Faster Chinese remaindering

Joris van der Hoeven

Laboratoire d'informatique, UMR 7161 CNRS
Campus de l'École polytechnique
1, rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing, CS35003
91120 Palaiseau

November 27, 2016

The Chinese remainder theorem is a key tool for the design of efficient multi-modular algorithms. In this paper, we study the case when the moduli $m_1, ..., m_\ell$ are fixed and can even be chosen by the user. Through an appropriate use of the technique of FFT-trading, we will show that this assumption allows for the gain of an asymptotic factor $O(\log\log\ell)$ in the complexity of "Chinese remaindering". For small ℓ, we will also show how to choose "gentle moduli" that allow for further gains at the other end. The multiplication of integer matrices is one typical application where we expect practical gains for various common matrix dimensions and integer bitsizes.

Keywords: Chinese remainder theorem, algorithm, complexity, integer matrix multiplication

1. Introduction

Modular reduction is an important tool in computer algebra and elsewhere for speeding up computations. The technique allows to reduce a problem that involves large integer or polynomial coefficients to one or more similar problems that only involve small modular coefficients. Depending on the application, the solution to the initial problem is reconstructed *via* the Chinese remainder theorem or Hensel's lemma. We refer to [10, chapter 5] for a gentle introduction to this topic.

In this paper, we will mainly be concerned with multi-modular algorithms over the integers that rely on the Chinese remainder theorem. The archetype of such an algorithm works as follows. We start with a polynomial function $f: \mathbb{Z}^r \to \mathbb{Z}^s$. For any modulus m, reduction of f modulo m yields a new function $f_m: (\mathbb{Z}/m\,\mathbb{Z})^r \to (\mathbb{Z}/m\,\mathbb{Z})^s$ such that

$$f(x_1, ..., x_r) \bmod m \;=\; f_m(x_1 \bmod m, ..., x_r \bmod m)$$

for all $x_1, ..., x_r \in \mathbb{Z}$. Given an algorithm to compute f that only uses ring operations on integers, it suffices to replace each ring operations by its reduction modulo m in order to obtain an algorithm that computes f_m. Now given integers $x_1, ..., x_r \in \mathbb{Z}$ and $(y_1, ..., y_s) = f(x_1, ..., x_r)$, assume that we know a bound $B \in \mathbb{N}$ with $|x_i| \leqslant B$ for $i = 1, ..., r$ and $|y_i| \leqslant B$ for $i = 1, ..., s$. Then the following multi-modular algorithm provides with an alternative way to compute $f(x_1, ..., x_r)$:

0. Select moduli $m_1, ..., m_\ell$ with $m_1 \cdots m_\ell > 2\,B$ that are mutually coprime.
1. For $i = 1, ..., r$, compute $x_{i,j} := x_i \bmod m_j$ for $j = 1, ..., \ell$.
2. For $j = 1, ..., \ell$, compute $(y_{1,j}, ..., y_{s,j}) := f_{m_j}(x_{1,j}, ..., x_{r,j})$.

3. For $i = 1, ..., s$, reconstruct y_i from the values $y_{i,j} := y_i \bmod m_j$ with $j = 1, ..., \ell$.

Step 1 consists of r *multi-modular reductions* (finding the $x_{i,j}$ as a function of x_i) and step 3 of s *multi-modular reconstructions* (finding y_i as a function the $y_{i,j}$); this is where the Chinese remainder theorem comes in. For a more detailed example with an application to integer matrix multiplication, we refer to section 4.5.

In favourable cases, the cost of steps 0, 1 and 3 is negligible with respect to the cost of step 2. In such situations, the multi-modular algorithm to compute f is usually much faster than the original algorithm. In less favourable cases, the cost of steps 1 and 3 can no longer be neglected. This raises the question whether it is possible to reduce the cost of these steps as much as possible.

Two observations are crucial here. First of all, the moduli $m_1, ..., m_\ell$ are the same for all r multi-modular reductions and s multi-modular reconstructions. If $r + s$ is large, then this means that we can essentially assume that $m_1, ..., m_\ell$ were fixed once and for all. Secondly, we are free to choose $m_1, ..., m_\ell$ in any way that suits us. By making each m_i fit into a machine word, one may ensure that every modular operation only takes a few cycles. Special "FFT-moduli" are often used as well for speeding up polynomial arithmetic.

In this paper, we will show how to exploit both of the above observations. For fixed moduli, we will show in section 3 how to make Chinese remaindering asymptotically more efficient by a factor $O(\log\log\ell)$ when ℓ gets large. In section 4, we show that it is possible to construct "gentle modulo" that allow for speed-ups when ℓ is small ($l \lesssim 64$). Both results can be combined in order to accelerate Chinese remaindering for all possible values of ℓ.

The new asymptotic complexity bounds make heavy use of discrete Fourier transforms. For our purposes, it is crucial to avoid "synthetic" FFT schemes that require the adjunction of artificial roots of unity as in Schönhage–Strassen multiplication [24]. Instead, one should use "inborn" FFT schemes that work with approximate roots of unity in $\mathbb{C}$ or roots of unity with high smooth orders in finite fields; see [24, section 3] and [23, 15]. Basic complexity properties of integer multiplication and division based on fast Fourier techniques are recalled in section 2.

Let $\mathsf{I}(n)$ be the bit complexity for multiplying two n-bit numbers. Given pairwise comprime moduli $m_1, ..., m_\ell$ of bit-size n, it is well known that multi-modular reduction and reconstruction can be carried out in time $O(\mathsf{I}(n\,\ell)\log\ell)$ using so called *remainder trees* [8, 20, 3]. Recent improvements of this technique can be found in [4, 2]. The main goal of section 3 is to show that this complexity essentially drops down to $O(\mathsf{I}(n\,\ell)\log\ell/\log\log\ell)$ in the case when all moduli $m_1, ..., m_\ell$ are fixed; see Theorems 6 and 10 for more precise statements. The main idea is to increase the arities of nodes in the remainder tree, while performing the bulk of the computations at each node using Fourier representations. This technique of trading faster algorithms against faster representations was also used in [16], where we called it *FFT-trading*; see also [1]. The same approach can also be applied to the problem of base conversion (see section 3.8) and for univariate polynomials instead of integers (see section 3.9).

Having obtained a non trivial asymptotic speed-up for large ℓ, we next turn our attention to the case when ℓ is small (say $\ell \lesssim 64$). The main goal of section 4 there is to exhibit the existence of *gentle moduli* $m_1, ..., m_\ell$ for which Chinese remaindering becomes more efficient than usual. The first idea is to pick moduli m_i of the form $2^{sw} - \varepsilon_i^2$, where w is somewhat smaller than the hardware word size, s is even, and $\varepsilon_i^2 < 2^w$. In section 4.1, we will show that multi-modular reduction and reconstruction both become a lot simpler for such moduli. Secondly, each m_i can be factored as $m_i = (2^{sw/2} - \varepsilon_i)\,(2^{sw/2} + \varepsilon_i)$ and, if we are lucky, then both $2^{sw/2} - \varepsilon_i$ and $2^{sw/2} + \varepsilon_i$ can be factored into $s/2$ moduli that fit into

machine words. If we are very lucky, then this allows us to obtain $w\,\ell$ moduli $m_{i,j}$ of bitsize $\approx w$ that are mutually coprime and for which Chinese remaindering can be implemented efficiently. Gentle moduli can be regarded as the integer analogue of "special sets of points" that allowed for speed-ups of multi-point evaluation and interpolation in [5].

Acknowledgments. We would like to thank Grégoire Lecerf for pointing us to Bernstein's work [2] on the topic of this paper.

2. Preliminaries

2.1. Integer multiplication

Throughout this paper we will assuming the deterministic multitape Turing model [21] in order to analyze the "bit complexity" of our algorithms. We will denote by $\mathsf{I}(n)$ the cost of n-bit integer multiplication. The best current bound [15] for $\mathsf{I}(n)$ is

$$\mathsf{I}(n) \;=\; O(n \log n\, 8^{\log^* n}),$$

where $\log^* n := \min \left\{ k \in \mathbb{N} : \left(\log \circ \overset{k\times}{\ldots} \circ \log\right)(n) \leqslant 1 \right\}$ is called the iterator of the logarithm.

For large n, it is well known that the fastest algorithms for integer multiplication [23, 24, 9, 15] are all based on the discrete Fourier transform [7]: denoting by $\mathsf{F}(2\,n)$ the cost of a "suitable Fourier transform" of bitsize $2\,n$ and by $\mathsf{N}(2\,n)$ the cost of the "inner multiplications" for this bitsize, one has

$$\mathsf{I}(n) \;=\; 3\,\mathsf{F}(2\,n) + \mathsf{N}(2\,n). \tag{1}$$

For the best current algorithm from [15], we have

$$\mathsf{F}(2\,n) \;=\; O(n \log n\, 8^{\log^* n}) \tag{2}$$
$$\mathsf{N}(2\,n) \;=\; O(n\, 4^{\log^* n}). \tag{3}$$

One always has $\mathsf{N}(2\;n) = o(\mathsf{F}(2\;n))$. The actual size of Fourier transforms is usually somewhat restricted: for efficiency reasons, it should be the product of powers of small prime numbers only, such as 2, 3 and 5. Fortunately, for large numbers n, it is always possible to find $n' \in 2^{\mathbb{N}}\, 3^{\mathbb{N}}\, 5^{\mathbb{N}}$ with $n' \geqslant n$ and $n'/n = 1 + o(1)$.

It is also well known that fast Fourier transforms allow for several tricks. For instance, if one of the multiplicands of an n-bit integer product is fixed, then its Fourier transform can be precomputed. This means that the cost of the multiplication drops down to

$$\mathsf{I}_{\text{fixed}}(n) \;=\; 2\,\mathsf{F}(2\,n) + \mathsf{N}(2\,n) \;\sim\; {}^2\!/_3\,\mathsf{I}(n).$$

In particular, the complexity $\mathsf{I}(N, n)$ of multiplying an N-bit integer with an n-bit one (for $N \geqslant n$) satisfies

$$\mathsf{I}(N, n) \;=\; \left({}^2\!/_3\,\frac{N}{n} + {}^1\!/_3 + o(1)\right) \mathsf{I}(n).$$

Squares of n-bit numbers can be computed in time $(2 + o(1))\,\mathsf{F}(2\,n) \sim {}^2\!/_3\,\mathsf{I}(n)$ for the same reason. Yet another example is the multiplication of two 2×2 matrices with $(n-1)$-bit integer entries: such multiplications can be done in time $(12 + o(1))\;\mathsf{F}(2\;n) \sim 4\;\mathsf{I}(n)$ by transforming the input matrices, multiplying the transformed matrices in the "Fourier model", and then transforming the result back.

In the remainder of this paper, we will systematically assume that asymptotically fast integer multiplication is based on fast Fourier transforms. In particular, we have (1) for certain functions F and N. We will also assume that the functions $\mathsf{F}(n)/(\mathsf{N}(n)\log n)$ and $\mathsf{N}(n)/n$ are (not necessarily strictly) increasing and that $\mathsf{F}(n) = o(\mathsf{N}(n)\log n \log\log n)$. These additional conditions are satisfied for (2) and (3). The first condition is violated whenever the FFT scheme requires the adjunction of artificial roots of unity. This happens for Schönhage–Strassen multiplication, in which case we have $\mathsf{F}(2\,n) = O(n\log n\log\log n)$ and $\mathsf{N}(2\,n) = O(n\log n)$). We will say that an FFT-scheme is *inborn* if it satisfies our additional requirements.

For a hypothetical integer multiplication that runs in time $\mathsf{I}(n) = o(n\log n\, 8^{\log^* n})$, but for which $\mathsf{I}(n)/(n\log n)$ is (not necessarily strictly) increasing, we also notice that it is possible to design an inborn FFT-based integer multiplication method that runs in time $O(\mathsf{I}(n))$; this is for instance used in [13].

2.2. Euclidean division of integers

Let $\mathsf{D}(N, n)$ denote the cost of euclidean division with remainder of an N-bit integer by an n-bit one. In [16, section 3.2], we gave an algorithm `divide` for the euclidean division of a polynomial of degree $<2\,n$ by another polynomial of degree $<n$. This algorithm is based on *FFT trading*, a technique that consists of doing as much work as possible in the FFT-model even at the expense of using naive, suboptimal algorithms in the FFT-model.

The straightforward adaptation of this division to integer arithmetic yields the asymptotic bound

$$\mathsf{D}(2\,n, n) \;\leqslant\; (5/3 + o(1))\,\mathsf{I}(n).$$

Furthermore, the discrete Fourier transforms for the dividend make up for roughly one fifth of the total amount of transforms. For $N \geqslant 2\,n$, the cost of the transforms for the dividend does not grow with N, which leads to the refinement

$$\mathsf{D}(N, n) \;\leqslant\; \left(4/3\,\frac{N}{n} - 1 + o(1)\right)\mathsf{I}(n) \qquad (N \geqslant 2\,n).$$

Similarly, if $n = o(N)$, then

$$\mathsf{D}(N, N-n) \;\leqslant\; (2/3 + o(1))\,\frac{N}{n}\,\mathsf{I}(n) \qquad (n = o(N)),$$

since the bulk of the computation consists of multiplying the approximate n-bit quotient with the $(N-n)$-bit dividend. If the dividend is fixed, then we even get

$$\mathsf{D}_{\text{fixed}}(N, N-n) \;\leqslant\; (1/3 + o(1))\,\frac{N}{n}\,\mathsf{I}(n) \qquad (n = o(N)),$$

since the Fourier transforms for the dividend can be precomputed.

2.3. Approximate products modulo one

Let us start with a few definitions and notations. Given $n \in \mathbb{N}$ and $e \in \mathbb{Z}$, we define

$$\mathbb{D}_{n,e} \;=\; \{k\,2^{e-n} : 0 \leqslant k < 2^n\}$$

be the set of dyadic fixed point numbers of bitsize n and with exponent e. Given $x \in \mathbb{R}$ and $m \in \mathbb{R}^>$, we denote by

$$x \operatorname{rem} m \;=\; x - \left\lfloor \frac{x}{m} \right\rfloor m \;\in\; [0, m)$$

the remainder of the euclidean division of x by m. Given $x \in \mathbb{R}$ and $\varepsilon \in \mathbb{R}^>$, we say that $\tilde{x} \in \mathbb{R}$ is an *ε-approximation* of x if $|\tilde{x} - x| < \varepsilon$. We also say that $\tilde{x}$ is a *circular ε-approximation* of x if $|\tilde{x} - x|_\circ < \varepsilon$. Here $|\tilde{x} - x|_\circ := \min_{k \in \mathbb{Z}} |\tilde{x} - x - k|$ denotes the *circular distance* between $\tilde{x}$ and x.

Let $x = k\, 2^{-2n} \in \mathbb{D}_{2n,-2n}$, $y = l\, 2^0 \in \mathbb{D}_{n,0}$ and $z = x\, y \operatorname{rem} 1 \in \mathbb{D}_{2n,-2n}$. *Mutatis mutandis*, Bernstein observed in [2] that we may compute a circular 2^{-n}-approximation $\tilde{z} \in \mathbb{D}_{n,-n}$ for z as follows. We first compute the product $m = k\, l \operatorname{rem} (2^{2n} - 1)$ of k and l modulo $2^{2n} - 1$ and then take $\tilde{z} = \lfloor 2^{-n}\, m \rfloor\, 2^{-n}$.

Let us show that $\tilde{z}$ indeed satisfies the required property. By construction, there exists an $a \in \mathbb{N}$ with $a < 2^n$ such that $m = k\, l - a\, (2^{2n} - 1)$. Therefore, $0 \leqslant m\, 2^{-2n} - \tilde{z} < 2^{-n}$ and $x\, y - m\, 2^{-2n} = a\, (2^{2n} - 1)\, 2^{-2n} = a - a\, 2^{-2n}$, whence $a - 2^{-n} < x\, y - \tilde{z} < a + 2^{-n}$.

More generally, if we only have a circular $[\tau\, 2^{-n}]$-approximation $\tilde{x} = k\, 2^{-2n} \in \mathbb{D}_{2n,-2n}$ of a number $x \in \mathbb{R}$ (instead of a number $x \in k\, 2^{-n} \in \mathbb{D}_{2n,-2n}$ as above), then the algorithm computes a circular $[(1 + y\, 2^{-n}\, \tau)\, 2^{-n}]$-approximation $\tilde{z}$ of $x\, y \operatorname{rem} 1$.

Bernstein's trick is illustrated in Figure 1: we are only interested in the highlighted portion of the product. We notice that these kinds of truncated products are reminiscent of the "middle product" in the case of polynomials [12]; in our setting, we also have to cope with carries and rounding errors. When using FFT-multiplication, products modulo $2^{2n} - 1$ can be computed using three discrete Fourier transforms of bitsize $2\, n$, so the cost is essentially the same as the cost of an n-bit integer multiplication. If one of the arguments is fixed, then the cost becomes asymptotic to $\frac{2}{3}\, \mathsf{I}(n)$.

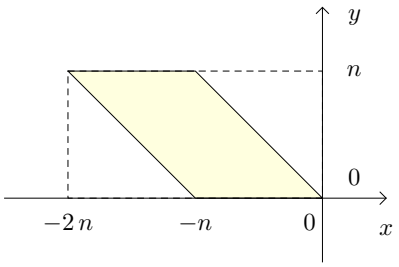

Figure 1. Product modulo one of $x \in \mathbb{D}_{2n,-2n}$ and $y \in \mathbb{D}_{n,0}$ with n significant bits.

More generally, for $\ell \in \{1, 2, \ldots\}$, let $\tilde{x} \in \mathbb{D}_{\ell n,-\ell n}$ be a circular $\tau\, 2^{-\ell n}$-approximation of a number $x \in \mathbb{R}$ and let $y \in \mathbb{D}_{(\ell-1)n,0}$. Then we may compute a circular approximation $\tilde{z} \in \mathbb{D}_{n,-n}$ of $z = x\, y \operatorname{rem} 1$ as follows. Consider the expansions

$$\tilde{x} = \sum_{i=-\ell}^{-1} \tilde{x}_i\, 2^{in}, \qquad y = \sum_{i=0}^{\ell-2} y_i\, 2^{in}, \tag{4}$$

with $\tilde{x}_{-\ell}, \ldots, \tilde{x}_{-1}, y_0, \ldots, y_{\ell-2} \in \mathbb{D}_{n,0}$; see Figure 2. By what precedes, for $i = 0, \ldots, \ell - 2$, we may compute circular 2^{-n}-approximations $\tilde{u}_i$ for

$$u_i \;=\; [(\tilde{x}_{-i-1}\, 2^{-2n} + \tilde{x}_{-i}\, 2^{-n})\, y_i] \operatorname{rem} 1.$$

Setting

$$v_i \;=\; [(\tilde{x}_{-\ell}\, 2^{-(\ell+1-i)n} + \cdots + \tilde{x}_{-i}\, 2^{-n})\, y_i] \operatorname{rem} 1,$$

it follows that $|\tilde{u}_i - v_i|_\circ < 2 \cdot 2^{-n}$, whereas $v := \tilde{x}\, y \operatorname{rem} 1 = (v_0 + \cdots + v_{\ell-2}) \operatorname{rem} 1$. Taking $\tilde{z} = (u_0 + \cdots + u_{\ell-2}) \operatorname{rem} 1$, it follows that $|\tilde{z} - v|_\circ < (2\, \ell - 2)\, 2^{-n}$ and

$$|\tilde{z} - z|_\circ \;<\; (2\, \ell - 2 + y\, 2^{-(\ell-1)n}\, \tau)\, 2^{-n}.$$

When using FFT-multiplication, we notice that the sum $v_0 + \cdots + v_{\ell-2}$ can be computed in the FFT-model, before being transformed back. In that way, we only need $2\,\ell + 1$ instead of $3\,\ell$ transforms of size $2\,n$, for a total cost of $(\frac{2}{3}\,\ell + \frac{1}{3} + o(1))\,\mathsf{I}(n)$.

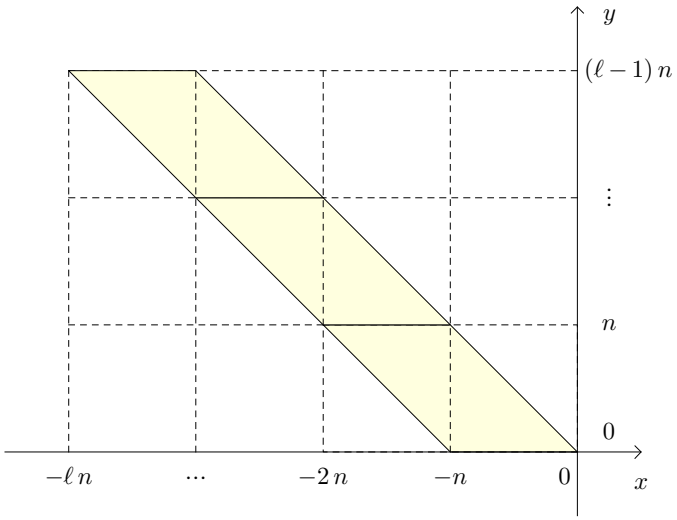

Figure 2. Product modulo one of $x \in \mathbb{D}_{\ell n, -\ell n}$ and $y \in \mathbb{D}_{(\ell-1)n, 0}$ with n significant bits.

2.4. Machine arithmetic

For actual machine implementations of large integer arithmetic, it is customary to choose a base of the form 2^w and to perform all computations with respect to that base. We will call w the *soft word size*. For processors that have good hardware support for integer arithmetic, taking $w = 32$ or $w = 64$ is usually most efficient. The GMP package [11] uses this approach.

However, modern processors are often better at floating point arithmetic. General purpose processors generally provide double precision IEEE-768 compliant instructions, whereas GPUs are frequently limited to single precision. The respective choices $w \approx 50$ and $w \approx 22$ are most adequate in these cases. It is good to pick w slightly below the maximum bitsize of the mantissa in order to accelerate carry handling. We refer to [17] for more details on the implementation of multiple precision arithmetic based on floating point arithmetic.

Another particularity of recent processors is the availability of ever wider SIMD (Single Instruction Multiple Data) instructions. For modern implementations of large integer arithmetic, we therefore recommend to focus on the problem of multiplying *several* $(1, 2, 4, 8, ...)$ large integers of a given bitsize instead of a single one. We again refer to [17] for more details.

In what follows, when using integer arithmetic, we will denote by W the maximal bitsize such that we have a hardware instruction to multiply two integers of W bits (e.g. $W = 32$ or $W = 64$). When using floating point arithmetic, we let W be the bitsize of a mantissa (i.e. $W = 23$ or $W = 53$). We will call W the *machine word size*. For implementations of multiple precision arithmetic, we always have $w \leqslant W$, but it can be interesting to take $w < W$.

For moduli m that fit into a machine word, arithmetic modulo m can be implemented efficiently using hardware instructions. In this case, the available algorithms again tend to be somewhat faster if the size of m is a few bit smaller than W. We refer to [19] for a survey of the best currently available algorithms in various cases and how to exploit SIMD instructions.

When using arithmetic modulo m, it is often possible to delay the reductions modulo m as much as possible. One typical example is modular matrix multiplication. Assume that we have two $r \times r$ matrices with coefficients modulo m (represented by integers between 0 and $m-1$, say). If $r\,m^2$ fits into a machine word, then we may multiply the matrices using integer or floating point arithmetic and reduce the result modulo m. This has the advantage that we may use standard, highly optimized implementations of matrix multiplication. One drawback is that the intermediate results before reduction require at least twice as much space. Also, the bitsize of the modulus is at least twice as small as W.

3. Asymptotically fast Chinese remaindering

3.1. The Chinese remainder theorem

For any integer $m \geqslant 1$, we will write $\mathcal{R}_m = \{0, ..., m-1\}$. We recall:

Chinese Remainder Theorem. *Let m_1, ..., m_ℓ be positive integers that are mutually coprime and denote $M = m_1 \cdots m_m$. Given $a_1 \in \mathcal{R}_{m_1}$, ..., $a_\ell \in \mathcal{R}_{m_\ell}$, there exists a unique $x \in \mathcal{R}_M$ with $x \equiv a_i \pmod{m_i}$ for $i = 1, ..., \ell$.*

We will prove the following more constructive version of this theorem.

Theorem 1. *Let m_1, ..., m_ℓ be positive integers that are mutually coprime and denote $M = m_1 \cdots m_m$. There exist c_1, ..., $c_\ell \in \mathcal{R}_M$ such that for any $a_1 \in \mathcal{R}_{m_1}$, ..., $a_\ell \in \mathcal{R}_{m_\ell}$, the number*

$$x \;=\; (c_1\,a_1 + \cdots + c_\ell\,a_\ell) \operatorname{rem} M$$

satisfies $x \equiv a_i \pmod{m_i}$ for $i = 1, ..., \ell$.

Notation. *We call $c_1, ..., c_\ell$ the cofactors for $m_1, ..., m_\ell$ in M and also denote these numbers by $c_{m_1;M} = c_1, ..., c_{m_\ell;M} = c_\ell$.*

Proof. If $\ell = 1$, then it suffices to take $c_1 = 1$. If $\ell = 2$, then the extended Euclidean algorithm allows us to compute a Bezout relation

$$k_1\,m_1 + k_2\,m_2 \;=\; 1, \tag{5}$$

where $k_1 \in \mathcal{R}_{m_2}$ and $k_2 \in \mathcal{R}_{m_1}$. Let us show that we may take

$$\begin{aligned} c_1 &= k_2\,m_2 \in \mathcal{R}_{m_1 m_2} \\ c_2 &= k_1\,m_1 \in \mathcal{R}_{m_1 m_2}. \end{aligned}$$

Indeed, given $a_1 \in \mathcal{R}_{m_1}$ and $a_2 \in \mathcal{R}_{m_2}$, we have

$$k_1\,m_1\,x + k_2\,m_2\,x \equiv x \equiv c_1\,a_1 + c_2\,a_2 \equiv k_1\,m_1\,a_2 + k_2\,m_2\,a_1 \pmod{m_1\,m_2}.$$

In particular, m_2 divides $k_1\,m_1\,(x - a_2)$. Since (5) implies $\gcd(k_1\,m_1, m_2) = 1$, it follows that $x \equiv a_2 \pmod{m_2}$. Similarly, $x \equiv a_1 \pmod{m_1}$.

For $\ell > 2$, we will use induction. Let $h = \lfloor \ell/2 \rfloor$, $M_1 = m_1 \cdots m_h$ and $M_2 = m_{h+1} \cdots m_l$. By induction, we may compute $c_{M_1;M}$, $c_{M_2;M}$, $c_{m_1;M_1}$, ..., $c_{m_h;M_1}$ and $c_{m_{h+1};M_2}$, ..., $c_{m_\ell;M_2}$. We claim that we may take

$$\begin{aligned} c_{m_i} &= c_{m_i;M_1}\,c_{M_1;M} \qquad (i = 1, ..., h) \\ c_{m_i} &= c_{m_i;M_2}\,c_{M_2;M} \qquad (i = h+1, ..., \ell). \end{aligned}$$

Indeed, for $i = 1, ..., h$, we get

$$\begin{aligned} x &\equiv c_{M_1;M}\,(c_{m_1;M_1}\,a_1 + \cdots + c_{m_h;M_1}\,a_h) + c_{M_2;M}\,(c_{m_{h+1};M_2}\,a_{h+1} + \cdots + c_{m_\ell;M_2}\,a_\ell) \\ &\equiv c_{m_1;M_1}\,a_1 + \cdots + c_{m_h;M_1}\,a_h \,(\mathrm{mod}\, M_1), \end{aligned}$$

whence $x \equiv a_i \,(\mathrm{mod}\, m_i)$. For $i = h+1, ..., \ell$ we obtain $x \equiv a_i \,(\mathrm{mod}\, m_i)$ in a similar way. □

3.2. Naive multi-modular reduction and reconstruction

Let $m_1, ..., m_\ell$, $M = m_1 \cdots m_\ell$, $a_1 \in \mathcal{R}_{m_1}, ..., a_\ell \in \mathcal{R}_{m_\ell}$ and $x \in \mathcal{R}_M$ be as in the Chinese remainder theorem. We will refer to the computation of $a_1, ..., a_\ell$ as a function of x as the problem of *multi-modular reduction*. The inverse problem is called *multi-modular reconstruction*. In what follows, we assume that $m_1, ..., m_\ell$ have been fixed once and for all.

The simplest way to perform multi-modular reduction is to simply take

$$a_i := x \,\mathrm{rem}\, m_i \qquad (i = 1, ..., \ell). \tag{6}$$

Inversely, Theorem 1 provides us with a formula for multi-modular reconstruction:

$$x := (c_{m_1;M}\,a_1 + \cdots + c_{m_\ell;M}\,a_\ell) \,\mathrm{rem}\, M. \tag{7}$$

Since $m_1, ..., m_\ell$ are fixed, the computation of the cofactors $c_{m_1;M}$ can be regarded as a pre-computation.

Let us analyze the cost of the above methods in terms of the complexity $\mathsf{I}(n)$ of n-bit integer multiplication. Assume that $m_i < 2^n$ for $i = 1, ..., \ell$. Then multi-modular reduction can clearly be done in time $\ell\,\mathsf{D}(\ell\,n, n) = (4/3\,\ell - 1 + o(1))\,\ell\,\mathsf{I}(n)$.

As to multi-modular reconstruction, assume that $m_i < 2^{n'}$ for $i = 1, ..., \ell$, where $n' := n - \lceil \log_2 \ell \rceil$ is such that $\ell\, 2^{n'} \leqslant 2^n$. Cutting the cofactors in chunks of n bits as in (4), we precompute the Fourier transform of all obtained chunks. The Fourier transforms of $a_1, ..., a_\ell$ can be computed in time $\leqslant \ell\,\mathsf{F}(2\,n)$. The sum $S = c_{m_1;M}\,a_1 + \cdots + c_{m_\ell;M}\,a_\ell$ can be computed in the Fourier model in time $\mathsf{N}(2\,n)\,\ell^2$ and transformed back in time $\mathsf{F}(2\,n)\,\ell + O(n\,\ell)$. Our assumption that $\ell\, m_i < 2^n$ for $i = 1, ..., \ell$ ensures that the computed sum is correct. The remainder S rem M can be computed in time $\mathsf{D}_{\mathrm{fixed}}((\ell\,+\,1)\,n, \ell\,n) \leqslant \mathsf{F}(2\,n)\,\ell + 5\,\mathsf{F}(2\,n) + O(\mathsf{N}(2\,n)\,\ell)$. The total time of the reconstruction is therefore bounded by

$$\mathsf{C}^*_{n',\mathrm{naive}}(\ell) = (\mathsf{N}(2\,n)\,\ell^2 + 2\,\mathsf{F}(2\,n)\,\ell + 5\,\mathsf{F}(2\,n) + O(\mathsf{N}(2\,n)\,\ell)). \tag{8}$$

If we only assume that $m_i < 2^n$, then we need to increase the bitsize n by $\lceil \log_2 \ell \rceil$. If $\ell \log \ell = O(n)$, then this means that we have to multiply the right-hand side of (8) by $1 + O(\log \ell / n) = 1 + O(\ell^{-1})$.

3.3. Scaled remainders

The above complexity analysis shows that naive multi-modular recomposition can be done faster than naive multi-modular reduction. In order to make the latter operation more efficient, one may work with *scaled remainders* that were introduced in [2]. The idea is that each remainder of the form $u \,\mathrm{rem}\, P$ is replaced by $\frac{u}{P} \,\mathrm{rem}\, 1$. The quotient $\frac{u}{P}$ is regarded as a real number and its remainder modulo 1 as a number in the interval $[0, 1)$.

If we allow ourselves to compute with exact real numbers, then this leads us to replace the relation (6) by

$$\frac{x}{m_i} \,\mathrm{rem}\, 1 = \left[\frac{M}{m_i} \left(\frac{x}{M} \,\mathrm{rem}\, 1 \right) \right] \mathrm{rem}\, 1 \qquad (i = 1, ..., \ell) \tag{9}$$

and (7) by

$$\frac{x}{M} \operatorname{rem} 1 \;=\; \left[\frac{c_{m_1;M}\, m_1}{M} \left(\frac{a_1}{m_1} \operatorname{rem} 1 \right) + \cdots + \frac{c_{m_\ell;M}\, m_\ell}{M} \left(\frac{a_\ell}{m_\ell} \operatorname{rem} 1 \right) \right] \operatorname{rem} 1. \tag{10}$$

For actual machine computations, we rather work with fixed point approximations of the scaled remainders. In order to retrieve the actual remainder $u \operatorname{rem} P$ from the scaled one $\frac{u}{P} \operatorname{rem} 1$, we need a circular $(2\,P)^{-1}$-approximation of $\frac{u}{P} \operatorname{rem} 1$.

Now assume that $m_1, \ldots, m_k \in \mathcal{R}_{2^{n'}}$ with

$$n' \;\leqslant\; n - \lceil \log_2 (4\,\ell) \rceil.$$

Given a circular $[\tau\, 2^{-\ell n}]$-approximation of $\frac{x}{M} \operatorname{rem} 1$ in $\mathbb{D}_{\ell n, -\ell n}$ with

$$\tau \;\leqslant\; 2^{(n-n')(\ell-1)},$$

the algorithm at the end of section 2.3 allows us to compute a circular $[2\,\ell\, 2^{-n}]$-approximation modulo 1 of $\frac{x}{m_i} \operatorname{rem} 1$, by applying the formula (9). Since $2\,\ell\, 2^{-n} \leqslant 2^{-n'-1}$, we may then recover the number $x \operatorname{rem} m_i$ using one final n-bit multiplication. Moreover, in the FFT-model, the transforms for $\frac{x}{M} \operatorname{rem} 1$ need to be computed only once and the transforms for the numbers $\frac{M}{m_i}$ can be precomputed. In summary, given an approximation for the scaled remainder $\frac{x}{M} \operatorname{rem} 1$, we may thus compute approximations for the scaled remainders $\frac{x}{m_i} \operatorname{rem} 1$ in time

$$\mathsf{C}_{n',\mathrm{scaled}}(\ell) \;=\; \mathsf{N}(2\,n)\, \ell^2 + 2\, \mathsf{F}(2\,n)\, \ell + O(\mathsf{N}(2\,n)\, \ell). \tag{11}$$

From this, we may recover the actual remainders $x \operatorname{rem} m_i$ in time $\ell\, \mathsf{I}(n)$.

Scaled remainders can also be used for multi-modular reconstruction, but carry handling requires more care and the overall payoff is less when compared to the algorithm from the previous subsection.

3.4. Remainder trees

It is well-known that Chinese remaindering can be accelerated using a similar dichotomic technique as in the proof of Theorem 1. This time, we subdivide $\mathcal{M} = \{m_1, \ldots, m_\ell\}$ into k parts $\mathcal{M}_1 = \{m_{\ell_0+1}, \ldots, m_{\ell_1}\}, \ldots, \mathcal{M}_k = \{m_{\ell_{k-1}+1} + 1, \ldots, m_{\ell_k}\}$ with $\ell_j = \lfloor (j\,\ell)/k \rfloor$ for $j = 0, \ldots, k$. We denote $M_j = m_{\ell_{j-1}+1} \cdots m_{\ell_j}$ and assume that $\ell \geqslant k$ (if $\ell < k$, then we apply the native algorithms from the previous subsections).

Fast multi-modular reduction proceeds as follows. We first compute

$$X_j \;=\; x \operatorname{rem} M_j \qquad (j = 1, \ldots, k) \tag{12}$$

using the algorithm from the previous subsection. Next, we recursively apply fast multi-modular reduction to obtain

$$a_i \;=\; X_j \operatorname{rem} m_i \qquad (i = \ell_{j-1} + 1, \ldots, \ell_j). \tag{13}$$

The computation process can be represented in the form of a so called *remainder tree*; see Figure 3. The root of the tree is labeled by $x \bmod M$. The children of the root are the remainder trees for X_j modulo M_j, where $j = 1, \ldots, k$. If needed, then the arity k can be adjusted as a function of the bitsize of the moduli and ℓ.

Fast multi-modular reconstruction is done in a similar way, following the opposite direction. We first reconstruct

$$X_j \;=\; \left(c_{m_{\ell_{j-1}+1};M_j}\, a_{\ell_{j-1}+1} + \cdots + c_{m_{\ell_j};M_j}\, a_{\ell_j} \right) \operatorname{rem} M_j \qquad (j = 1, \ldots, k), \tag{14}$$

followed by

$$x \;=\; (c_{M_1;M}\, X_1 + \cdots + c_{M_k;M}\, X_k) \operatorname{rem} M. \tag{15}$$

The computation flow can again be represented using a tree, but this time the computations are done in a bottom-up way.

Following the approach from subsection 3.3, it is also possible to systematically work with fixed point approximations of scaled remainders $\frac{u}{P} \operatorname{rem} 1$ instead of usual remainders $u \operatorname{rem} P$. In that case, the computation process gives rise to a *scaled remainder tree* as in Figure 4. Of course, the precision of the approximations has to be chosen with care. Before we come to this, let us first show how to choose k.

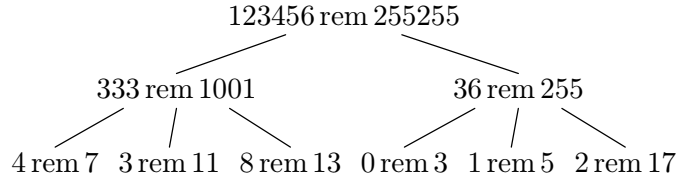

Figure 3. Example of a remainder tree with arities $k=2$ and $k=3$ at the two recursion levels. In the case of a reduction, the remainders are computed top-down. In the case of a reconstruction, they are reconstructed in a bottom-up fashion.

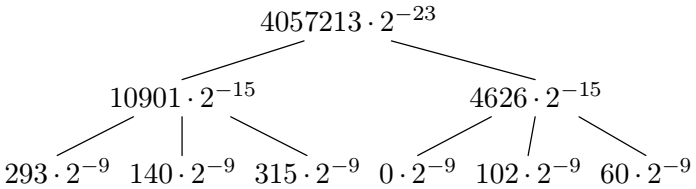

Figure 4. The scaled remainder tree corresponding to Example 3, while using fixed point approximations for the scaled remainders.

3.5. Specification of the arities of nodes in the remainder tree

Let us first focus on multi-modular reconstruction. In order to make this process as efficient as possible, the arity k should be selected as a function of n and ℓ so as to make $\mathsf{N}(2\,n)\,\ell^2$ as large as possible in (8), while remaining negligible with respect to $\mathsf{F}(2\,n)\,\ell$. Let

$$\Phi_n(\ell) \;=\; \frac{\mathsf{F}(2\,\ell\,n)}{\mathsf{N}(2\,\ell\,n)\log^2\log(2\,\ell\,n)} \;=\; o\left(\frac{\mathsf{F}(2\,\ell\,n)}{\mathsf{N}(2\,\ell\,n)}\right).$$

For inborn FFT schemes, we notice that

$$\Phi_n(\ell) \;=\; o\left(\frac{\log(2\,\ell\,n)}{\log\log(2\,\ell\,n)}\right)$$

$$\Phi_n(\ell)^{-1} \;=\; O\left(\frac{\log^2\log(2\,\ell\,n)}{\log(2\,\ell\,n)}\right) \;=\; O\left(\frac{\log^2\log n}{\log n}\right).$$

For the root of the remainder tree, we take

$$k \;=\; \Psi_n(\ell) \;=\; \begin{cases} \ell & \text{if } \ell \leqslant \Phi_n(\ell) \\ \lceil \sqrt{\ell} \rceil & \text{if } \ell^{2/3} \leqslant \Phi_n(\ell) < \ell \\ \Phi_n(\ell) & \text{otherwise} \end{cases}$$

Using the same formula recursively for the other levels of the remainder tree, it is natural to define the sequences $\ell_1, \ldots, \ell_{r+1}$ and $k_1, \ldots, k_r$ by $\ell_1 = \ell$, $k_i = \Psi_n(\ell_i)$ and $\ell_{i+1} = \lceil \ell_i / k_i \rceil$ for $i = 1, \ldots, r$; the construction stops as soon as $\ell_{r+1} = 1$. Notice that we always have $\Psi_\ell(n) \leqslant \Phi_\ell(n)$. The precise choice of $k = \Psi_n(\ell)$ is motivated by the following lemmas.

LEMMA 2. *If $r > 1$, then $\ell_r^{-1} = O((\log n)^{-1/3})$.*

Proof. We clearly cannot have $\ell_{r-1} \leqslant \Phi_n(\ell_{r-1})$, since otherwise $\ell_r = 1$. If $\ell_{r-1}^{2/3} \leqslant \Phi_n(\ell_{r-1}) < \ell_{r-1}$, then

$$\ell_r^{-1} = \lceil \ell_{r-1} / \lceil \sqrt{\ell_{r-1}} \rceil \rceil^{-1} = O\big(\ell_{r-1}^{-1/2}\big) \leqslant O(\Phi_n(\ell_{r-1})^{-1/2}) = O((\log n)^{-1/3}).$$

If $\Phi_n(\ell_{r-1}) < \ell_{r-1}^{2/3}$, then

$$\ell_r^{-1} = \lceil \ell_{r-1} / \Phi_n(\ell_{r-1}) \rceil^{-1} < \ell_{r-1}^{-1/3} < \Phi_n(\ell_{r-1})^{-1/2} = O((\log n)^{-1/3}).$$

This proves the result in all cases. □

LEMMA 3. *If $r > 1$, then we have $k_1 \cdots k_{i-1} \ell_i \sim \ell$ for $i = 1, \ldots, r+1$.*

Proof. For $i = 1, \ldots, r-1$, we have $\ell_i^{-1} < \Phi_n(\ell_i)^{-1} = O(\log^2 \log n / \log n)$, so that $k_i \geqslant 3$ and $\ell_{i+1} < \ell_i / 2$ whenever n is sufficiently large. By construction, we also have

$$k_i \ell_{i+1} \leqslant \left(1 + \tfrac{1}{\ell_i}\right) \ell_i.$$

By induction, it follows that

$$\begin{aligned} k_1 \cdots k_{i-1} \ell_i &\leqslant \left(1 + \tfrac{1}{\ell_i}\right) \cdots \left(1 + \tfrac{1}{\ell_1}\right) \ell_1 \\ &\leqslant \exp\left(\tfrac{1}{\ell_r} + \cdots + \tfrac{1}{\ell_1}\right) \ell_1 \\ &\leqslant \exp\left(\tfrac{2}{\ell_r}\right) \ell_1 \\ &\sim \ell, \end{aligned}$$

for $i = 1, \ldots, r$. We also have $k_1 \cdots k_r \ell_{r+1} = k_1 \cdots k_{r-1} \ell_r \sim \ell$. □

LEMMA 4. *We have $r \leqslant (1 + o(1)) \frac{\log \ell}{\log \log (\ell n)}$.*

Proof. Let $\epsilon > 0$ and let s be smallest such that $\log \Phi_n(\ell_s) < (1 - \epsilon) \log \log (n \ell) - 1$. For all $i < s$, we have $\log \ell_i \geqslant (1 - \epsilon) \log \log (n \ell) + \log \ell_{i+1}$, whence

$$s \leqslant \frac{\log \ell}{(1 - \epsilon) \log \log (\ell n)}. \tag{16}$$

Let $c > 0$ be a constant such that $\Phi_n(\ell) > c \frac{\log (\ell n)}{\log^2 \log (\ell n)}$ for all ℓ. We also have

$$\log\left(c \frac{\log (\ell_s n)}{\log^2 \log (\ell_s n)} \right) < \log \Phi_n(\ell_s) < (1 - \epsilon) \log \log (n \ell),$$

so that

$$\begin{aligned} \log(\ell_s n) &= O(\log^2 \log (\ell_s n) (\log (\ell n))^{1-\epsilon}) \\ &= O(\log^2 \log (\ell n) (\log (\ell n))^{1-\epsilon}). \end{aligned}$$

Since $\ell_i \geqslant 2 \ell_{i+1}$ for all $i < r$, it follows that

$$r - s \leqslant \frac{\log \ell_s}{\log 2} = O(\log^2 \log (\ell n) (\log (\ell n))^{1-\epsilon}). \tag{17}$$

Adding up (16) and (17) while letting ϵ tend to zero, the result follows. □

LEMMA 5. *We have $\ell_r = O(\log n)$.*

Proof. By construction, $\ell_r \leqslant \Phi_n(\ell_r) = O(\log (\ell_r n))$. For large n, this means that $\ell_r < n$, since otherwise $\ell_r = O(\log(\ell_r^2)) = O(\log \ell_r)$, which is impossible. Consequently, $\ell_r = O(\log(n^2)) = O(\log n)$. □

3.6. Complexity analysis of multi-modular reconstruction

Let $\mathsf{C}^*_n(\ell)$ be the complexity multi-modular reconstruction for fixed moduli $m_1, ..., m_\ell$ with $m_i < 2^n$ for $i = 1, ..., \ell$.

THEOREM 6. *If integer multiplication is based on an inborn FFT scheme, then*

$$\mathsf{C}^*_n(\ell) \leqslant (2/3 + o(1))\,\mathsf{I}(\ell\, n) \max\left(\frac{\log \ell}{\log\log(n\,\ell)}, 1 + O(\ell^{-1})\right). \tag{18}$$

This bound holds uniformly in ℓ for $n \to \infty$.

Proof. In the special case when $r = 1$, the bound (8) yields

$$\begin{aligned}
\mathsf{C}^*_n(\ell) &\leqslant (1 + O(\ell^{-1}))\,(\mathsf{N}(2\,n)\,\ell^2 + 2\,\mathsf{F}(2\,n)\,\ell + 5\,\mathsf{F}(2\,n)) + O(\mathsf{N}(2\,n)\,\ell)\\
&= (2 + O(\ell^{-1}))\,\mathsf{F}(2\,n)\,\ell + (1 + O(\ell^{-1}))\,\mathsf{N}(2\,n)\,\ell^2\\
&\leqslant (2 + O(\ell^{-1}))\,\mathsf{F}(2\,n)\,\ell + (1 + O(\ell^{-1}))\,\mathsf{N}(2\,n)\,\Phi_n(\ell)\,\ell\\
&= (2 + O(\ell^{-1}) + o(1))\,\mathsf{F}(2\,n)\,\ell\\
&= (2/3 + o(1))\,\mathsf{I}(n)\,\ell + O(\mathsf{I}(n)),
\end{aligned}$$

and we are done. If $r > 1$, then (8) implies

$$\begin{aligned}
\mathsf{C}^*_n(\ell_i) &\leqslant (2 + O(k_i^{-1}))\,\mathsf{F}(2\,n\,\ell_{i+1})\,k_i + (1 + O(k_i^{-1}))\,\mathsf{N}(2\,n\,\ell_{i+1})\,k_i^2 + \mathsf{C}^*_n(\ell_{i+1})\,k_i\\
&\leqslant (2 + o(1))\,\mathsf{F}(2\,n\,\ell_{i+1})\,k_i + \mathsf{C}^*_n(\ell_{i+1})\,k_i,
\end{aligned}$$

for $i = 1, ..., r$. By induction, and using the fact that $\mathsf{C}^*_n(1) = 0$, we get

$$\begin{aligned}
\mathsf{C}^*_n(\ell) &\leqslant \sum_{i=1}^{r} (2 + o(1))\,\mathsf{F}(2\,n\,\ell_{i+1})\,k_1 \cdots k_i.\\
&= \sum_{i=1}^{r} (2 + o(1))\,\mathsf{F}(2\,n\,\ell_{i+1})\,\frac{\ell}{\ell_{i+1}}.\\
&\leqslant \sum_{i=1}^{r} (2 + o(1))\,\mathsf{F}(2\,n\,\ell)\\
&= (2 + o(1))\,r\,\mathsf{F}(2\,n\,\ell).
\end{aligned}$$

The result now follows from Lemma 4 and (1). □

Remark 7. For $\ell^{-1} = o(1)$ and $\ell = O(\log n)$, the bound (18) simplifies into

$$\mathsf{C}^*_n(\ell) \leqslant (2/3 + o(1))\,\mathsf{I}(\ell\, n).$$

If $\log n = O(\log \ell)$, then the bound becomes

$$\mathsf{C}^*_n(\ell) \leqslant (2/3 + o(1))\,\mathsf{I}(\ell\, n)\,\frac{\log \ell}{\log\log \ell}.$$

Remark 8. It is interesting to examine the cost of the precomputations as a function of the parameters n, ℓ and $m_1, ..., m_\ell$. For a node of the remainder tree at level i, we essentially need to compute k_i cofactors and their transforms. This can be done in time $O(k_i\,\mathsf{I}(n\,\ell_i))$. Since we have $k_1 \cdots k_{i-1}$ nodes at level i, the total precomputation time at level i is therefore bounded by $O(k_1 \cdots k_i\,\mathsf{I}(n\,\ell_i)) = O(k_i\,\mathsf{I}(n\,\ell))$. Now $k_i = o(\log(n\,\ell_i)/\log\log(n\,\ell_i)) = o(\log(n\,\ell)/\log\log(n\,\ell))$. Consequently, the total precomputation time is bounded by

$$\begin{aligned}
\mathsf{C}^*_{n,\mathrm{pre}}(\ell) &= o\left(r\,\mathsf{I}(n\,\ell)\,\frac{\log(n\,\ell)}{\log\log(n\,\ell)}\right)\\
&= o\left(\mathsf{I}(n\,\ell)\,\frac{\log(n\,\ell)\log\ell}{\log^2\log(n\,\ell)}\right).
\end{aligned}$$

3.7. Complexity analysis of multi-modular reduction

Let us now consider the complexity $\mathsf{C}_n(\ell)$ of multi-modular reduction for fixed moduli $m_1, ..., m_\ell$ with $m_i < 2^n$ for $i = 1, ..., \ell$. In this case, it is most efficient to work with scaled remainders, so the algorithm contains three main steps:

1. The initial conversion of $x \operatorname{rem} M$ into (an approximation of) $\frac{x}{M} \operatorname{rem} 1$.
2. The computation of (approximations of) the scaled remainders $\frac{x}{m_i} \operatorname{rem} 1$.
3. The final conversions of (approximations of) $\frac{x}{m_i} \operatorname{rem} 1$ into $x \operatorname{rem} m_i$.

At a first stage, we will assume that $m_1, ..., m_\ell < 2^{n'}$, where $n' < n$ is sufficiently small such that the final approximations of the scaled remainders $\frac{x}{m_i}$ rem 1 allow us to recover the usual remainders $x \operatorname{rem} m_i$.

Let $\tilde{\mathsf{C}}_n(\ell)$ denote the cost of step 2. The conversions in steps 1 and 3 boil down to multiplications with fixed arguments, so that

$$\mathsf{C}_{n'}(\ell) \leqslant \tilde{\mathsf{C}}_n(\ell) + (4/3 + o(1))\, \mathsf{I}(n\,\ell). \tag{19}$$

For step 2, we use the scaled remainder tree algorithm from subsection 3.4, while taking the arities k as in subsection 3.5.

Our next task is to show that $n' := n - \lceil \log_2(4\,\ell_r) \rceil$ is small enough in order to recover all remainders $x \operatorname{rem} m_i$.

LEMMA 9. *There exists a constant n_0 such that for all $n \geqslant n_0$ and $i = 1, ..., r$, we have*

$$2\,k_i \leqslant 2^{\lceil \log_2(4\ell_r) \rceil\, \ell_{i+1}} = 2^{(n-n')\ell_{i+1}}.$$

Proof. For $i = r$ the result clearly holds, so assume that $i < r$. In particular, if n is sufficiently large, then it follows that $k_i \leqslant \lceil \sqrt{\ell_i} \rceil$. Now assume for contradiction that $2\,k_i > 2^{\lceil \log_2(4\ell_r) \rceil\, \ell_{i+1}} > 2 \cdot 2^{\ell_{i+1}}$. Then we would get $\ell_i \leqslant k_i\, \ell_{i+1} < k_i \log_2 k_i < \lceil \sqrt{\ell_i} \rceil \log_2 \lceil \sqrt{\ell_i} \rceil$. This is impossible for $\ell_i \geqslant 2$. □

Now starting with a circular $2^{-\ell n}$-approximation of $\frac{x}{M}$ rem 1, the scaled reduction algorithm from subsection 3.3 yields circular $[2\ k_1\ 2^{-\ell_2}]$-approximations for the scaled remainders $\frac{X_j}{M_j} \operatorname{rem} 1$ at level $i = 2$. Lemma 9 now shows that $\tau = 2\,k_1$ is sufficiently small for a continued application of the same algorithm for the next level. By induction over i, the same reasoning shows that the scaled remainders at the $(i+1)$-th level are computed with an error below

$$2\,k_i\, 2^{-\ell_{i+1} n} \leqslant 2^{(n-n')\ell_{i+1}} \cdot 2^{-\ell_{i+1} n} \leqslant 2^{(n-n')\ell_{i+1}(k_{i+1}-1)} \cdot 2^{-\ell_{i+1} n}.$$

At the very end, we obtain $(2\,k_r\, 2^{-n})$-approximations for the scaled remainders $\frac{x}{m_i} \operatorname{rem} 1$. Since $\ell_r = k_r$, this allows us to reconstruct each remainder $x \operatorname{rem} m_i$ using a multiplication by m_i. This shows that n' is indeed small enough for the algorithm to work.

THEOREM 10. *If integer multiplication is based on an inborn FFT scheme, then*

$$\mathsf{C}_n(\ell) \leqslant (2/3 + o(1))\, \mathsf{I}(\ell\, n) \left[\max\left(\frac{\log \ell}{\log\log(n\,\ell)}, 1 \right) + 2 \right]. \tag{20}$$

This bound holds uniformly in ℓ for $n \to \infty$.

Proof. A similar cost analysis as in the proof of Theorem 6 yields

$$\tilde{\mathsf{C}}_n(\ell) \leqslant \mathsf{N}(2\,n)\, \ell^2 + 2\,\mathsf{F}(2\,n)\, \ell + O(\mathsf{N}(2\,n)\, \ell) = (2/3 + o(1))\, \mathsf{I}(n)\, \ell$$

when $r = 1$ and

$$\tilde{\mathsf{C}}_n(\ell) \;\leqslant\; (2 + o(1))\, r\, \mathsf{F}(2\, n\, \ell)$$

when $r > 1$. In both cases, combination with (19) yields

$$\mathsf{C}_{n'}(\ell) \;\leqslant\; (2/3 + o(1))\, \mathsf{I}(\ell\, n) \left[\max\left(\frac{\log \ell}{\log\log (n\, \ell)}, 1 \right) + 2 \right].$$

Notice also that $n' = n - \lceil \log_2(4\, \ell_r) \rceil \geqslant n - O(\log n)$, by Lemma 5.

Given a number $n^* > n$, we may construct a similar sequence $\ell_1^*, \ldots, \ell_{r^*+1}^*$ when using n^* in the role of n. Taking n^* minimal such that $n^* - \lceil \log_2(4\, \ell_{r^*}^*) \rceil \geqslant n$, we have

$$\mathsf{C}_n(\ell) \;\leqslant\; (2/3 + o(1))\, \mathsf{I}(\ell\, n^*) \left[\max\left(\frac{\log \ell}{\log\log (n^*\, \ell)}, 1 \right) + 2 \right]. \tag{21}$$

Moreover, $n \geqslant n^* - O(\log n^*)$, which implies $n^* \leqslant n + O(\log n)$. Plugging this into (21), the result follows, since $\log\log ((n + O(\log n))\, \ell) \sim \log\log (n\, \ell)$ and the assumption that $\mathsf{I}(n) / (n \log n)$ is increasing implies $\mathsf{I}(\ell\, (n + O(\log n))) \sim \mathsf{I}(\ell\, n)$. □

Remark 11. For $\ell^{-1} = o(1)$ and $\ell = O(\log n)$, the bound (20) simplifies into

$$\mathsf{C}_n(\ell) \;\leqslant\; (2 + o(1))\, \mathsf{I}(\ell\, n).$$

If $\log\log n = o(\log \ell)$, then the bound (20) becomes

$$\mathsf{C}_n(\ell) \;\leqslant\; (2/3 + o(1))\, \mathsf{I}(\ell\, n)\, \frac{\log \ell}{\log\log (n\, \ell)}.$$

For very large ℓ with $\log n = O(\log \ell)$, this yields

$$\mathsf{C}_n(\ell) \;\leqslant\; (2/3 + o(1))\, \mathsf{I}(\ell\, n)\, \frac{\log \ell}{\log\log \ell}.$$

Remark 12. Using a similar analysis as in Remark 8, the cost of all precomputations as a function of n, ℓ and $m_1, \ldots, m_\ell$ is again bounded by

$$\mathsf{C}_{n,\mathrm{pre}}(\ell) \;=\; o\left(\mathsf{I}(n\, \ell)\, \frac{\log (n\, \ell) \log \ell}{\log^2 \log (n\, \ell)} \right).$$

3.8. Base conversion

The approach of this section can also be used for the related problem such as base conversion. Let $b \in \mathcal{R}_{2^n}$ and ℓ be a fixed base and order. Given a number $x \in \mathcal{R}_{b^\ell}$, the problem is to compute $x_0, \ldots, x_{\ell-1} \in \mathcal{R}_b$ with

$$x \;=\; x_0 + x_1\, b + \cdots + x_{\ell-1}\, b^{\ell-1}. \tag{22}$$

Inversely, one may wish to reconstruct x from x_0, ..., $x_{\ell-1}$. It is well known that both problems can be solved using a similar remainder tree process as in the case of Chinese remainders. The analogues for the formulas (7) and (9) are (22) and

$$\frac{x_i}{b} \operatorname{rem} 1 \;=\; \left[b^{\ell-1-i} \left(\frac{x}{b^\ell} \operatorname{rem} 1 \right) \right] \operatorname{rem} 1 \qquad (i = 0, \ldots, \ell - 1). \tag{23}$$

The analogue of the recursive process of subsection 3.4 reduces a problem of size ℓ to k similar problems of size $\lceil \ell \,/\, k \rceil$ and one similar problem of size k but for the base $b^{\lceil \ell / k \rceil}$. A routine verification shows that the complexity bounds (18) and (20) also apply in this context.

Moreover, for nodes of the remainder tree at the same level, the analogues of the cofactors and the multiplicands M / m_i in (9) do not vary as a function of the node. For this reason, the required precomputations as a function of b and ℓ can actually be done much faster. This makes it possible to drop the hypothesis that b and ℓ are fixed and consider these parameters as part of the input. Let us denote by $\mathsf{B}_n(\ell)$ and $\mathsf{B}_n^*(\ell)$ the complexities of computing $x_0, ..., x_{\ell-1}$ as a function of x and *vice versa*.

THEOREM 13. *If integer multiplication is based on an inborn FFT scheme, then*

$$\mathsf{B}_n^*(\ell) \leqslant (2/3 + o(1))\, \mathsf{I}(\ell n) \left(\frac{\log \ell}{\log\log (n\,\ell)} + O(\log\log \ell) \right) \tag{24}$$

$$\mathsf{B}_n(\ell) \leqslant (2/3 + o(1))\, \mathsf{I}(\ell n) \left(\frac{\log \ell}{\log\log (n\,\ell)} + O(\log\log \ell) \right). \tag{25}$$

These bound holds uniformly in ℓ for $n \to \infty$.

Proof. Let us estimate the cost of the precomputations as a function of b and ℓ. The analysis is similar as in Remark 8 except that we only have to do the precomputations for a single node of the tree at each level i. Consequently, the precomputation time is now bounded by $O(k_1\, \mathsf{I}(n\,\ell_1) + \cdots + k_r\, \mathsf{I}(n\,\ell_r))$. Since the $\ell_1, \ell_2, ...$ decrease with at least geometric speed, this cost is dominated by $O(k_1\, \mathsf{I}(n\,\ell_1)) = o\Big(\mathsf{I}(n\,\ell)\, \frac{\log(n\,\ell)}{\log\log(n\,\ell)}\Big)$. This proves the result under the condition that $\log n = O(\log \ell)$.

If $\log \ell = o(\log n)$, then we need to construct k_1, ..., k_r in a slightly different way. Assuming that n is sufficiently large, let t be maximal such that

$$2^{2^{t-1}} \leqslant \Psi_n\Big(\Big\lceil \frac{2\,\ell}{2^{2^{t-1}}} \Big\rceil\Big).$$

Notice that

$$t \leqslant \lceil \log_2 \log_2 \ell \rceil.$$

We again set $\ell_1 = \ell$ and $\ell_{i+1} = \lceil \ell_i / k_i \rceil$. This time, we take $k_i = 2^{2^{i-1}}$ for $i \leqslant t$ and proceed with the usual construction $k_i = k(\ell_i)$ for $i > t$. It follows that

$$\begin{aligned}
\ell_i &= \Big\lceil \frac{2\,\ell}{2^{2^{i-1}}} \Big\rceil && (i = 1, ..., t+1)\\
k_i &\leqslant \Psi_n(\ell_i) && (i = 1, ..., t)\\
k_{t+1} &= \Psi_n(\ell_{t+1}) < 2^{2^t}\\
k_{t+1}\,\ell_{t+1} &= O(\ell)
\end{aligned}$$

and

$$r \leqslant t + \frac{\log \ell}{\log\log (n\,\ell)} + o(1).$$

Using the new bound for r, a similar complexity analysis as in the proofs of Theorems 6 and 10 yields the bounds (24) and (25) when forgetting about the cost of the precomputations. Now the cost P_1 of the precomputations for the first t levels is bounded by

$$\begin{aligned}
\mathsf{P}_1 &= O(k_1\, \mathsf{I}(n\,\ell_1) + k_2\, \mathsf{I}(n\,\ell_2) + \cdots + k_t\, \mathsf{I}(n\,\ell_t))\\
&= O\Big(2\, \mathsf{I}\Big(\frac{2\,n\,\ell}{2}\Big) + 4\, \mathsf{I}\Big(\frac{2\,n\,\ell}{4}\Big) + \cdots + 2^{2^{t-1}}\, \mathsf{I}\Big(\frac{2\,n\,\ell}{2^{2^{t-1}}}\Big)\Big)\\
&= O(t\, \mathsf{I}(n\,\ell))
\end{aligned}$$

and the cost P_2 for the remaining levels by

$$\begin{aligned}
\mathsf{P}_2 &= O(k_{t+1}\, \mathsf{I}(n\,\ell_{t+1}) + \cdots + k_r\, \mathsf{I}(n\,\ell_r))\\
&= O(k_{t+1}\, \mathsf{I}(n\,\ell_{t+1}))\\
&= O\Big(k_{t+1}\, \frac{\ell_{t+1}}{\ell}\, \mathsf{I}(n\,\ell)\Big)\\
&= O(\mathsf{I}(n\,\ell)).
\end{aligned}$$

We conclude that the right-hand sides of (24) and (25) absorb the cost $\mathsf{P}_1 + \mathsf{P}_2$ of the precomputations. □

3.9. Polynomial analogues

It is quite straightforward to adapt the theory of this section to univariate polynomials instead of integers. An example of this kind of adaptations can be found in [2]. In particular, this yields efficient algorithms for multi-point evaluation and interpolation in the case when the evaluation points are fixed. The analogues of our algorithms for base conversion yield efficient methods for p-adic expansions and reconstruction.

More precisely, let R be an effective commutative ring and let $\mathsf{M}(n)$ be the cost of multiplying two polynomials of degree $<n$ in $R[x]$. Assume that R allows for inborn FFT multiplication. Then $\mathsf{M}(n) = 3\,\mathsf{F}(2\,n) + \mathsf{N}(2\,n)$, where F and N satisfy similar properties as in the integer case. Let $Q_1, ..., Q_\ell$ be ℓ monic polynomials of degree n. Given a polynomial P of degree $<\ell\,n$ in $R[x]$ we may then compute the remainders $P \bmod Q_i$ for $i = 1, ..., \ell$ in time

$$\mathsf{C}_{n,R}(\ell) \;\leqslant\; (2/3 + o(1))\,\mathsf{M}(\ell\,n)\left[\max\left(\frac{\log \ell}{\log\log(n\,\ell)}, 1\right) + 2\right].$$

The reconstruction of P from these remainders can be done in time

$$\mathsf{C}^*_{n,R}(\ell) \;\leqslant\; (2/3 + o(1))\,\mathsf{M}(\ell\,n)\max\left(\frac{\log \ell}{\log\log(n\,\ell)}, 1 + O(\ell^{-1})\right).$$

The assumption that R admits a suitable inborn FFT scheme is in particular satisfied if R is a finite field [14]. When working in an algebraic complexity model, this is still the case if R is any field of positive characteristic [14]. For general fields of characteristic zero, the best known FFT schemes rely on the adjunction of artificial roots of unity [6]. In that case, our techniques only result in an asymptotic speed-up by a factor $\log\log\log(n\,\ell)$ instead of $\log\log(n\,\ell)$. Nevertheless, the field of complex numbers does admit roots of unity of any order, and our algorithms can be used for fixed point approximations of complex numbers at any precision.

Multi-point evaluation has several interesting applications, but it is important to keep in mind that our speed-ups only apply when the moduli are fixed. For instance, assume that we computed approximate zeros $z_1, ..., z_\ell$ to a complex polynomial of degree ℓ, using a bit-precision n. Then we may use multi-point evaluation in order to apply Newton's method simultaneously to all roots and find approximations of bit-precision $\approx 2\,n$. Unfortunately, our speed-up does not work in this case, since the approximate zeros $z_1, ..., z_\ell$ are not fixed. On the other hand, if the polynomial has degree $k\,\ell$ instead of ℓ and we are still given ℓ approximate zeros z_1, ..., z_ℓ (say all zeros in some disk), then the same moduli are used k times, and one may hope for some speed-up when using our methods.

At another extremity, it is instructive to consider the approximate evaluation of a fixed polynomial $P = P_0 + \cdots + P_{\ell-1}\,x^{\ell-1}$ with fixed point coefficients P_0, ..., $P_{\ell-1} \in \mathbb{D}_{n,0}$ at a single point $a \in \mathbb{D}_{n,0}$. We may thus assume that suitable Fourier transforms of the P_i have been precomputed. Now we rewrite $P = P_{[0]} + \cdots + P_{[d-1]}\,(x^k)^{d-1}$ with $k = \lceil \sqrt{\ell}\,\rceil$, $d = \lceil \ell/k \rceil$ and $P_{[i]} = P_{ki} + \cdots + P_{ki+k-1}\,x^{k-1}$. In order to evaluate each of the $P_{[i]}$ at a, it suffices to transform the numbers 1, x, ..., x^{k-1}, to perform the evaluations in the Fourier representation and then transform the results back. This can be achieved in time $O(k\,\mathsf{I}(n)) + k\,d\,\mathsf{N}(2\,n) + d\,\mathsf{F}(2\,n)$. We may finally compute $P(a) = P_{[0]}(a) + \cdots + P_{[d-1]}(a)\,(a^k)^{d-1}$ using Horner's method, in time $O(d\,\mathsf{I}(n))$. For large ℓ, the dominant term of the computation time is $k\,d\,\mathsf{N}(2\,n) \sim \ell\,\mathsf{N}(2\,n)$.

4. Gentle moduli

4.1. The base algorithms revisited for special moduli

Let us now reconsider the naive algorithms from section 3.2, but in the case when the moduli $m_1, ..., m_\ell$ are all close to a specific power of two. More precisely, we assume that

$$m_i \;=\; 2^{sw} + \delta_i \qquad (i = 1, ..., \ell),$$

where $|\delta_i| \leqslant 2^{w-1}$ and $s \geqslant 2$ a small number. As usual, we assume that the m_i are pairwise coprime and we let $M = m_1 \cdots m_\ell$.

For such moduli, the naive algorithm for the euclidean division of a number $x \in \mathcal{R}_{2^{\ell s w}}$ by m_i becomes particularly simple and essentially boils down to the multiplication of δ_i with the quotient of this division. In other words, the remainder can be computed in time $\ell\, s\, \mathsf{I}(w) + O(\ell\, s\, w)$ instead of $\mathsf{D}(\ell\, s\, w, s\, w)$. For small values of ℓ, s and w, this gives rise to a speedup by a factor s at least. More generally, the computation of ℓ remainders $a_1 = x \operatorname{rem} m_1, ..., a_\ell = x \operatorname{rem} m_\ell$ can be done in time $\ell^2\, s\, \mathsf{I}(w) + O(\ell^2\, s\, w)$.

Multi-modular reconstruction can also be done faster, as follows, using a similar technique as in [5]. Let $x \in \mathcal{R}_M$. Besides the usual binary representation of x and the multi-modular representation $(a_1, ..., a_\ell) = (x \operatorname{rem} m_1, ..., x \operatorname{rem} m_\ell)$, it is also possible to use the *Newton representation*

$$x \;=\; b_1 + b_2\, m_1 + b_3\, m_1\, m_2 + \cdots + b_\ell\, m_1 \cdots m_{\ell-1},$$

where $b_i \in \mathcal{R}_{m_i}$. Let us now show how to obtain $(b_1, ..., b_\ell)$ efficiently from $(a_1, ..., a_\ell)$. Since $x \operatorname{rem} m_1 = b_1 = a_1$, we must take $b_1 = a_1$. Assume that $b_1, ..., b_{i-1}$ have been computed. For $j = i-1, ..., 1$ we next compute

$$u_{j,i} \;=\; (b_j + b_{j+1}\, m_j + \cdots + b_{i-1}\, m_j \cdots m_{i-2}) \operatorname{rem} m_i$$

using $u_{i-1,i} = b_{i-1}$ and

$$\begin{aligned} u_{j,i} &= (b_j + u_{j+1,i}\, m_j) \operatorname{rem} m_i \\ &= (b_j + u_{j+1,i} \cdot (\delta_j - \delta_i)) \operatorname{rem} m_i \qquad (j = i-2, ..., 1). \end{aligned}$$

Notice that $u_{i-1,i}, ..., u_{1,i}$ can be computed in time $(i-1)\,(s+1)\,\mathsf{I}(w) + O(i\, s\, w)$. We have

$$x \operatorname{rem} m_i \;=\; (u_{1,i} + b_i\, m_1 \cdots m_{i-1}) \operatorname{rem} m_i \;=\; a_i.$$

Now the inverse v_i of $m_1 \cdots m_{i-1}$ modulo m_i can be precomputed. We finally compute

$$b_i \;=\; v_i\, (a_i - u_1) \operatorname{rem} m_i,$$

which can be done in time $\mathsf{I}(s\, w) + O(s\, w)$. For small values of i, we notice that it may be faster to divide successively by $m_1, ..., m_{i-1}$ modulo m_i instead of multiplying with v_i. In total, the computation of the Newton representation $(b_1, ..., b_\ell)$ can be done in time $\binom{\ell}{2}\,(s+1)\,\mathsf{I}(w) + \mathsf{I}(s\, w)\,\ell + O(\ell^2\, s\, w)$. Having computed the Newton representation, we next compute

$$x_i \;=\; b_i + b_{i+1}\, m_i + \cdots + b_\ell\, m_i \cdots m_{\ell-1}$$

for $i = \ell, ..., 1$, using the recurrence relation

$$x_i \;=\; b_i + x_{i+1}\, m_i.$$

Since $x_{i+1} \in \mathcal{R}_{2^{(\ell-i)sw}}$, the computation of x_i takes a time $(l-i)\,s\,\mathsf{I}(w) + O((l-i)\,s\,w)$. Altogether, the computation of $x = x_1$ from $(a_1, ..., a_\ell)$ can therefore be done in time $\binom{\ell}{2}(2s+1)\,\mathsf{I}(w) + \mathsf{I}(sw)\,\ell + O(\ell^2 s\,w) \approx \ell^2 s\,\mathsf{I}(w)$.

4.2. The gentle modulus hunt

For practical applications, we usually wish to work with moduli that fit into one word or half a word. Now the algorithm from the previous subsection is particularly efficient if the numbers δ_i also fit into one word or half a word. This means that we need to impose the further requirement that each modulus m_i can be factored

$$m_i \;=\; m_{i,1} \cdots m_{i,s},$$

with $m_{i,1}, ..., m_{i,s} < 2^w$. If this is possible, then the m_i are called *s-gentle moduli*. For given bitsizes w and $s \geqslant 2$, the main questions are now: do such moduli indeed exist? If so, then how to find them?

If $s = 2$, then it is easy to construct s-gentle moduli $m_i = 2^{2w} + \delta_i$ by taking $\delta_i = -\varepsilon_i^2$, where $0 \leqslant \varepsilon_i < 2^{(w-1)/2}$ is odd. Indeed,

$$2^{2w} - \varepsilon_i^2 \;=\; (2^w + \varepsilon_i)\,(2^w - \varepsilon_i)$$

and $\gcd(2^w + \varepsilon_i,\, 2^w - \varepsilon_i) = \gcd(2^w + \varepsilon_i,\, 2\,\varepsilon_i) = \gcd(2^w + \varepsilon_i,\, \varepsilon_i) = \gcd(2^w,\, \varepsilon_i) = 1$. Unfortunately, this trick does not generalize to higher values $s \geqslant 3$. Indeed, consider a product

$$\begin{aligned}(2^w + \eta_1) \cdots (2^w + \eta_s) \;&=\; 2^{sw} + (\eta_1 + \cdots + \eta_s)\,2^{(s-1)w} + \\ &\quad ((\eta_1 + \cdots + \eta_s)^2 - (\eta_1^2 + \cdots + \eta_s^2))\,2^{(s-2)w-1} + \cdots,\end{aligned}$$

where $\eta_1, ..., \eta_s$ are small compared to 2^w. If the coefficient $\eta_1 + \cdots + \eta_s$ of $2^{(s-1)w}$ vanishes, then the coefficient of $2^{(s-2)w-1}$ becomes the opposite $-(\eta_1^2 + \cdots + \eta_s^2)$ of a sum of squares. In particular, both coefficients cannot vanish simultaneously, unless $\eta_1 = \cdots = \eta_s = 0$.

If $s > 2$, then we are left with the option to search s-gentle moduli by brute force. As long as s is "reasonably small" (say $s \leqslant 8$), the probability to hit an s-gentle modulus for a randomly chosen δ_i often remains significantly larger than 2^{-w}. We may then use sieving to find such moduli. By what precedes, it is also desirable to systematically take $\delta_i = -\varepsilon_i^2$ for $0 \leqslant \varepsilon_i < 2^{(w-1)/2}$. This has the additional benefit that we "only" have to consider $2^{(w-1)/2}$ possibilities for ε_i.

4.3. The sieving procedure

We implemented a sieving procedure in MATHEMAGIX [18] that uses the MPARI package with an interface to PARI-GP [22]. Given parameters s, w, w' and μ, the goal of our procedure is to find s-gentle moduli of the form

$$M \;=\; (2^{sw/2} - \varepsilon)\,(2^{sw/2} - \varepsilon) \;=\; m_1 \cdots m_s$$

with the constraints that

$$\begin{aligned} m_i \;&<\; 2^{w'} \\ \gcd(m_i, 2^\mu!) \;&=\; 1, \end{aligned}$$

for $i = 1, ..., s$, and $m_1 \leqslant \cdots \leqslant m_s$. The parameter s is small and even. One should interpret w and w' as the intended and maximal bitsize of the small moduli m_i. The parameter μ stands for the minimal bitsize of a prime factor of m_i. The parameter ε should be such that $4\,\varepsilon^2$ fits into a machine word.

In Table 1 below we have shown some experimental results for this sieving procedure in the case when $s=6$, $w=22$, $w'=25$ and $\mu=4$. For $\varepsilon<1000000$, the table provides us with ε, the moduli $m_1,\ldots,m_s$, as well as the smallest prime power factors of the product M. Many hits admit small prime factors, which increases the risk that different hits are not coprime. For instance, the number 17 divides both $2^{132}-311385^2$ and $2^{132}-376563^2$, whence these 6-gentle moduli cannot be selected simultaneously (except if one is ready to sacrifice a few bits by working modulo $\operatorname{lcm}(2^{132}-311385^2, 2^{132}-376563^2)$ instead of $(2^{132}-311385^2)\cdot(2^{132}-376563^2)$).

In the case when we use multi-modular arithmetic for computations with rational numbers instead of integers (see [10, section 5 and, more particularly, section 5.10]), then small prime factors should completely be prohibited, since they increase the probability of divisions by zero. For such applications, it is therefore desirable that $m_1,\ldots,m_s$ are all prime. In our table, this occurs for $\varepsilon=57267$ (we indicated this by highlighting the list of prime factors of M).

In order to make multi-modular reduction and reconstruction as efficient as possible, a desirable property of the moduli m_i is that they either divide $2^{sw/2}-\varepsilon$ or $2^{sw/2}+\varepsilon$. In our table, we highlighted the ε for which this happens. We notice that this is automatically the case if $m_1,\ldots,m_s$ are all prime. If only a small number of m_i (say a single one) do not divide either $2^{sw/2}-\varepsilon$ or $2^{sw/2}+\varepsilon$, then we remark that it should still be possible to design reasonably efficient *ad hoc* algorithms for multi-modular reduction and reconstruction.

Another desirable property of the moduli $m_1\leqslant\cdots\leqslant m_s$ is that m_s is as small as possible: the spare bits can for instance be used to speed up matrix multiplication modulo m_s. Notice however that one "occasional" large modulus m_s only impacts on one out of s modular matrix products; this alleviates the negative impact of such moduli. We refer to section 4.5 below for more details.

For actual applications, one should select gentle moduli that combine all desirable properties mentioned above. If not enough such moduli can be found, then it it depends on the application which criteria are most important and which ones can be released.

ε	m_1	m_2	m_3	m_4	m_5	m_6	$p_1^{\nu_1}, p_2^{\nu_2}, \ldots$
27657	28867	4365919	6343559	13248371	20526577	25042063	$29, 41, 43, 547, \ldots$
57267	416459	1278617	2041469	6879443	25754563	28268089	$416459, \ldots$
77565	7759	8077463	8261833	18751793	19509473	28741799	$59, 641, \ldots$
95253	724567	965411	3993107	4382527	19140643	23236813	$43, 724567, \ldots$
294537	190297	283729	8804561	19522819	19861189	29537129	$23^2, 151, 1879, \ldots$
311385	145991	4440391	4888427	6812881	7796203	32346631	$17, 79, 131, \ldots$
348597	114299	643619	6190673	11389121	32355397	32442427	$31, 277, \ldots$
376563	175897	1785527	2715133	7047419	30030061	30168739	$17, 127, 1471, \ldots$
462165	39841	3746641	7550339	13195943	18119681	20203643	$67, 641, 907, \ldots$
559713	353201	873023	2595031	11217163	18624077	32569529	$19, 59, 14797, \ldots$
649485	21727	1186571	14199517	15248119	31033397	31430173	$19, 109, 227, \ldots$
656997	233341	1523807	5654437	8563679	17566069	18001723	$79, 89, 63533, \ldots$
735753	115151	923207	3040187	23655187	26289379	27088541	$53, 17419, \ldots$
801687	873767	1136111	3245041	7357871	8826871	26023391	$23, 383777, \ldots$
826863	187177	943099	6839467	11439319	12923753	30502721	$73, 157, 6007, \ldots$
862143	15373	3115219	11890829	18563267	19622017	26248351	$31, 83, 157, \ldots$
877623	514649	654749	4034687	4276583	27931549	33525223	$41, 98407, \ldots$
892455	91453	2660297	3448999	12237457	21065299	25169783	$29, 397, 2141, \ldots$

Table 1. List of 6-gentle moduli for $w=22$, $w'=25$, $\mu=4$ and $\varepsilon<1000000$.

4.4. Influence of the parameters s, w and w'

Ideally speaking, we want s to be as large as possible. Furthermore, in order to waste as few bits as possible, w' should be close to the word size (or half of it) and $w' - w$ should be minimized. When using double precision floating point arithmetic, this means that we wish to take $w' \in \{24, 25, 26, 50, 51, 52\}$. Whenever we have efficient hardware support for integer arithmetic, then we might prefer $w \in \{30, 31, 32, 62, 63, 64\}$.

Let us start by studying the influence of $w' - w$ on the number of hits. In Table 2, we have increased w by one with respect to Table 1. This results in an approximate 5% increase of the "capacity" $s\,w$ of the modulus M. On the one hand, we observe that the hit rate of the sieve procedure roughly decreases by a factor of thirty. On the other hand, we notice that the rare gentle moduli that we do find are often of high quality (on four occasions the moduli $m_1, ..., m_s$ are all prime in Table 2).

ε	m_1	m_2	m_3	m_4	m_5	m_6	$p_1^{\nu_1}, p_2^{\nu_2}, ...$
936465	543889	4920329	12408421	15115957	24645539	28167253	$19, 59, 417721, ...$
2475879	867689	4051001	11023091	13219163	24046943	28290833	$867689, ...$
3205689	110161	12290741	16762897	22976783	25740731	25958183	$59, 79, 509, ...$
3932205	4244431	5180213	5474789	8058377	14140817	25402873	$4244431, ...$
5665359	241739	5084221	18693097	21474613	23893447	29558531	$31, 41, 137, ...$
5998191	30971	21307063	21919111	22953967	31415123	33407281	$101, 911, 941, ...$
6762459	3905819	5996041	7513223	7911173	8584189	29160587	$43, 137, 90833, ...$
9245919	2749717	4002833	8274689	9800633	15046937	25943587	$2749717, ...$
9655335	119809	9512309	20179259	21664469	22954369	30468101	$17, 89, 149, ...$
12356475	1842887	2720359	7216357	13607779	23538769	30069449	$1842887, ...$
15257781	1012619	5408467	9547273	11431841	20472121	28474807	$31, 660391, ...$

Table 2. List of 6-gentle moduli for $w = 23$, $w' = 25$, $\mu = 4$ and $\varepsilon < 16000000$.

Without surprise, the hit rate also sharply decreases if we attempt to increase s. The results for $s = 8$ and $w = 22$ are shown in Table 3. A further infortunate side effect is that the quality of the gentle moduli that we do find also decreases. Indeed, on the one hand, M tends to systematically admit at least one small prime factor. On the other hand, it is rarely the case that each m_i divides either $2^{sw/2} - \varepsilon$ or $2^{sw/2} + \varepsilon$ (this might nevertheless be the case for other recombinations of the prime factors of M, but only modulo a further increase of m_s).

ε	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	$p_1^{\nu_1}, p_2^{\nu_2}, ...$
889305	50551	1146547	4312709	5888899	14533283	16044143	16257529	17164793	$17, 31, 31, 59, ...$
2447427	53407	689303	3666613	4837253	7944481	21607589	25976179	32897273	$31, 61, 103, ...$
2674557	109841	1843447	2624971	5653049	7030883	8334373	18557837	29313433	$103, 223, 659, ...$
3964365	10501	2464403	6335801	9625841	10329269	13186219	17436197	25553771	$23, 163, 607, ...$
4237383	10859	3248809	5940709	6557599	9566959	11249039	22707323	28518509	$23, 163, 1709, ...$
5312763	517877	616529	879169	4689089	9034687	11849077	24539909	27699229	$43, 616529, ...$
6785367	22013	1408219	4466089	7867589	9176941	12150997	26724877	29507689	$23, 41, 197, ...$
7929033	30781	730859	4756351	9404807	13807231	15433939	19766077	22596193	$31, 307, 503, ...$
8168565	10667	3133103	3245621	6663029	15270019	18957559	20791819	22018021	$43, 409, 467, ...$
8186205	41047	2122039	2410867	6611533	9515951	14582849	16507739	30115277	$23, 167, 251, ...$

Table 3. List of 8-gentle moduli for $w = 22$, $w' = 25$, $\mu = 4$ and $\varepsilon < 10000000$.

An increase of w' while maintaining s and $w' - w$ fixed also results in a decrease of the hit rate. Nevertheless, when going from $w' = 25$ (floating point arithmetic) to $w' = 31$ (integer arithmetic), this is counterbalanced by the fact that ε can also be taken larger (namely $\varepsilon < 2^{w'}$); see Table 4 for a concrete example. When doubling w and w' while keeping the same upper bound for ε, the hit rate remains more or less unchanged, but the rate of high quality hits tends to decrease somewhat: see Table 5.

It should be possible to analyze the hit rate as a function of the parameters s, w, w' and μ from a probabilistic point of view, using the idea that a random number n is prime with probability $(\log n)^{-1}$. However, from a practical point of view, the priority is to focus on the case when $w' \leqslant 64$. For the most significant choices of parameters $\mu < w < w' \leqslant 64$ and s, it seems in principle to be possible to compile full tables of s-gentle moduli. Unfortunately, our current implementation is still somewhat inefficient for $w' > 32$. A helpful feature for upcoming versions of PARI would be a function to find all prime factors of an integer below a specified maximum $2^{w'}$ (the current version only does this for prime factors that can be tabulated).

ε	m_1	m_2	m_3	m_4	m_5	m_6	$p_1^{\nu_1}, p_2^{\nu_2}, \ldots$
303513	42947057	53568313	331496959	382981453	1089261409	1176003149	$29^2, 1480933, \ldots$
851463	10195123	213437143	470595299	522887483	692654273	1008798563	$17, 41, 67, \ldots$
1001373	307261	611187931	936166801	1137875633	1196117147	1563634747	$47, 151, \ldots$
1422507	3950603	349507391	490215667	684876553	693342113	1164052193	$29, 211, 349, \ldots$
1446963	7068563	94667021	313871791	877885639	1009764377	2009551553	$23, 71, 241, \ldots$
1551267	303551	383417351	610444753	1178193077	2101890797	2126487631	$29, 43, 2293, \ldots$
1555365	16360997	65165071	369550981	507979403	1067200639	1751653069	$17, 23, 67, \ldots$
4003545	20601941	144707873	203956547	624375041	655374931	1503716491	$47, 67, \ldots$
4325475	11677753	139113383	210843443	659463289	936654347	1768402001	$19, 41, \ldots$
4702665	8221903	131321017	296701997	496437899	1485084431	1584149417	$8221903, \ldots$
5231445	25265791	49122743	433700843	474825677	907918279	1612324823	$17, 1486223, \ldots$
5425527	37197571	145692101	250849363	291039937	456174539	2072965393	$37197571, \ldots$
6883797	97798097	124868683	180349291	234776683	842430863	858917923	$97798097, \ldots$
7989543	4833137	50181011	604045619	638131951	1986024421	2015143349	$23, 367, \ldots$

Table 4. List of 6-gentle moduli for $w = 28$, $w' = 31$, $\mu = 4$ and $\varepsilon < 1600000$. Followed by some of the next gentle moduli for which each m_i divides either $2^{sw/2} - \alpha$ or $2^{sw/2} + \alpha$.

ε	m_1	m_2	$\cdots$	m_5	m_6	$p_1^{\nu_1}, p_2^{\nu_2}, \ldots$
15123	380344780931	774267432193	$\cdots$	463904018985637	591951338196847	$37, 47, 239, \ldots$
34023	9053503517	13181369695139	$\cdots$	680835893479031	723236090375863	$29, 35617, \ldots$
40617	3500059133	510738813367	$\cdots$	824394263006533	1039946916817703	$23, 61, 347, \ldots$
87363	745270007	55797244348441	$\cdots$	224580313861483	886387548974947	$71, 9209, \ldots$
95007	40134716987	2565724842229	$\cdots$	130760921456911	393701833767607	$19, 67, \ldots$
101307	72633113401	12070694419543	$\cdots$	95036720090209	183377870340761	$41, 401, \ldots$
140313	13370367761	202513228811	$\cdots$	397041457462499	897476961701171	$379, 1187, \ldots$
193533	35210831	15416115621749	$\cdots$	727365428298107	770048329509499	$59, 79, \ldots$
519747	34123521053	685883716741	$\cdots$	705516472454581	836861326275781	$127, 587, \ldots$
637863	554285276371	1345202287357	$\cdots$	344203886091451	463103013579761	$79, 1979, \ldots$
775173	322131291353	379775454593	$\cdots$	194236314135719	1026557288284007	$322131291353, \ldots$
913113	704777248393	1413212491811	$\cdots$	217740328855369	261977228819083	$37, 163, 677, \ldots$
1400583	21426322331	42328735049	$\cdots$	411780268096919	626448556280293	$21426322331, \ldots$

Table 5. List of 6-gentle moduli for $w = 44$, $w' = 50$, $\mu = 4$ and $\varepsilon < 200000$. Followed by some of the next gentle moduli for which each m_i divides either $2^{sw/2} - \alpha$ or $2^{sw/2} + \alpha$.

4.5. Application to matrix multiplication

One of our favourite applications of multi-modular arithmetic is the multiplication of integer matrices $A, B \in \mathbb{Z}^{r \times r}$. We proceed as follows:

1. Compute $A \operatorname{rem} m_i$ and $B \operatorname{rem} m_i$ for $i = 1, \ldots, \ell$, using $2\, r^2$ multi-modular reductions.

2. Multiply $C \operatorname{rem} m_i := (A \operatorname{rem} m_i)\,(B \operatorname{rem} m_i) \operatorname{rem} m_i$ for $i = 1, ..., \ell$.

3. Reconstruct $C \operatorname{rem} M$ using r^2 multi-modular reconstructions.

If M is larger than $2\,|(A\,B)_{i,j}|$ for all i and j, then $A\,B$ can be read off from $A\,B \operatorname{rem} M$.

From a practical point of view, the second step can be implemented very efficiently if $r\,m_i^2$ fits into the size of a word. When using floating point arithmetic, this means that we should have $r\,m_i^2 < 2^{52}$ for all i. For large values of r, this is unrealistic; in that case, we subdivide the $r \times r$ matrices into smaller $r_i \times r_i$ matrices with $r_i\,m_i^2 < 2^{52}$. The fact that r_i may depend on i is very significant. First of all, the larger we can take r_i, the faster we can multiply matrices modulo m_i. Secondly, the m_i in the tables from the previous sections often vary in bitsize. It frequently happens that we may take all r_i large except for the last modulus m_ℓ. The fact that matrix multiplications modulo the worst modulus m_ℓ are somewhat slower is compensated by the fact that they only account for one out of every ℓ modular matrix products.

Several of the tables in the previous subsections were made with the application to integer matrix multiplication in mind. Consider for instance the modulus $M = m_1 \cdots m_6 = 2^{132} - 656997^2$ from Table 1. When using floating point arithmetic, we obtain $r_1 \leqslant 82713$, $r_2 \leqslant 1939$, $r_3 \leqslant 140$, $r_4 \leqslant 61$, $r_5 \leqslant 14$ and $r_6 \leqslant 13$. Clearly, there is a trade-off between the efficiency of the modular matrix multiplications (high values of r_i are better) and the bitsize $\approx \ell\,w$ of M (larger capacities are better).

If r is large with respect to $\log^2 M$, then the modular matrix multiplication step is the main bottleneck, so it is important to take all m_i approximately of the same size (i.e. $w' - w$ should be small) and in such a way that the corresponding r_i lead to the best complexity ($\log_2 r_i \approx 6$ tends to work well). This can often only be achieved by lowering s to $s = 4$ or $s = 2$. For r closer to $\log^2 M$, the Chinese remaindering steps become more and more expensive, which makes it interesting to consider larger values of s and to increase the difference $w' - w$. For r significantly below $\log^2 M$, we resort to FFT-based matrix multiplication. This corresponds to taking roughly twice as many moduli, but the transformations become approximately $\log \log M$ times less expensive.

4.6. Implementation issues

We have not yet implemented any of the algorithms in this paper. The implementation that we envision selects appropriate code as a function of the parameters w, w' and ℓ. The parameters w and w' highly depend on the application: see the above discussion in the case of integer matrix multiplication. Generally speaking, w' is bounded by the bitsize W of a machine word or by $W/2$.

So far, we have described four main strategies for solving Chinese remaindering problems for moduli $M = m_1 \cdots m_\ell$:

G. The "gentle modulus strategy" requires $M = 2^{ws} - \varepsilon^2$ to be an s-gentle modulus with $s = \ell$. Conversions between $x \operatorname{rem} M$ and $(x \operatorname{rem} (2^{ws/2} - \varepsilon), x \operatorname{rem} (2^{ws} + \varepsilon))$ can be done fast. The further conversions from and to $(x \operatorname{rem} m_1, ..., x \operatorname{rem} m_s)$ are done in a naive manner.

P. The "gentle product strategy" strategy requires $m_1, ..., m_\ell$ to be gentle moduli. We now perform the multi-modular reductions and recombinations using the algorithms from subsection 4.1.

N. The "naive strategy" for Chinese remaindering has been described in subsections 3.2 and 3.3.

F. The asymptotically fast "FFT-based strategy" from Theorems 6 and 10, which attempts to do as much work as possible using Fourier representations.

The idea is now to apply each of these strategies at the appropriate levels of the remainder tree. At the very bottom, we use **G**, followed by **P** and **N**. At the top levels, we use **F**.

As a function of ℓ, we need to decide how much work we wish to perform at each of these levels. For small $\ell \lesssim 64$, it suffices to combine the strategies **G** and **P**. As soon as $s\,w'/2$ exceeds W, some of the modular reductions in the strategy **G** may become expensive. For this reason, it is generally better to let **P** do a bit more work, i.e. to take $s^2 < \ell$. It is also a good practice to use w as the "soft word size" for multiple integer arithmetic (see subsection 2.4).

As soon as ℓ starts to get somewhat larger, say $64 \lesssim \ell \lesssim 256$, then some intermediate levels may be necessary before that FFT multiplication becomes plainly efficient. For these levels, we use strategy **N** with arity two. It still has to be found out how large this intermediate region is, exactly. Indeed, the ability to do more work using the Fourier representation often lowers the threshold at which such methods become efficient. If one manages to design extremely efficient implementations for the strategies **G** and **P** (e.g. if multi-modular reduction and reconstruction can approximately be done as fast as multiplication itself), then one may also consider the use of Chinese remaindering instead of Fourier transforms in **F**.

For large $\ell \gtrsim 256$, the FFT-based algorithms should become fastest and we expect that the theoretical $\log\log(w\,\ell)$ speed-up should result in practical gains of a factor three at least. As emphasized before, it is crucial to rely on inborn FFT strategies. For very large ℓ, we may also run out of s-gentle moduli. In that case, we may need to resort to lower values of s, with the consequence that some of the lower levels may become somewhat more expensive.

4.7. Alternative moduli

A very intriguing question is whether it is possible to select moduli that allow for even faster Chinese remaindering. In the analogue case of polynomials, highly efficient algorithms exist for multi-point evaluation and interpolation at points that form a geometric sequence [5]. Geometric sequences of moduli do not work in our case, since they violate the mutual coprime requirement and they grow too fast anyway. Arithmetic progressions are more promising, although only algorithms with a better constant factor are known in the polynomial case [5], and the elements of such sequences generically admit small prime factors in common that have to be treated with care.

Another natural idea is to chose products $M = m_1 \cdots m_\ell$ of the form $M = 2^n - 1$, where n is highly composite. Each divisor d of n naturally induces a divisor $\Phi_d(2)$ of M, where Φ_d denotes the d-th cyclotomic polynomial. For instance, the number $2^{6\cdot 60} - 1$ is divisible by $\Phi_1(2^{60}) = 2^{60} - 1$, $\Phi_2(2^{60}) = 2^{60} + 1$, $\Phi_3(2^{60}) = 2^{120} + 2^{60} + 1$ and $\Phi_6(2^{60}) = 2^{120} - 2^{60} + 1$. Now euclidean division by $\Phi_1(2^{60}), \Phi_2(2^{60}), \Phi_3(2^{60})$ and $\Phi_6(2^{60})$ is easy since these numbers admit an extremely sparse and regular binary representations. Furthermore, the numbers $\Phi_3(2^{60}) = m_3\,m_4$ and $\Phi_6(2^{60}) = m_5\,m_6$ can both be factored into products of two integers of less than 64 bits. Taking $m_1 = 2^{60} - 1$ and $m_2 = 2^{60} + 1$, it should therefore be possible to design a reasonably efficient Chinese remaindering scheme for $M = m_1\,m_2\,m_3\,m_4\,m_5\,m_6$ on a 64-bit architecture.

There are several downsides to this approach. Most importantly, the largest prime divisor $\lambda(n)$ of 2^n-1 grows quite quickly with n, even if n is very smooth. For instance, the four largest n for which $\lambda(n)<2^{32}$ are $n=180, 200, 204, 210$ and the four largest n for which $\lambda(n)<2^{64}$ are $n=420, 440, 540, 648$. By construction, the number 2^n-1 also admits many divisors if n is smooth, so several moduli of this type are generally not coprime. Both obstructions together imply that we can only construct Chinese remaindering schemes with a small number of moduli in this way. For instance, on a 64 bit architecture, one of the best schemes would use $n=648$ and twelve moduli of an average size of 54 bits. In comparison, there are many 4-gentle and 6-gentle moduli that are products of prime numbers of approximately 54 bits (or more), and combining a few of them leads to efficient Chinese remaindering schemes for twelve (or more) prime moduli of approximately 54 bits.

BIBLIOGRAPHY

[1] D. Bernstein. Removing redundancy in high precision Newton iteration. Available from http://cr.yp.to/fastnewton.html, 2000.

[2] D. Bernstein. Scaled remainder trees. Available from https://cr.yp.to/arith/scaledmod-20040820.pdf, 2004.

[3] A. Borodin and R.T. Moenck. Fast modular transforms. *Journal of Computer and System Sciences*, 8:366–386, 1974.

[4] A. Bostan, G. Lecerf, and É. Schost. Tellegen's principle into practice. In *Proceedings of ISSAC 2003*, pages 37–44. ACM Press, 2003.

[5] A. Bostan and É. Schost. Polynomial evaluation and interpolation on special sets of points. *Journal of Complexity*, 21(4):420–446, August 2005. Festschrift for the 70th Birthday of Arnold Schönhage.

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

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

[8] C.M. Fiduccia. Polynomial evaluation via the division algorithm: the fast fourier transform revisited. In A.L. Rosenberg, editor, *Fourth annual ACM symposium on theory of computing*, pages 88–93, 1972.

[9] M. Fürer. Faster integer multiplication. *SIAM J. Comp.*, 39(3):979–1005, 2009.

[10] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 3rd edition, 2013.

[11] T. Granlund et al. GMP, the GNU multiple precision arithmetic library. http://www.swox.com/gmp, 1991.

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

[13] D. Harvey and J. van der Hoeven. On the complexity of integer matrix multiplication. Technical report, HAL, 2014. http://hal.archives-ouvertes.fr/hal-01071191, accepted for publication in JSC.

[14] D. Harvey, J. van der Hoeven, and G. Lecerf. Faster polynomial multiplication over finite fields. Technical report, ArXiv, 2014. http://arxiv.org/abs/1407.3361, accepted for publication in JACM.

[15] D. Harvey, J. van der Hoeven, and G. Lecerf. Even faster integer multiplication. *Journal of Complexity*, 36:1–30, 2016.

[16] J. van der Hoeven. Newton's method and FFT trading. *JSC*, 45(8):857–878, 2010.

[17] J. van der Hoeven and G. Lecerf. Faster FFTs in medium precision. In *22nd Symposium on Computer Arithmetic (ARITH)*, pages 75–82, June 2015.

[18] J. van der Hoeven, G. Lecerf, B. Mourrain, et al. Mathemagix, 2002. http://www.mathemagix.org.

[19] J. van der Hoeven, G. Lecerf, and G. Quintin. Modular SIMD arithmetic in Mathemagix. *ACM Trans. Math. Softw.*, 43(1):5:1–5:37, 2016.

[20] R.T. Moenck and A. Borodin. Fast modular transforms via division. In *Thirteenth annual IEEE symposium on switching and automata theory*, pages 90–96, Univ. Maryland, College Park, Md., 1972.

[21] C.H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[22] The PARI Group, Bordeaux. *PARI/GP*, 2012. Software available from http://pari.math.u-bordeaux.fr.

[23] J.M. Pollard. The fast Fourier transform in a finite field. *Mathematics of Computation*, 25(114):365–374, 1971.

[24] A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7:281–292, 1971.