Fast Chinese remaindering in practice

by 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

October 24, 2017

Abstract

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 $m_1, \ldots, m_\ell$ are fixed and can even be chosen by the user. If ℓ is small or moderately large, then we show how to choose *gentle moduli* that allow for speedier Chinese remaindering. The multiplication of integer matrices is one typical application where we expect practical gains for various common matrix dimensions and bitsizes of the coefficients.

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

1 Introduction

Modular reduction is an important tool for speeding up computations in computer arithmetic, symbolic computation, and elsewhere. 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 [9, 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. Given $a, m \in \mathbb{Z}$ with $m > 1$, we will denote by $a \operatorname{rem} m \in \mathcal{R}_m := \{0, \ldots, m-1\}$ the remainder of the Euclidean division of a by m. Given an $r \times r$ matrix $A \in \mathbb{Z}^{r \times r}$ with integer coefficients, we will also denote $A \operatorname{rem} m \in \mathbb{Z}^{r \times r}$ for the matrix with coefficients $(A \operatorname{rem} m)_{i,j} = A_{i,j} \operatorname{rem} m$.

One typical application of Chinese remaindering is the multiplication of $r \times r$ integer matrices A, $B \in \mathbb{Z}^{r \times r}$. Assuming that we have a bound M with $2\,|(A\,B)_{i,j}| < M$ for all i, j, we proceed as follows:

0. Select moduli $m_1, \ldots, m_\ell$ with $m_1 \cdots m_\ell > M$ that are mutually coprime.
1. Compute $A \operatorname{rem} m_k$ and $B \operatorname{rem} m_k$ for $k = 1, \ldots, \ell$.
2. Multiply $C \operatorname{rem} m_k := (A \operatorname{rem} m_k)\,(B \operatorname{rem} m_k) \operatorname{rem} m_k$ for $k = 1, \ldots, \ell$.

3. Reconstruct $C \operatorname{rem} M$ from the $C \operatorname{rem} m_k$ with $k = 1, \ldots, \ell$.

The simultaneous computation of $A_{i,j} \operatorname{rem} m_k$ from $A_{i,j}$ for all $k = 1, \ldots, \ell$ is called the problem of *multi-modular reduction*. In step 1, we need to perform $2\, r^2$ multi-modular reductions for the coefficients of A and B. The inverse problem of reconstructing $C_{i,j} \operatorname{rem} M$ from the $C_{i,j} \operatorname{rem} m_k$ with $k = 1, \ldots, \ell$ is called the problem of *multi-modular reconstruction*. We need to perform r^2 such reconstructions in step 3. Our hypothesis on M allows us to recover C from $C \operatorname{rem} M$.

Let us quickly examine when and why the above strategy pays off. In this paper, the number ℓ should be small or moderately large, say $\ell \leqslant 64$. The moduli $m_1, \ldots, m_\ell$ typically fit into a machine word. Denoting by μ the bitsize of a machine word (say $\mu = 32$ or $\mu = 64$), the coefficients of A and B should therefore be of bitsize $\approx \ell\, \mu / 2$.

For small ℓ, integer multiplications of bitsize $\mu\, \ell / 2$ are usually carried out using a naive algorithm, of complexity $\Theta(\ell^2)$. If we directly compute the product $A\, B$ using r^3 naive integer multiplications, the computation time is therefore of order $\Theta(r^3\, \ell^2)$. In comparison, as we will see, one naive multi-modular reduction or reconstruction for ℓ moduli roughly requires $\Theta(\ell^2)$ machine operations, whereas an $r \times r$ matrix multiplication modulo any of the m_k can be done in time $\Theta(r^3)$. Altogether, this means that the above multi-modular algorithm for integer matrix multiplication has running time $\Theta(\ell^2\, r^2 + r^3\, \ell)$, which is $\Theta(\min(\ell, r))$ times faster than the naive algorithm.

If $\ell \ll r$, then the cost $\Theta(\ell^2\, r^2)$ of steps 1 and 3 is negligible with respect to the cost $\Theta(r^3\, \ell)$ of step 2. However, if ℓ and r are of the same order of magnitude, then Chinese remaindering may take an important part of the computation time; the main purpose of this paper is to reduce this cost. If $\ell \gg r$, then we notice that other algorithms for matrix multiplication usually become faster, such as naive multiplication for small ℓ, Karatsuba multiplication [13] for larger ℓ, or FFT-based techniques [6] for very large ℓ.

Two observations are crucial for reducing the cost of Chinese remaindering. First of all, the moduli $m_1, \ldots, m_\ell$ are the same for all $2\, r^2$ multi-modular reductions and r^2 multi-modular reconstructions in steps 1 and 3. If r is large, then this means that we can essentially assume that $m_1, \ldots, m_\ell$ were fixed once and for all. Secondly, we are free to choose $m_1, \ldots, m_\ell$ in any way that suits us. We will exploit these observations by precomputing *gentle moduli* for which Chinese remaindering can be performed more efficiently than for ordinary moduli.

The first idea behind gentle moduli is to consider moduli m_i of the form $2^{sw} - \varepsilon_i^2$, where w is somewhat smaller than μ, where s is even, and $\varepsilon_i^2 < 2^w$. In section 3.1, 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 of bitsize $< \mu$. 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.

Let us briefly outline the structure of this paper. In section 2, we rapidly recall basic facts about Chinese remaindering and naive algorithms for this task. In section 3, we introduce gentle moduli and describe how to speed up Chinese remaindering with respect to such moduli. The last section 4 is dedicated to the brute force search of gentle moduli for specific values of s and w. We implemented a sieving method in MATHEMAGIX which allowed us to compute tables with gentle moduli. For practical purposes, it turns out that gentle moduli exist in sufficient number for $s \leqslant 8$. We expect our technique to be efficient for $\ell \lesssim s^2$, but this still needs to be confirmed *via* an actual implementation. The application to integer matrix multiplication in section 4.3 also has not been implemented yet.

Let us finally discuss a few related results. In this paper, we have chosen to highlight integer matrix multiplication as one typical application in computer algebra. Multi-modular methods are used in many other areas and the operations of multi-modular reduction and reconstruction are also known as conversions between the positional number system (PNS) and the residue number system (RNS). Asymptotically fast algorithms are based on *remainder trees* [8, 14, 3], with recent improvements in [4, 2, 10]; we expect such algorithms to become more efficient when ℓ exceeds s^2.

Special moduli of the form $2^n - \varepsilon$ are also known as *pseudo-Mersenne moduli*. They have been exploited before in cryptography [1] in a similar way as in section 3.1, but with a slightly different focus: whereas the authors of [1] are keen on reducing the number of additions (good for circuit complexity), we rather optimize the number of machine instructions on recent general purpose CPUs (good for software implementations). Our idea to choose moduli $2^n - \varepsilon$ that can be factored into smaller moduli is new.

Other algorithms for speeding up multiple multi-modular reductions and reconstructions for the same moduli (while allowing for additional pre-computations) have recently been proposed in [7]. These algorithms essentially replace all divisions by simple multiplications and can be used in conjunction with our new algorithms for conversions between residues modulo $m_i = m_{i,1} \cdots m_{i,s}$ and residues modulo $m_{i,1}, \ldots, m_{i,s}$.

2 Chinese remaindering

2.1 The Chinese remainder theorem

For any integer $m \geqslant 1$, we recall that $\mathcal{R}_m = \{0, \ldots, m-1\}$. For machine computations, it is convenient to use the following effective version of the Chinese remainder theorem:

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

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

satisfies $x \operatorname{rem} m_i = a_i$ *for* $i = 1, \ldots, \ell$.

Proof. For each $i = 1, \ldots, \ell$, let $\pi_i = M / m_i$. Since π_i and m_i are coprime, π_i admits an inverse u_i modulo m_i in $\mathcal{R}_{m_i}$. We claim that we may take $c_i = \pi_i u_i$. Indeed, for $x = (c_1 a_1 + \cdots + c_\ell a_\ell) \operatorname{rem} M$ and any $i \in \{1, \ldots, \ell\}$, we have

$$x \equiv a_1 \pi_1 u_1 + \cdots + a_\ell \pi_\ell u_\ell \pmod{m_i}$$

Since π_j is divisible by m_i for all $j \neq i$, this congruence relation simplifies into

$$x \equiv a_i \pi_i u_i \equiv a_i \pmod{m_i}.$$

This proves our claim and the theorem. □

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

2.2 Modular arithmetic

For practical computations, the moduli m_i are usually chosen such that they fit into single machine words. Let μ denote the bitsize of a machine word, so that we typically have $\mu = 32$ or $\mu = 64$. It depends on specifics of the processor how basic arithmetic operations modulo m_i can be implemented most efficiently.

For instance, some processors have instructions for multiplying two μ-bit integers and return the exact $(2\ \mu)$-bit product. If not, then we rather have to assume that the moduli m_i fit into $\mu / 2$ instead of μ bits, or replace μ by $\mu / 2$. Some processors do not provide efficient integer arithmetic at all. In that case, one might rather wish to rely on floating point arithmetic and take $\mu = 52$ (assuming that we have hardware support for double precision). For floating point arithmetic it also matters whether the processor offers a "fused-multiply-add" (FMA) instruction; this essentially provides us with an efficient way to multiply two μ-bit integers exactly using floating point arithmetic.

It is also recommended to choose moduli m_i that fit into slightly less than μ bits whenever possible. Such extra bits can be used to significantly accelerate implementations of modular arithmetic. For a more detailed survey of practically efficient algorithms for modular arithmetic, we refer to [12].

2.3 Naive multi-modular reduction and reconstruction

Let $m_1, \ldots, m_\ell$, $M = m_1 \cdots m_\ell$, $a_1 \in \mathcal{R}_{m_1}, \ldots, a_\ell \in \mathcal{R}_{m_\ell}$ and $x \in \mathcal{R}_M$ be as in the Chinese remainder theorem. We will refer to the computation of $a_1, \ldots, a_\ell$ 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 $m_1, \ldots, m_\ell$ have been fixed once and for all.

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

$$a_i \;:=\; x \operatorname{rem} m_i \qquad (i=1,\ldots,\ell). \tag{1}$$

Inversely, the Chinese remainder theorem provides us with a formula for multi-modular reconstruction:

$$x \;:=\; (c_{m_1;M}\,a_1+\cdots+c_{m_\ell;M}\,a_\ell) \operatorname{rem} M. \tag{2}$$

Since m_1, ..., m_ℓ are fixed, the computation of the cofactors $c_{m_1;M}$ can be regarded as a precomputation.

Assume that our hardware provides an instruction for the exact multiplication of two integers that fit into a machine word. If m_i fits into a machine word, then so does the remainder $a_i = x \operatorname{rem} m_i$. Cutting $c_{m_i;M}$ into ℓ machine words, it follows that the product $c_{m_i;M}\,a_i$ can be computed using ℓ hardware products and ℓ hardware additions. Inversely, the Euclidean division of an ℓ-word integer x by m_i can be done using $2\,\ell+O(1)$ multiplications and $2\,\ell+O(1)$ additions/subtractions: we essentially have to multiply the quotient by m_i and subtract the result from x; each next word of the quotient is obtained through a one word multiplication with an approximate inverse of m_i.

The above analysis shows that the naive algorithm for multi-modular reduction of x modulo $m_1,\ldots,m_\ell$ requires $2\,\ell^2 + O(\ell)$ hardware multiplications and $2\,\ell^2 + O(\ell)$ additions. The multi-modular reconstruction of x rem M can be done using only $\ell^2 + O(\ell)$ multiplications and $\ell^2 + O(\ell)$ additions. Depending on the hardware, the moduli m_i, and the way we implement things, $O(\ell^2)$ more operations may be required for the carry handling—but it is beyond the scope of this paper to go into this level of detail.

3 Gentle moduli

3.1 The naive algorithms revisited for special moduli

Let us now reconsider the naive algorithms from section 2.3, but in the case when the moduli $m_1,\ldots,m_\ell$ are all close to a specific power of two. More precisely, we assume that

$$m_i \;=\; 2^{sw}+\delta_i \qquad (i=1,\ldots,\ell), \tag{3}$$

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 $M = m_1 \cdots m_\ell$. We also assume that w is slightly smaller than μ and that we have a hardware instruction for the exact multiplication of μ-bit integers.

For moduli m_i as in (3), 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 using $\sim \ell\, s$ hardware multiplications. In comparison, the algorithm from section 2.3 requires $\sim 2\ \ell\ s^2$ multiplication when applied to $(s\, w)$-bit (instead of w-bit) integers. More generally, the computation of ℓ remainders $a_1 = x \operatorname{rem} m_1, \ldots, a_\ell = x \operatorname{rem} m_\ell$ can be done using $\sim \ell^2\, s$ instead of $\sim 2\, \ell^2\, s^2$ multiplications. This leads to a potential gain of a factor $2\, s$, although the remainders are $(s\, w)$-bit integers instead of w-bit integers, for the time being.

Multi-modular reconstruction can also be done faster, as follows, using similar techniques as in [1, 5]. Let $x \in \mathcal{R}_M$. Besides the usual binary representation of x and the multi-modular representation $(a_1, \ldots, a_\ell) = (x \operatorname{rem} m_1, \ldots, x \operatorname{rem} m_\ell)$, it is also possible to use the *mixed radix representation* (or *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 $(b_1, \ldots, b_\ell)$ efficiently from $(a_1, \ldots, a_\ell)$. Since $x \operatorname{rem} m_1 = b_1 = a_1$, we must take $b_1 = a_1$. Assume that $b_1, \ldots,$ b_{i-1} have been computed. For $j = i-1, \ldots, 1$ 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

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

Notice that $u_{i-1,i}, \ldots, u_{1,i}$ can be computed using $(i-1)\ (s+1)$ hardware multiplications. 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

$$b_i = v_i\, (a_i - u_{1,i}) \operatorname{rem} m_i,$$

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

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

for $i = \ell, \ldots, 1$, 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 requires $(\ell - i)\, s$ multiplications. Altogether, the computation of $x = x_1$ from $(a_1, \ldots, a_\ell)$ can therefore be done using $\binom{\ell}{2}(2\,s + 1) + \ell\, s^2 \approx \ell\, s\,(\ell + s)$ hardware multiplications.

3.2 Combining special moduli into gentle moduli

For practical applications, we usually wish to work with moduli that fit into one word (instead of s words). With the m_i as in the previous subsection, 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 $m_{i,1}, \ldots, m_{i,s} < 2^{\mu}$. 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 $\eta_1, \ldots, \eta_s$ 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.

We will discuss sieving in more detail in the next section. Assuming that we indeed have found s-gentle moduli $m_1, \ldots, m_\ell$, we may use the naive algorithms from section 2.3 to compute $(x \operatorname{rem} m_{i,1}, \ldots, x \operatorname{rem} m_{i,s})$ from $x \operatorname{rem} m_i$ and *vice versa* for $i = 1, \ldots, \ell$. Given $x \operatorname{rem} m_i$ for all $i = 1, \ldots, \ell$, this allows us to compute all remainders $x \operatorname{rem} m_{i,j}$ using $2\,\ell\, s^2 + O(\ell\, s)$ hardware multiplications, whereas the opposite conversion only requires $\ell\, s^2 + O(\ell\, s)$ multiplications. Altogether, we may thus obtain the remainders $x \operatorname{rem} m_{i,j}$ from $x \operatorname{rem} M$ and *vice versa* using $\sim \ell\, s\, (\ell + 2\, s)$ multiplications.

4 The gentle modulus hunt

4.1 The sieving procedure

We implemented a sieving procedure in MATHEMAGIX [11] that uses the MPARI package with an interface to PARI-GP [15]. 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 $i = 1, \ldots, s$, 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, ..., 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 [9, 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, ..., 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, ..., 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 4.3 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 $\varepsilon<1000000$.

4.2 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 $m_1, \ldots, m_s$ are all prime in Table 2).

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

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

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 perspective, 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 should 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 $2^{sw/2} + \alpha$.

4.3 Application to matrix multiplication

Let us finally return to our favourite application of multi-modular arithmetic to the multiplication of integer matrices $A, B \in \mathbb{Z}^{r \times r}$. From a practical point of view, the second step of the algorithm from the introduction 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).

Bibliography

[1] J.-C. Bajard, M. E. Kaihara, and T. Plantard. Selected rns bases for modular multiplication. In *Proc. of the 19th IEEE Symposium on Computer Arithmetic*, pages 25–32, 2009.

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

[7] J. Doliskani, P. Giorgi, R. Lebreton, and É. Schost. Simultaneous conversions with the Residue Number System using linear algebra. Technical Report `https://hal-lirmm.ccsd.cnrs.fr/lirmm-01415472`, HAL, 2016.

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

[10] J. van der Hoeven. Faster Chinese remaindering. Technical report, HAL, 2016. `http://hal.archives-ouvertes.fr/hal-01403810`.

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

[12] 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.

[13] A. Karatsuba and J. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 7:595–596, 1963.

[14] 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.

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