguished subsets (non necessarily disjoint), $\mathcal{I}$ for the *input* and $\mathcal{O}$ for the *output*: before execution $\mathcal{I}$ contains the input values, at the end $\mathcal{O}$ contains the output.
- A finite sequence of instructions $(\phi_i)_{i\in\{1,\dots,L\}}$ of length L. Each instruction is a linear endomorphism of $R^{\mathcal{D}}$ of the following type, with p, q in $\mathcal{D}$, a in R and f in $R^{\mathcal{D}}$:
  - $\mathbf{p} += \mathbf{q}$ denotes the map that sends f to the function g defined by $g(r) = f(r)$ if $r \neq p$ and $g(p) = f(p) + f(q)$.
  - $\mathbf{p} *= \mathbf{a}$ denotes the map that sends f to the function g defined by $g(r) = f(r)$ if $r \neq p$ and $g(p) = af(p)$.
  - $\mathbf{p} = \mathbf{0}$ denotes the map that sends f to the function g defined by $g(r) = f(r)$ if $r \neq p$ and $g(p) = 0$.

Let I (resp. P) be the injection $R^{\mathcal{I}} \to R^{\mathcal{D}}$ (resp. the projection $R^{\mathcal{D}} \to R^{\mathcal{O}}$). Then we say that the program $\mathcal{P}$ *computes* the linear map $\Phi : R^{\mathcal{I}} \to R^{\mathcal{O}}$ defined by $\mathsf{P} \circ \phi_L \circ \phi_{L-1} \circ \cdots \circ \phi_1 \circ \mathsf{I}$.

We can now define the *transpose* $\mathcal{P}^t(\mathcal{D}^t, \mathcal{I}^t, \mathcal{O}^t)$ of $\mathcal{P}$:

- The set of registers of $\mathcal{P}^t$ is still $\mathcal{D}^t = \mathcal{D}$ but we let $\mathcal{I}^t = \mathcal{O}$ and $\mathcal{O}^t = \mathcal{I}$: input and output are swapped.
- The instructions of $\mathcal{P}^t$ are $\phi_L^t, \phi_{L-1}^t, \dots, \phi_1^t$.

Let us verify that $\mathcal{P}^t$ is well-defined. We examine the transpose of each type of instructions:

- The transpose ϕ^t of an instruction ϕ of type $p += q$ is the instruction $q += p$.
- The last two instructions are symmetric maps.

The equality $(\phi_L \circ \phi_{L-1} \circ \cdots \circ \phi_1)^t = \phi_1^t \circ \phi_2^t \circ \cdots \circ \phi_L^t$ then shows that $\mathcal{P}^t$ computes the linear map Φ^t. This yields the following form of Tellegen's principle:

PROPOSITION 2. *According to the above notation, let $\phi : R^n \to R^m$ be a linear map that can be computed by a program with D registers and L instructions. Then ϕ^t can be computed by a program with D registers and L instructions.*

As an example, consider 5 registers, x_1, x_2 for input, y_1, y_2 for output and r for temporaries. Given a, b, c, d in R, consider the instructions: $y_1 += x_1$, $y_1 *= a$, $r += x_2$, $r *= b$, $y_1 += r$, $r = 0$, $y_2 += x_1$, $y_2 *= c$, $r += x_2$, $r *= d$, $y_2 += r$. We let the reader check that the linear map computed by this program is the 2×2 matrix-vector product presented in the introduction. The transposed program is $r += y_2$, $r *= d$, $x_2 += r$, $y_2 *= c$, $x_1 += y_2$, $r = 0$, $r += y_1$, $r *= b$, $x_2 += r$, $y_1 *= a$, $x_1 += y_1$.

We could have used more classical instructions such as $p = q + r$ and $p = aq$, $a \in R$. We let the reader check that such programs can be rewritten in our model within the same space complexity but a constant increase of time complexity. For such programs, Tellegen's principle would be stated with no increase in space complexity, but a constant increase in time complexity. This is why we use straight-line programs for our complexity estimates in Section 6.

Open Problem 6 in [13] asks for a transposition theorem without space complexity swell. The above Proposition sheds new light on this problem: In the present computational model, it is immediate to observe that memory consumption is left unchanged under transposition.

Comments. The above models compute functions of fixed input and output size: in the sequel, we will actually write families of programs, one for each size. We also use control instructions as **for** and **if**, and calls to previously defined subroutines. Last, we will consider algorithms mixing linear and non-linear precomputations; the transposition principle leaves the latter unchanged.

4. POLYNOMIAL MULTIPLICATION

In this section a is a fixed polynomial of degree m. For an integer $n \geq 0$, consider the multiplication map by a:

$$\begin{array}{rccl} \mathrm{mul}(a,.): & R[x]_n & \to & R[x]_{m+n} \\ & b & \mapsto & ab. \end{array}$$

The transposed map is denoted by $\mathrm{mul}^t(n, a, .)$; to write our pseudo-code, it is necessary to consider n as an argument of the transposed function. The next subsection shows that this map is:

$$\begin{array}{rccl} \mathrm{mul}^t(n,a,.): & R[x]_{m+n} & \to & R[x]_n \\ & c & \mapsto & [\mathrm{rev}(m,a)c]_m^{n+m+1}. \end{array}$$

We adopt the following convention: if $\deg(c) > m+n$ then $\mathrm{mul}^t(n,a,c)$ returns an error.

The above formula explains why the transposed multiplication is also called the *middle product* [10]. Performing this operation fast is the first task to accomplish before transposing higher level algorithms. Observe that computing $\mathrm{mul}^t(n,a,c)$ by multiplying $\mathrm{rev}(m,a)$ by c before extracting the middle part requires to multiply two polynomials of degrees m and $m+n$.

Tellegen's principle implies that this transposed computation can be performed for the cost of the multiplication of two polynomials of degrees m and n only. In what follows, we make this explicit for the plain, Karatsuba and Fast Fourier Transform multiplications.

4.1 Plain multiplication

In the canonical monomial bases, the matrix of the linear map $\mathrm{mul}(a,.) : R_n[x] \to R_{m+n}[x]$ is the following Toeplitz matrix T with $m+n+1$ rows and $n+1$ columns:

$$T = \begin{bmatrix} a_0 & 0 & \dots & 0 \\ a_1 & a_0 & \ddots & \vdots \\ \vdots & a_1 & \ddots & 0 \\ \vdots & \vdots & \ddots & a_0 \\ a_m & \vdots & \ddots & a_1 \\ 0 & a_m & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & a_m \end{bmatrix}.$$

Basically, the plain multiplication algorithm corresponds to performing the matrix vector product $c = Tb$ naively using the following sequence of instructions:

$c \leftarrow 0$;
for i **from** 0 **to** $m+n$ **do**
    for j **from** $\max(0, i-n)$ **to** $\min(m,i)$ **do**
        $c_i \leftarrow c_i + a_j b_{i-j}$;

Reversing the computation flow, $b = \mathrm{mul}^t(n,a,c)$ is computed by the following program:

$b \leftarrow 0$;
for i **from** $m+n$ **downto** 0 **do**
    for j **from** $\min(m,i)$ **downto** $\max(0,i-n)$ **do**
        $b_{i-j} \leftarrow b_{i-j} + a_j c_i$;

Observing that the entries of b are given by:

$$b_i = \sum_{j=i}^{i+m} a_{j-i} c_j, \quad i \in \{0, \dots, n\},$$

we deduce that the map $\mathrm{mul}^t(n,a,.)$ reformulates in terms of the *middle product* [10] by the relation

$$\mathrm{mul}^t(n,a,c) = [\mathrm{rev}(m,a)c]_m^{n+m+1},$$

as announced above.

4.2 Karatsuba's algorithm

The next paragraphs are devoted to the transposition of Karatsuba's multiplication algorithm. Concretely, we consider the NTL implementation and present the transpose version we made of it. Figures 3 and 4 describe Karatsuba's algorithm and its transpose. As for the mul^t function, the transposed Karatsuba multiplication takes n as an additional argument.

Figure 3: KarMul(a,b)

$m \leftarrow \deg(a)$;
$n \leftarrow \deg(b)$;
if $n = 0$ **then return** $b_0 a$;
if $m \leq 0$ **then return** $a_0 b$;
$\mu \leftarrow \lfloor m/2 \rfloor + 1$;
$\nu \leftarrow \lfloor n/2 \rfloor + 1$;
if $\mu > n$ **then**
    $u \leftarrow \mathrm{KarMul}(\lceil a \rceil^{\mu}, b)$;
    $v \leftarrow \mathrm{KarMul}(\lfloor a \rfloor_{\mu}, b)$;
    return $u + x^{\mu} v$;
if $\nu > m$ **then**
    $u \leftarrow \mathrm{KarMul}(a, \lceil b \rceil^{\nu})$;
    $v \leftarrow \mathrm{KarMul}(a, \lfloor b \rfloor_{\nu})$;
    return $u + x^{\nu} v$;
$h \leftarrow \max(\mu, \nu)$;
$r \leftarrow \mathrm{KarMul}(\lceil a \rceil^h, \lceil b \rceil^h)$;
$s \leftarrow \mathrm{KarMul}(\lfloor a \rfloor_h, \lfloor b \rfloor_h)$;
$t \leftarrow \mathrm{KarMul}(\lceil a \rceil^h + \lfloor a \rfloor_h, \lceil b \rceil^h + \lfloor b \rfloor_h)$;
return $r + x^h(t - s - r) + x^{2h} s$;

In order to prepare the proof of Algorithm TKarMul we first decompose Algorithm KarMul into linear maps. Following Algorithm KarMul we enter the procedure with polynomials a of degree m and b of degree n. We let $\mu = \lfloor m/2 \rfloor + 1$, $\nu = \lfloor n/2 \rfloor + 1$ and $h = \max(\mu, \nu)$ and distinguish three exclusive cases:

Normal case: ($m < n$ and $\nu \leq m$) or ($m \geq n$ and $\mu \leq n$). Note that $\deg(\lfloor a \rfloor_h) = m - h$, let $\rho = \deg(\lceil a \rceil^h)$ and $\lambda = \deg(\lceil a \rceil^h + \lfloor a \rfloor_h)$. Let $M_l : R[x]_{h-1} \to R[x]_{\rho+h-1}$, $M_h : R[x]_{n-h} \to R[x]_{m+n-2h}$ and $M_f : R[x]_{h-1} \to R[x]_{\lambda+h-1}$ be the linear maps of multiplication by resp. $\lceil a \rceil^h$, $\lfloor a \rfloor_h$ and $\lceil a \rceil^h + \lfloor a \rfloor_h$. Noticing that $n - h \leq h - 1$, we construct the following block matrix M, where we take $r = m+n$ and $q = \rho + h$:

$$M = \left[\begin{array}{c} \begin{array}{c|c} M_l & 0_{q,n-h+1} \\ \hline 0_{r-2h+1,h} & M_h \end{array} \\ \hline M_f \cdot \left[\begin{array}{c|c} 1_h & \begin{array}{c} 1_{n-h+1} \\ \hline 0_{2h-n-1,n-h+1} \end{array} \end{array}\right] \end{array}\right].$$

Remark that $(r,s,t) = Mb$, when considering that $r \in R[x]_{\rho+h-1}$, $s \in R[x]_{m+n-2h}$ and $t \in R[x]_{\lambda+h-1}$. In order to recover the product $c = ab$ it remains to compute $c = (N_+ - N_-)(r,s,t)$, with

$$N_+ = \left[\begin{array}{c|c|c} \begin{array}{c} 1_{\rho+h} \\ \hline 0_{r+1-q,q} \end{array} & \begin{array}{c} 0_{2h,r-2h+1} \\ \hline 1_{r-2h+1} \end{array} & \begin{array}{c} 0_{h,\lambda+h} \\ \hline 1_{\lambda+h} \\ \hline 0_{r+1-\lambda-2h,\lambda+h} \end{array} \end{array}\right],$$

$$N_- = \left[\begin{array}{c|c|c} \begin{array}{c} 0_{h,\rho+h} \\ \hline 1_{\rho+h} \\ \hline 0_{r-\rho-2h+1,\rho+h} \end{array} & \begin{array}{c} 0_{h,r-2h+1} \\ \hline 1_{r-2h+1} \\ \hline 0_{h,r-2h+1} \end{array} & 0_{r+1,\lambda+h} \end{array}\right].$$

Figure 4: TKarMul(n, a, c)

$m \leftarrow \deg(a)$;
if $\deg(c) > m+n$ **then** Error;
if $n = 0$ **then return** $\sum_{k=0}^{m} a_k c_k$;
if $m \leq 0$ **then return** $a_0 c$;
$\mu \leftarrow \lfloor m/2 \rfloor + 1$;
$\nu \leftarrow \lfloor n/2 \rfloor + 1$;
$\rho \leftarrow \deg(\lceil a \rceil^h)$;
if $\mu > n$ **then**
  $u \leftarrow$ TKarMul$(n, \lceil a \rceil^{\mu}, \lceil c \rceil^{n+\rho})$;
  $v \leftarrow$ TKarMul$(n, \lfloor a \rfloor_{\mu}, \lfloor c \rfloor_{\mu})$;
  return $u + v$;
if $\nu > m$ **then**
  $u \leftarrow$ TKarMul$(\nu - 1, a, \lceil c \rceil^{m+\nu})$;
  $v \leftarrow$ TKarMul$(n - \nu, a, \lfloor c \rfloor_{\nu})$;
  return $u + v$;
$h \leftarrow \max(\mu, \nu)$;
$\lambda \leftarrow \deg(\lceil a \rceil^h + \lfloor a \rfloor_h)$;
$r \leftarrow$ TKarMul$(h-1, \lceil a \rceil^h, \lceil c \rceil^{\rho+h} - [c]_h^{\rho+2h})$;
$s \leftarrow$ TKarMul$(n-h, \lfloor a \rfloor_h, \lfloor c \rfloor_{2h} - [c]_h^{m+n-h+1})$;
$t \leftarrow$ TKarMul$(h-1, \lceil a \rceil^h + \lfloor a \rfloor_h, [c]_h^{\lambda+2h})$;
return $r + t + x^h(s + \lceil t \rceil^{n-h+1})$;

Degenerate case 1: $\mu > n$. We let $\rho = \deg(\lceil a \rceil^{\mu})$ and consider $u \in R[x]_{\rho+n}$, $v \in R[x]_{m+n-\mu}$ so that we have $ab = N_1(u, v)$, where N_1 is:

$$N_1 = \left[\begin{array}{c|c} \frac{1_{\rho+n+1}}{0_{m-\rho,\rho+n+1}} & \frac{0_{\mu,r-\mu+1}}{1_{r-\mu+1}} \end{array} \right].$$

Degenerate case 2: $\nu > m$. We consider $u \in R[x]_{m+\nu-1}$, $v \in R[x]_{m+n-\nu}$ so that we have $ab = N_2(u, v)$, with

$$N_2 = \left[\begin{array}{c|c} \frac{1_{m+\nu}}{0_{n-\nu+1,m+\nu}} & \frac{0_{\nu,r-\nu+1}}{1_{r-\nu+1}} \end{array} \right].$$

Proposition 3. *Algorithm* TKarMul *is correct.*

Proof. We use the notation of Figure 4 and distinguish the same three cases. The normal case follows from the equalities

$$\begin{aligned} N_+^t c &= (\lceil c \rceil^{\rho+h}, \lfloor c \rfloor_{2h}, [c]_h^{\lambda+2h}), \\ N_-^t c &= ([c]_h^{\rho+2h}, [c]_h^{m+n-h+1}, 0). \end{aligned}$$

In Degenerate cases 1 and 2, we respectively compute

$$N_1^t c = (\lceil c \rceil^{n+\rho}, \lfloor c \rfloor_{\mu}) \text{ and } N_2^t c = (\lceil c \rceil^{m+\nu}, \lfloor c \rfloor_{\nu}).$$

□

Some refinements may be done when implementing Algorithm TKarMul. First observe that it saves memory to compute $\lceil c \rceil^{m+n-h+1} - \lfloor c \rfloor_h$ in order to deduce r and $-s$ and then one propagates this change of sign by returning $r + t + x^h(\lceil t \rceil^{n-h+1} - s)$. Another observation is that Karatsuba's multiplication can be done in a different manner using the identity $\lceil a \rceil^h \lfloor b \rfloor_h + \lfloor a \rfloor_h \lceil b \rceil^h = \lceil a \rceil^h \lceil b \rceil^h + \lfloor a \rfloor_h \lfloor b \rfloor_h - (\lceil a \rceil^h - \lfloor a \rfloor_h)(\lceil b \rceil^h - \lfloor b \rfloor_h)$. When transposing this slightly different version, we obtain the middle product algorithm presented in [10].

4.3 The Fast Fourier Transform

Multiplication algorithms using the Fast Fourier Transform are quite easy to transpose since the matrices involved are symmetric. We only give a brief presentation and refer to [9] for more details about the discrete Fourier transform.

Let $l \in \mathbb{N}$ such that $m + n + 1 \leq 2^l$ and that R contains a primitive 2^l-th root ω of unity. The discrete Fourier transform $\mathrm{DFT}(\omega, a)$ of the polynomial a is the vector

$$\left(a(1), a(\omega), \ldots, a(\omega^{2^l-1})\right) \in R^{2^l}.$$

Let $\mathrm{DFT}^{-1}(\omega, .) : R^{2^l} \to R[x]_{2^l-1}$ be the inverse map of $\mathrm{DFT}(\omega, .)$ and H the diagonal matrix of diagonal $\mathrm{DFT}(\omega, a)$. Then we have the equality

$$ab = \mathrm{DFT}^{-1}(\omega, H\,\mathrm{DFT}(\omega, b)).$$

Since $\mathrm{DFT}(\omega, .)$ and H are symmetric, we deduce that:

$$\mathrm{mul}^t(n, a, c) = \left\lceil \mathrm{DFT}(\omega, H\,\mathrm{DFT}^{-1}(\omega, c)) \right\rceil^{n+1},$$

for any polynomial c of degree at most $m+n$. Letting $\widetilde{H}$ be the diagonal $\mathrm{DFT}(\omega, \mathrm{rev}(m, a))$ matrix and using

$$\mathrm{DFT}^{-1}(\omega, .) = \frac{1}{2^l}\mathrm{DFT}(\omega^{-1}, .),$$

we deduce the equalities

$$\begin{aligned} \mathrm{mul}^t(n, a, c) &= \left\lceil \mathrm{DFT}^{-1}(\omega, \omega^{-m}\widetilde{H}\,\mathrm{DFT}(\omega, c)) \right\rceil^{n+1} \\ &= \left[\mathrm{DFT}^{-1}(\omega, \widetilde{H}\,\mathrm{DFT}(\omega, c)) \right]_m^{m+n+1}, \end{aligned}$$

which can also be obtained from the middle product formulation.

5. POLYNOMIAL DIVISION

We come now to the transposition of the Euclidean division. In this section we are given a polynomial $a \neq 0$ of degree m whose leading coefficient a_m is invertible. For a polynomial b, we write the division of b by a as $b = aq + r$ with $\deg(r) < m$ and define the maps

$$\begin{array}{rccc} \mathrm{quo}(a, .) : & R[x]_n & \to & R[x]_{n-m} \\ & b & \mapsto & q, \end{array}$$

$$\begin{array}{rccc} \mathrm{rem}(a, .) : & R[x]_n & \to & R[x]_{m-1} \\ & b & \mapsto & r, \end{array}$$

$$\begin{array}{rccc} \mathrm{quorem}(a, .) : & R[x]_n & \to & R[x]_{n-m} \times R[x]_{m-1} \\ & b & \mapsto & (q, r). \end{array}$$

The transposed operations are written $\mathrm{quorem}^t(n, a, q, r)$, $\mathrm{quo}^t(n, a, q)$ and $\mathrm{rem}^t(n, a, r)$, with the convention that these functions return an error if $\deg(r) \geq \deg(a)$ or $\deg(q) > n - \deg(a)$.

The next paragraphs are devoted to transpose the quorem map through the plain and Sieveking-Kung's division algorithms. We will prove that the transposed remainder is given by:

$$\begin{array}{rccc} \mathrm{rem}^t(n, a, .) : & R^m & \to & R^{n+1} \\ & (r_0, \ldots, r_{m-1}) & \mapsto & (b_0, \ldots, b_n), \end{array}$$

where the b_j are defined by the linear recurrence

$$(*) \quad b_j = -\frac{1}{a_m}\Big(a_{m-1}b_{j-1} + \cdots + a_0 b_{j-m}\Big), \quad m \leq j$$

with initial conditions $b_j = r_j, \quad j \in \{0, \ldots, m-1\}$, so that linear recurrence sequence extension is dual to remainder computation.

5.1 Plain division

We enter the plain division procedure with the two polynomials a and b and compute q and r by

$q \leftarrow 0$;
$r \leftarrow b$;
for i **from** 0 **to** $n-m$ **do**
  $q \leftarrow xq + r_{n-i}/a_m$;
  $r \leftarrow r - r_{n-i}/a_m x^{n-m-i} a$;

To transpose this algorithm we introduce the sequences

$$q^{\{i\}} \in R[x]_{i-1},\ r^{\{i\}} \in R[x]_{n-i} \quad i = 0, \ldots, n-m+1.$$

They are defined by $q^{\{0\}} = 0, r^{\{0\}} = b$ and for $i \geq 1$

$$\begin{aligned} q^{\{i+1\}} &= xq^{\{i\}} + r^{\{i\}}_{n-i}/a_m, \\ r^{\{i+1\}} &= r^{\{i\}} - r^{\{i\}}_{n-i}/a_m x^{n-m-i} a, \end{aligned}$$

so that the relation $b = ax^{n-m+1-i} q^{\{i\}} + r^{\{i\}}$ holds. For $i = n-m+1$, we have $q = q^{\{n-m+1\}}$ and $r = r^{\{n-m+1\}}$.

In order to formulate the algorithm in terms of linear maps we introduce $v^{\{i\}} = (q^{\{i\}}, r^{\{i\}}) \in R^{n+1}$. Then we have $v^{\{i+1\}} = Mv^{\{i\}}$, where

$$M = \left[\begin{array}{c|c} 0_{1,n} & 1/a_m \\ \hline & 0_{n-m,1} \\ & -a_0/a_m \\ 1_n & \vdots \\ & -a_{m-1}/a_m \end{array}\right].$$

Now reverse the flow of calculation: we start with a vector $v^{\{n-m+1\}} = (q^{\{n-m+1\}}, r^{\{n-m+1\}}) \in R^{n+1}$ with $q^{\{n-m+1\}} \in R[x]_{n-m}$ and $r^{\{n-m+1\}} \in R[x]_{m-1}$. Then we compute $v^{\{i\}} = M^t v^{\{i+1\}}$ by the formulae

$$\begin{aligned} q^{\{i\}} &= \left\lfloor q^{\{i+1\}} \right\rfloor_1, \\ r^{\{i\}} &= r^{\{i+1\}} + \frac{1}{a_m} x^{n-i} \left(q_0^{\{i+1\}} - \sum_{j=0}^{m-1} a_{m-1-j} r^{\{i+1\}}_{n-i-j-1} \right). \end{aligned}$$

We deduce the following transposed algorithm for computing $b = \mathrm{quorem}^t(n, a, q, r)$:

$b \leftarrow r$;
for i **from** m **to** n **do**
  $b_i \leftarrow \left(q_{i-m} - \sum_{j=0}^{m-1} a_{m-1-j} b_{i-j-1} \right) / a_m$;

The maps $\mathrm{quo}(a, .)$ and $\mathrm{rem}(a, .)$ can be obtained by composing a projection after $\mathrm{quorem}(a, .)$ but in practice it is better to implement specific optimized procedures for each. It is easy to implement these optimizations; we refer to our NTL implementation for details.

Since $b = \mathrm{rem}^t(n, a, r) = \mathrm{quorem}^t(n, a, 0, r)$, the coefficients of b satisfy the linear recurrence relation with constant coefficients $(*)$, as announced above.

5.2 Sieveking-Kung's division

Sieveking-Kung's algorithm is based on the formula

$$\mathrm{rev}(n, b) = \mathrm{rev}(n-m, q)\mathrm{rev}(m, a) + x^{n-m+1}\mathrm{rev}(m-1, r),$$

see [9] for details. This yields

$$q = \mathrm{rev}(n-m, \lceil \mathrm{mul}(\alpha, \lceil \mathrm{rev}(n, b) \rceil^{n-m+1}) \rceil^{n-m+1}),$$

where $\alpha = \lceil \mathrm{rev}(m, a)^{-1} \rceil^{n-m+1}$. Then we deduce r from q using $r = b - aq$.

Transposing these equalities, we see that for $q \in R[x]_{n-m}$

$$\mathrm{quo}^t(n, a, q) = \mathrm{rev}(n, \mathrm{mul}^t(n-m, \alpha, \mathrm{rev}(n-m, q))).$$

For $r \in R[x]_{m-1}$, it follows:

$$\mathrm{rem}^t(n, a, r) = r - \mathrm{quo}^t(n, a, \mathrm{mul}^t(n-m, a, r)).$$

Let $s = \mathrm{rem}^t(n, a, r)$ and $p = \mathrm{mul}^t(n-m, a, r)$. Using the middle product formula to express this last quo^t expression yields

$$s = r - \mathrm{rev}(n, [\mathrm{rev}(n-m, \alpha)\mathrm{rev}(n-m, p)]_{n-m}^{2(n-m)+1}),$$

which simplifies to

$$s = r - \mathrm{rev}(n, \mathrm{rev}(n-m, \lceil \alpha p \rceil^{n-m+1})).$$

Last we obtain

$$\mathrm{rem}^t(n, a, r) = r - x^m \left\lceil \alpha \mathrm{mul}^t(n-m, a, r) \right\rceil^{n-m+1},$$

which can also be rewritten this way, using the middle product formula again:

$$\mathrm{rem}^t(n, a, r) = r - x^m \left\lceil \alpha \left[\mathrm{rev}(m, a) r \right]_m^{n+1} \right\rceil^{n-m+1}.$$

This actually coincides with the algorithm given in [21] for extending linear recurrence sequences.

Remark. More generally, the following formula holds:

$$\mathrm{quorem}^t(n, a, q, r) = r - x^m \left\lceil \alpha ([\mathrm{rev}(m, a) r]_m^{n+1} - q) \right\rceil^{n-m+1}.$$

5.3 Modular multiplication

As a byproduct, our algorithms enable to transpose the *modular multiplication*. Given a monic polynomial a of degree m with invertible leading coefficient and a polynomial b of degree at most $m-1$, consider the composed map

$$\begin{array}{ccccc} R[x]_{m-1} & \to & R[x]_{2m-2} & \to & R[x]_{m-1} \\ c & \mapsto & bc & \mapsto & bc \mod a. \end{array}$$

An ad hoc algorithm for the transpose map is detailed in [23]. Using our remark on the transpose of polynomial remaindering, it is seen to essentially coincide with the one obtained by composing the algorithms for transposed multiplication and transposed remainder. A constant factor is lost in [23], as no middle product algorithm is used to transpose the first map; this was already pointed out in [10].

6. TRANSPOSED VANDERMONDE

We now focus on algorithms for Vandermonde matrices; we assume that $R = k$ is a field and consider $m+1$ pairwise distinct elements $a_0, \ldots, a_m$ in k. Even if not required by the algorithms, we take $m = 2^l - 1$, with $l \in \mathbb{N}$, in order to simplify the complexity estimates.

The Vandermonde matrix V_a is the square matrix:

$$V_a = \begin{bmatrix} 1 & a_0 & a_0^2 & \ldots & a_0^m \\ 1 & a_1 & a_1^2 & \ldots & a_1^m \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & a_m & a_m^2 & \ldots & a_m^m \end{bmatrix}.$$

If b is in $k[x]_m$ then $V_a b$ is the vector $(b(a_0), \ldots, b(a_m))$. This computation is commonly referred to as *multipoint evaluation*. The inverse problem is *interpolation*: for $c \in k^{m+1}$, $V_a^{-1}c$ corresponds to the polynomial $b \in k[x]_m$ satisfying $b(a_i) = c_i$ for $i \in \{0, \ldots, m\}$.

We first recall the notion of *subproduct tree* and refer to [9, §10.1] for details and historical notes. Then we design a fast algorithm for transposed evaluation. We transpose it backwards to obtain an improved evaluation algorithm, and deduce similar improvements for interpolation and its transpose.

For complexity analysis we use the straight-line program model [5, Chapter 4]. The function $M(n)$ denotes the complexity of multiplying a polynomial of degree less than n by a fixed polynomial of degree less than n, restricting to linear straight-line programs only. This way Proposition 1 implies that the complexity of the transpose of this multiplication is at most $M(n) + n$ operations in k. Restricting to linear straight-line programs is actually not embarrassing since all known multiplication algorithms [9] for the straight-line program model fit into this setting.

As in [9, §8.3] we assume that $M(n_1 + n_2) \geq M(n_1) + M(n_2)$ for any positive integers n_1 and n_2. For the sake of simplicity we also assume that $n\log(n) \in O(M(n))$.

6.1 Going up the subproduct tree

A common piece of the following algorithms is the computation of the *subproduct tree* T with leaves $x - a_0, x - a_1, \ldots, x - a_m$. It is defined recursively, together with the sequence of integers m_i, by

$$T_{0,j} = x - a_j, \text{ for } j \in \{0, \ldots, m\}, \quad m_0 = m + 1,$$

and for $i \geq 1$ by

$$T_{i,j} = T_{i-1,\lfloor j/2 \rfloor} T_{i-1,\lfloor j/2 \rfloor + 1}, \text{ for } j < h_i = \lfloor m_i/2 \rfloor .$$

If $m_i = 2h_i + 1$ we let $T_{i,h_i} = T_{i-1,m_i-1}$ and $m_{i+1} = h_i + 1$, otherwise we just let $m_{i+1} = h_i$. Let d be the smallest integer such that $m_d = 1$; we get $T_{d,0} = \prod_{j=0}^{m}(x-a_j)$. By [9, Exercise 10.3], T can be computed within $1/2M(m)\log(m) + O(m\log(m))$ operations.

The following algorithm describes our basic use of T. On input $c = (c_0, \ldots, c_m)$, it computes the polynomial $b = \sum_{j=0}^{m} c_j \frac{T_{d,0}}{x-a_j}$.

UpTree(c)
  $b \leftarrow c$;
  for $i \leftarrow 0$ **to** $d-1$ **do**
    for $j \leftarrow 0$ **to** $h_i - 1$ **do**
      $b_j \leftarrow T_{i,2j+1}b_{2j} + T_{i,2j}b_{2j+1}$;
    if $m_i = 2h_i + 1$ **then** $b_{h_i} \leftarrow b_{m_i-1}$;
  return b_0;

We deduce its transpose as follows:

TUpTree(b)
  $c_0 \leftarrow b$;
  for $i \leftarrow d-1$ **downto** 0 **do**
    if $m_i = 2h_i + 1$ **then** $c_{m_i-1} \leftarrow c_{h_i}$;
    for $j \leftarrow h_i - 1$ **downto** 0 **do**
      $n \leftarrow \deg(T_{i+1,j}) - 1$;
      $c_{2j+1} \leftarrow \mathrm{mul}^t(n, T_{i,2j}, c_j)$;
      $c_{2j} \leftarrow \mathrm{mul}^t(n, T_{i,2j+1}, c_j)$;
  return c;

On input $b \in k[x]_m$, it outputs the coefficients of x^m in the polynomial products $\mathrm{rev}(m, b)\frac{T_{d,0}}{x-a_j}$, for $j = 0, \ldots, m$.

Once T is computed, using Proposition 1 and [9, Th. 10.10], both algorithms require $M(m)\log(m) + O(m\log(m))$ operations.

6.2 Multipoint evaluation

We first treat the transposed problem. Let $c_0, \ldots, c_m$ be in k. A direct computation shows that the entries of $b = V_a^t c$ are the first $m+1$ coefficients of the Taylor expansion of $S(x) =$

$$\sum_{j=0}^{m} \frac{c_j}{1 - a_j x} = \frac{1}{\mathrm{rev}(m+1, T_{d,0})} \sum_{j=0}^{m} \frac{c_j \mathrm{rev}(m+1, T_{d,0})}{1 - a_j x}.$$

The last sum is obtained by computing UpTree(c) and reversing the result. Computing the Taylor expansion of S requires one additional power series inversion and one multiplication.

$\alpha \leftarrow 1/\mathrm{rev}(m+1, T_{d,0}) \mod x^{m+1}$;
$s \leftarrow \mathrm{UpTree}(c)$;
$t \leftarrow \mathrm{rev}(m, s)$;
$b \leftarrow \mathrm{mul}(\alpha, t) \mod x^{m+1}$;

We deduce the following algorithm for evaluating a polynomial b:

$\alpha \leftarrow 1/\mathrm{rev}(m+1, T_{d,0}) \mod x^{m+1}$;
$t \leftarrow \mathrm{mul}^t(m, \alpha, b)$;
$s \leftarrow \mathrm{rev}(m, t)$;
$c \leftarrow \mathrm{TUpTree}(s)$;

By the above complexity results, these algorithms require $3/2M(m)\log(m) + O(M(m))$ operations, since the additional operations have negligible cost. We gain a constant factor on the usual algorithm of repeated remaindering, of complexity $7/2M(m)\log(m) + O(M(m))$, see [18] or [9, Exercise 10.9]. We also gain on $13/6M(m)\log(m) + O(M(m))$ given in [18, §3.7] for base fields k allowing Fast Fourier Transform in $k[x]$.

Moreover, if many evaluations at the same set of points a_i have to be performed, then all data depending only on the evaluation points (the tree T and α) may be precomputed and stored, and the cost of evaluation drops to essentially $M(m)\log(m) + O(M(m))$. This improves [9, Exercise 10.11] by a factor of 2.

As a remark, our interpretation of the transposed remainder shows that the algorithm in [21] for transposed evaluation is the exact transposition of the classical evaluation algorithm as given in [9, §10.1], so ours is faster. The algorithm of [6] is still slower by a constant factor, since it uses an interpolation routine.

6.3 Interpolation

The fast interpolation algorithm $c = V_a^{-1}b$ proceeds this way, see [9, §10.2]:

$p \leftarrow dT_{d,0}/dx$;
$z \leftarrow (p(a_0), \ldots, p(a_m))$;
$c \leftarrow (b_0/z_0, \ldots, b_m/z_m)$;
$c \leftarrow \mathrm{UpTree}(c)$;

Here, z is computed using fast multipoint evaluation. At the end, c contains the interpolating polynomial. Reversing the computation flow, we get the transposed algorithm for computing $b = (V_a^{-1})^t c$:

$p \leftarrow dT_{d,0}/dx$;
$z \leftarrow (p(a_0), \dots, p(a_m))$;
$b \leftarrow \mathrm{TUpTree}(c)$;
$b \leftarrow (b_0/z_0, \dots, b_m/z_m)$;

Using the results given above, these algorithms require $5/2M(m)\log(m) + O(M(m))$ operations. This improves again the complexity results of [9].

The algorithm of [15] for transposed interpolation does the same precomputation as ours, but the call to TUpTree is replaced by (mainly) a multipoint evaluation. Using precomputations and our result on evaluation, that algorithm also has complexity $5/2M(m)\log(m) + O(M(m))$.

7. CONCLUSION, FUTURE WORK

A first implementation of our improved evaluation algorithm gains a factor of about 1.2 to 1.5 over the classical one [9, §10.1], for degrees about 10000. But let us mention that the crossover point between repeating Horner's rule and the classical fast evaluation we have implemented is about 32. In consequence, it seems interesting to explore the constant factors hidden behind the above quantities $O(M(m))$.

Our results of Section 6 can be generalized to solve the *simultaneous modular reduction* and the *Chinese remainder* problems faster than in [9, Chapter 10]. Theoretical and practical developments related to these problems are work in progress.

Transposed computations with Vandermonde matrices are the basis of fast multiplication algorithms for sparse and dense multivariate polynomials [2, 6, 25] and power series [11, 17]. Up to logarithmic factors, over a base field of characteristic zero, these multiplications have linear complexities in the size of the output. Their implementation is the subject of future work.

Acknowledgments. We thank G. Hanrot, M. Quercia and P. Zimmermann for useful discussions and the anonymous referees for their constructive remarks.

8. REFERENCES

[1] A. Antoniou. *Digital Filters: Analysis and Design.* McGraw-Hill Book Co., 1979.
[2] M. Ben-Or and P. Tiwari. A deterministic algorithm for sparse multivariate polynomial interpolation. In *20th Annual ACM Symposium on the Theory of Computing*, pages 301–309, 1988.
[3] J. L. Bordewijk. Inter-reciprocity applied to electrical networks. *Appl. Sci. Res. B.*, 6:1–74, 1956.
[4] A. Bostan, B. Salvy, and É. Schost. Fast Algorithms for Zero-Dimensional Polynomial Systems Using Duality. *Applicable Algebra in Engineering, Communication and Computing*. To appear.
[5] P. Bürgisser, M. Clausen, and A. Shokrollahi. *Algebraic Complexity Theory.* Springer, 1997.
[6] J. Canny, E. Kaltofen, and L. Yagati. Solving systems of non-linear polynomial equations faster. In *Proceedings of ISSAC'89*, pages 121–128. ACM, 1989.
[7] C. M. Fiduccia. On obtaining upper bounds on the complexity of matrix multiplication. In *Complexity of computer computations (Proc. Sympos., IBM Thomas J. Watson Res. Center, Yorktown Heights, N.Y., 1972)*, pages 31–40. Plenum, New York, 1972.
[8] C. M. Fiduccia. *On the algebraic complexity of matrix multiplication.* PhD thesis, Brown Univ., Providence, RI, Center Comput. Inform. Sci., Div. Engin., 1973.
[9] J. von zur Gathen and J. Gerhard. *Modern computer algebra.* Cambridge University Press, 1999.
[10] G. Hanrot, M. Quercia, and P. Zimmermann. The middle product algorithm, I. Speeding up the division and square root of power series. Technical report, RR INRIA 4664, 2002.
[11] J. van der Hoeven. Relax, but don't be too lazy. *Journal of Symbolic Computation*, 34(9):479 – 542, 2002.
[12] J. Hopcroft and J. Musinski. Duality applied to the complexity of matrix multiplication and other bilinear forms. *SIAM Journal on Computing*, 2:159–173, 1973.
[13] E. Kaltofen. Challenges of symbolic computation: my favorite open problems. *Journal of Symbolic Computation*, 29(6):891–919, 2000.
[14] E. Kaltofen and V. Shoup. Subquadratic-time factoring of polynomials over finite fields. *Mathematics of Computation*, 67(223):1179–1197, 1998.
[15] E. Kaltofen and L. Yagati. Improved sparse multivariate polynomial interpolation algorithms. In P. Gianni, editor, *Proceedings of ISSAC'88*, volume 358 of *LNCS*, pages 467–474. Springer Verlag, 1989.
[16] M. Kaminski, D. Kirkpatrick, and N. Bshouty. Addition requirements for matrix and transposed matrix products. *Journal of Algorithms*, 9(3):354–364, 1988.
[17] G. Lecerf and É. Schost. Fast multivariate power series multiplication in characteristic zero. *SADIO Electronic Journal on Informatics and Operations Research.* To appear, manuscript of December 2002.
[18] P. L. Montgomery. *An FFT extension of the elliptic curve method of factorization.* PhD thesis, University of California, Los Angeles CA, 1992.
[19] P. Penfield, Jr., R. Spencer, and S. Duinker. *Tellegen's theorem and electrical networks.* The M.I.T. Press, Cambridge, Mass.-London, 1970.
[20] V. Shoup. NTL: A library for doing number theory. `http://www.shoup.net`.
[21] V. Shoup. A fast deterministic algorithm for factoring polynomials over finite fields of small characteristic. In *Proceedings of ISSAC'91*, pages 14–21. ACM, 1991.
[22] V. Shoup. A new polynomial factorization algorithm and its implementation. *Journal of Symbolic Computation*, 20(4):363–397, 1995.
[23] V. Shoup. Efficient computation of minimal polynomials in algebraic extensions of finite fields. In *Proceedings of ISSAC'99*, pages 53–58. ACM, 1999.
[24] B. Tellegen. A general network theorem, with applications. *Philips Research Reports*, 7:259–269, 1952.
[25] R. Zippel. Interpolating polynomials from their values. *Journal of Symbolic Computation*, 9(3):375–403, 1990.