LMS J. Comput. Math. 17 (Special issue A) (2014) 257–273
doi:10.1112/S1461157014000187

Computing Hasse–Witt matrices of hyperelliptic curves in average polynomial time

David Harvey and Andrew V. Sutherland

Abstract

We present an efficient algorithm to compute the Hasse–Witt matrix of a hyperelliptic curve $C/\mathbb{Q}$ modulo all primes of good reduction up to a given bound N, based on the average polynomial-time algorithm recently proposed by the first author. An implementation for hyperelliptic curves of genus 2 and 3 is more than an order of magnitude faster than alternative methods for $N = 2^{26}$.

1. *Introduction*

Let $C/\mathbb{Q}$ be a smooth projective hyperelliptic curve of genus g defined by an affine equation

$$y^2 = f(x) = \sum_{i=0}^{d} f_i x^i, \quad f_i \in \mathbb{Z},$$

where $d = \deg f$ is either $2g+1$ or $2g+2$ (generically, $d = 2g+2$). If C has good reduction at an odd prime p, the associated *Hasse–Witt matrix* $W_p = [w_{ij}]$ is the $g \times g$ matrix over $\mathbb{Z}/p\mathbb{Z}$ with entries

$$w_{ij} = f_{pi-j}^{(p-1)/2} \bmod p \quad (1 \leqslant i, j \leqslant g),$$

where f_k^n denotes the coefficient of x^k in $f(x)^n$; see [7, 27]. We have the identity

$$\chi(\lambda) \equiv (-1)^g \lambda^g \det(W_p - \lambda I) \bmod p, \tag{1.1}$$

where $\chi(\lambda) \in \mathbb{Z}[\lambda]$ is the characteristic polynomial of the Frobenius endomorphism of the Jacobian of the reduction of C at p; see [17]. In particular, the Weil bounds imply that for $p > 16g^2$ the trace of W_p uniquely determines the trace of Frobenius, hence the number of points $p + 1 - \mathrm{tr}(\mathrm{Frob}_p)$ on the reduction of C at p.

We say that a prime p is *admissible* (*for* C) if p is odd, C has good reduction at p, and p does not divide f_0 or f_d (the constant and leading coefficients of f). The goals of this paper are to give a fast algorithm for computing W_p simultaneously for all admissible primes p up to a given bound N, and to demonstrate the practicality of the algorithm for $g = 2$ and $g = 3$. Applications include numerical investigations of the generalized Sato–Tate conjecture [3, 16] and computing the L-series of C [15].

The algorithm presented here is inspired by [12], which gives an algorithm to compute $\chi(\lambda)$ (not just $\chi(\lambda) \bmod p$) for all primes $p \leqslant N$ of good reduction, in the case that d is odd (which implies that C has a rational Weierstrass point). The running time of that algorithm is $O(g^{8+\epsilon} N \log^{3+\epsilon} N)$; when averaged over primes $p \leqslant N$, this is $O(g^{8+\epsilon} \log^{4+\epsilon} p)$, the first

Received 27 February 2014; revised 23 May 2014.

2010 Mathematics Subject Classification 11G20 (primary), 11Y16, 11M38, 14G10 (secondary).

Contributed to the Algorithmic Number Theory Symposium XI, GyeongJu, Korea, 6–11 August 2014.

The first author was supported by the Australian Research Council, DECRA Grant DE120101293. The second author was supported by NSF grant DMS-1115455.

such result that is polynomial in both g and $\log p$. Critically, the exponent 4 of $\log p$ does not depend on g, and it is already better than that of Schoof's algorithm [20] in genus 1, which has an exponent of 5 when suitably implemented[†]. Pila's generalization of Schoof's algorithm [18] has an exponent of 8 in genus 2 (see [5, 6]), and Eric Schost has suggested (personal communication) that the exponent is 12 in genus 3 (Pila's bound in [18] gives a much larger exponent).

For our implementation we focus on the cases $g \leqslant 3$, where knowledge of $\chi(\lambda) \bmod p$ allows one to efficiently determine $\chi(\lambda)$ using a generic group algorithm, as described in [15]. When $g = 3$, the time required to deduce $\chi(\lambda)$ from $\chi(\lambda) \bmod p$ is $O(p^{1/4+\epsilon})$; while this is exponential in $\log p$, within the feasible range of $p \leqslant N$ (say $N \leqslant 2^{32}$), the time to derive $\chi(\lambda)$ from $\chi(\lambda) \bmod p$ is actually negligible compared to the average time to compute $\chi(\lambda) \bmod p$. We handle all hyperelliptic curves, not just those with a rational Weierstrass point, which in general will not be present. We also introduce optimizations that improve the space complexity by a logarithmic factor, compared to [12], without increasing the running time; indeed, the running time is significantly reduced, as may be seen in Table 3 in § 5.

Asymptotically, we obtain the following theorem bounding the complexity of the algorithm ComputeHasseWittMatrices, which computes W_p for all admissible $p \leqslant N$ (see § 4 for the algorithm and a proof of the theorem). We denote by $\|f\|$ the maximum of the absolute value of the coefficients of f, and by $\mathsf{M}(n)$ the time to multiply two n-bit integers. We may take $\mathsf{M}(n) = O(n \log n \log\log n)$, via [19].

Theorem 1.1. *Assume that $g = O(\log N)$. The running time of the algorithm* ComputeHasseWittMatrices *is*

$$O(g^5 \mathsf{M}(N \log(\|f\| N)) \log N),$$

and it uses

$$O\left(g^2 N \left(1 + \frac{\log \|f\|}{\log N}\right)\right)$$

space.

Assuming $\log \|f\| = O(\log N)$, the bounds in Theorem 1.1 simplify to $O(g^5 N \log^{3+\epsilon} N)$ time and $O(g^2 N)$ space.

In practical terms, the new algorithm is substantially faster than previous methods. We benchmarked our implementation against two of the fastest software packages available for these computations, as analyzed in [15]: the `hypellfrob` [9] and `smalljac` [22] libraries. In genus 2 the new algorithm outperforms both libraries for $N \geqslant 2^{19}$, and is more than 10 times faster for $N = 2^{26}$. In genus 3 the new algorithm is faster across the board, and more than 20 times faster for $N = 2^{26}$. Key to achieving these performance improvements are a faster and more space-efficient algorithm for computing the accumulating remainder trees that play a crucial role in [12], and an optimized fast Fourier transform (FFT) implementation for multiplying integer matrices with very large coefficients.

2. *Overview*

Each row of the Hasse–Witt matrix W_p of C consists of g consecutive coefficients of f^n reduced modulo p, where $n = (p-1)/2$. The total size of all the polynomials f^n needed to compute W_p

[†]This assumes fast integer arithmetic is used, which we do throughout. Under heuristic assumptions, the (probabilistic) SEA algorithm reduces the exponent to 4, but for $g = 1$ generic algorithms that run in $O(p^{1/4+\epsilon})$ time are superior within the feasible range of $p \leqslant N$ in any case.

https://doi.org/10.1112/S1461157014000187 Published online by Cambridge University Press

for $p \leqslant N$ is $O(N^3 \|f\|)$ bits; this makes a naïve approach hopelessly inefficient. Two key optimizations are required to achieve a running time that is quasilinear in N.

First, for a given row of W_p, we only require g coefficients of each f^n. In § 3 we define an r-dimensional row vector v_n, where $r \approx 2g$, consisting of r consecutive coefficients of f^n, including the g coefficients of interest. The coefficients of f^{n+1} corresponding to v_{n+1} are closely related to the coefficients of f^n corresponding to v_n. We use this to derive a linear recurrence $v_{n+1} = v_n T_n$, where T_n is an explicit $r \times r$ transition matrix. The entries of T_n lie in $\mathbb{Q}$, but not necessarily in $\mathbb{Z}$; this requires us to handle the denominators explicitly. These recurrence relations are analogous to the technique of 'reduction towards zero' introduced in [12]; the key point is that the coefficients of the recurrence are *independent of* p. This is in contrast to the recurrence relations used to derive the Hasse–Witt matrix in [1], whose coefficients do depend on p, and which are analogous to the 'horizontal reductions' in [10] and [12].

Second, we only need to know the coefficients of each vector v_n modulo $p = 2n + 1$. The essential difficulty here is that the modulus is different for each n. Following [12], we use an *accumulating remainder tree* to circumvent this problem. More precisely, in § 4 we give an algorithm RemainderTree that takes as input a sequence of integer matrices $A_0, \ldots, A_{b-2}$, a sequence of integer moduli $m_1, \ldots, m_{b-1}$, and an integer row vector V (the 'initial condition'), and computes the reduced partial products (row vectors)

$$C_n := V A_0 \ldots A_{n-1} \bmod m_n,$$

simultaneously for all $0 \leqslant n < b$. The remarkable feature of this algorithm is that its complexity is quasilinear in b.

We may apply RemainderTree to our situation in the following way. During the course of finding an explicit expression for T_n, we will write it as $T_n = M_n/D_n$ where M_n is an integer matrix and D_n is a nonzero integer. It turns out that for any sufficiently large admissible prime $p = 2n + 1$, the p-adic valuation of $D_0 \ldots D_{n-1}$ is at most d. Thus to obtain

$$v_n = v_0 M_0 \ldots M_{n-1} / D_0 \ldots D_{n-1}$$

modulo p, it suffices to compute

$$v_0 M_0 \ldots M_{n-1} \bmod p^{d+1} \quad \text{and} \quad D_0 \ldots D_{n-1} \bmod p^{d+1}.$$

We run RemainderTree twice, first with $V = v_0$ and $A_j = M_j$, and then with $V = 1$ and $A_j = D_j$ (regarding the D_j as 1×1 matrices). In both cases we take the moduli $m_n = p^{d+1}$ if $p = 2n + 1$ is an admissible prime, and let $m_n = 1$ otherwise.

For $g \leqslant 3$, we will show how to tweak this strategy to use the smaller moduli $m_n = p^g$. This has a significant impact on the overall performance and memory consumption. We conjecture that one can always use $m_n = p^g$ (for p sufficiently large compared to g), but we will not attempt to prove this here.

3. *Recurrence relations*

For technical reasons it will be convenient to distinguish between the cases $f_0 \neq 0$ and $f_0 = 0$ (the same distinction arises in [12]). Let

$$r = \begin{cases} d & \text{if } f_0 \neq 0, \\ d - 1 & \text{if } f_0 = 0. \end{cases}$$

For each $1 \leqslant i \leqslant g$, consider the sequence of vectors

$$v_n^{(i)} = [f_{2in+i-r}^n, \ldots, f_{2in+i-1}^n] \in \mathbb{Z}^r \quad (n \geqslant 0).$$

https://doi.org/10.1112/S1461157014000187 Published online by Cambridge University Press

For each admissible prime $p = 2n + 1$, the last g entries of $v_n^{(i)}$ are, modulo p, precisely the entries of the ith row of the Hasse–Witt matrix W_p (in reversed order).

The aim of this section is to develop a recurrence for the $v_n^{(i)}$. For each $n \geqslant 0$, we will construct an $r \times r$ integer matrix $M_n^{(i)}$, and a nonzero integer $D_n^{(i)}$, such that

$$v_{n+1}^{(i)} = v_n^{(i)} M_n^{(i)} / D_n^{(i)}.$$

The entries of $M_n^{(i)}$, and $D_n^{(i)}$, turn out to be polynomials in n and the coefficients of f, which allows us to analyze the p-adic valuation of the partial products of the $D_n^{(i)}$.

The construction proceeds as follows. For any $n \geqslant 0$, the identities

$$f^{n+1} = f f^n \quad \text{and} \quad (f^{n+1})' = (n+1) f' f^n$$

imply the relations

$$f_k^{n+1} = \sum_{j=0}^{d} f_j f_{k-j}^n, \tag{3.1}$$

$$k f_k^{n+1} = (n+1) \sum_{j=1}^{d} j f_j f_{k-j}^n. \tag{3.2}$$

Multiplying (3.1) by k and subtracting (3.2) yields the relation

$$\sum_{j=0}^{d} (nj - k + j) f_j f_{k-j}^n = 0 \tag{3.3}$$

among the coefficients of f^n.

Suppose we are in the case $f_0 \neq 0$, $r = d$. Solving (3.3) for f_k^n yields

$$k f_0 f_k^n = \sum_{j=1}^{d} (nj - k + j) f_j f_{k-j}^n. \tag{3.4}$$

For $k \neq 0$, this expresses f_k^n as a linear combination of d consecutive coefficients of f^n to the 'left' of f_k^n. On the other hand, replacing k by $k + d$ and j by $d - j$ in (3.3) gives

$$(nd - k) f_d f_k^n = -\sum_{j=1}^{d} (n(d-j) - k - j) f_{d-j} f_{k+j}^n. \tag{3.5}$$

For $k \neq nd$, this expresses f_k^n as a linear combination of d consecutive coefficients to the 'right' of f_k^n. Now, suppose we are given as input

$$v_n^{(i)} = [f_{2in+i-d}^n, \ldots, f_{2in+i-1}^n].$$

After $2i$ applications of (3.4), that is, for $k = 2in + i, \ldots, 2in + 3i - 1$ (in that order), and $d - 2i$ applications of (3.5), that is, for $k = 2in + i - d - 1, \ldots, 2in + 3i - 2d$ (in that order), we have extended our knowledge of the coefficients of f^n to the vector

$$[f_{2in+3i-2d}^n, \ldots, f_{2in+3i-1}^n].$$

of length $2d$. From (3.1) we then obtain

$$v_{n+1}^{(i)} = [f_{2in+3i-d}^{n+1}, \ldots, f_{2in+3i-1}^{n+1}].$$

https://doi.org/10.1112/S1461157014000187 Published online by Cambridge University Press

The above procedure defines a $d \times d$ transition matrix $T_n^{(i)}$ mapping $v_n^{(i)}$ to $v_{n+1}^{(i)}$, whose entries are rational functions in $\mathbb{Q}(n, f_0, \ldots, f_d)$. Denominators arise from the divisions by kf_0 and $(nd-k)f_d$ in the various applications of (3.4) and (3.5). Each such divisor is a linear polynomial in $\mathbb{Z}[n]$ multiplied by either f_0 or f_d; thus the denominators of the entries of $T_n^{(i)}$ are polynomials in $\mathbb{Z}[n, f_0, f_d]$. We will take $D_n^{(i)}$ to be the least common denominator of the entries of $T_n^{(i)}$. Since there are d applications of (3.4) and (3.5) altogether, the degree of $D_n^{(i)}$ with respect to n is at most d (it may be smaller due to cancellation).

The case $f_0 = 0$ with $r = d-1$ is similar. We have $f_1 \neq 0$, because f is assumed to be squarefree, and the analogues of (3.4) and (3.5) are

$$(n-k)f_1 f_k^n = -\sum_{j=1}^{d-1}(n(j+1)-k+j)f_{j+1}f_{k-j}^n, \tag{3.6}$$

$$(nd-k)f_d f_k^n = -\sum_{j=1}^{d-1}(n(d-j)-k-j)f_{d-j}f_{k+j}^n, \tag{3.7}$$

which express f_k^n in terms of $d-1$ consecutive coefficients to the left, or right, of f_k^n. Given

$$v_n^{(i)} = [f_{2in+i-d+1}^n, \ldots, f_{2in+i-1}^n],$$

we use these relations to extend $v_n^{(i)}$ to the vector $[f_{2in+3i-2d+1}^n, \ldots, f_{2in+3i-1}^n]$ of length $2d-1$, from which we obtain $v_{n+1}^{(i)}$ from (3.1) as above.

In the subsections that follow we carry out the above procedure explicitly for the specific cases that arise when $g \leqslant 3$.

3.1. *Genus* 1, *quartic model*

Suppose that $C/\mathbb{Q}$ has genus 1. If C has a rational point, then C is an elliptic curve and can be put in Weierstrass form $y^2 = f(x)$ with f cubic, but we first consider the generic case where this need not hold. So let $f(x) = f_4x^4 + f_3x^3 + f_2x^2 + f_1x + f_0$ with $f_0f_4 \neq 0$; then $r = d = 4$. Since the only relevant value of i is 1, we omit the superscripts on $v_n^{(1)}$, $M_n^{(1)}$, $D_n^{(1)}$.

We wish to construct a linear recurrence that expresses the vector

$$v_{n+1} = [f_{2n-1}^{n+1}, f_{2n}^{n+1}, f_{2n+1}^{n+1}, f_{2n+2}^{n+1}] \in \mathbb{Z}^4$$

in terms of the vector

$$v_n = [f_{2n-3}^n, f_{2n-2}^n, f_{2n-1}^n, f_{2n}^n] \in \mathbb{Z}^4;$$

that is, we want a 4×4 integer matrix M_n and a nonzero integer D_n such that

$$v_{n+1} = v_n M_n / D_n.$$

For each odd prime $p = 2n+1$, the Hasse–Witt matrix W_p consists of just the single entry $f_{2n}^n \bmod p$, which is the last entry of $v_n \bmod p$.

We start by extending v_n 'rightwards', using (3.4) with $k = 2n+1$. This yields

$$(2n+1)f_0 f_{2n+1}^n = (2n+3)f_4 f_{2n-3}^n + (n+2)f_3 f_{2n-2}^n + f_2 f_{2n-1}^n - nf_1 f_{2n}^n.$$

Using (3.4) again with $k = 2n+2$, we get

$$(2n+2)f_0 f_{2n+2}^n = (2n+2)f_4 f_{2n-2}^n + (n+1)f_3 f_{2n-1}^n - (n+1)f_1 f_{2n+1}^n.$$

https://doi.org/10.1112/S1461157014000187 Published online by Cambridge University Press

Combining these equations yields

$$\begin{aligned}2(2n+1)f_0^2 f_{2n+2}^n = &-(2n+3)f_1f_4f_{2n-3}^n\\ &+(2(2n+1)f_0f_4-(n+2)f_1f_3)f_{2n-2}^n\\ &+((2n+1)f_0f_3-f_1f_2)f_{2n-1}^n\\ &+nf_1^2f_{2n}^n.\end{aligned}$$

Next we extend v_n 'leftwards' by applying (3.5) with $k = 2n-4$, obtaining

$$(2n+4)f_4f_{2n-4}^n = -(n+3)f_3f_{2n-3}^n - 2f_2f_{2n-2}^n + (n-1)f_1f_{2n-1}^n + 2nf_0f_{2n}^n.$$

With $k = 2n-5$ we get

$$(2n+5)f_4f_{2n-5}^n = -(n+4)f_3f_{2n-4}^n - 3f_2f_{2n-3}^n + (n-2)f_1f_{2n-2}^n + (2n-1)f_0f_{2n-1}^n,$$

and therefore

$$\begin{aligned}(2n+5)(2n+4)f_4^2f_{2n-5}^n = &((n+3)(n+4)f_3^2-3(2n+4)f_2f_4)f_{2n-3}^n\\ &+(2(n+4)f_2f_3+(n-2)(2n+4)f_1f_4)f_{2n-2}^n\\ &+(-(n-1)(n+4)f_1f_3+(2n-1)(2n+4)f_0f_4)f_{2n-1}^n\\ &-2n(n+4)f_0f_3f_{2n}^n.\end{aligned}$$

We have expressions for $f_{2n-5}^n, \ldots, f_{2n+2}^n$ in terms of $f_{2n-3}^n, \ldots, f_{2n}^n$, and we obtain v_{n+1} via

$$\begin{aligned}f_{2n-1}^{n+1} &= f_4f_{2n-5}^n + \ldots + f_0f_{2n-1}^n,\\ &\vdots\\ f_{2n+2}^{n+1} &= f_4f_{2n-2}^n + \ldots + f_0f_{2n+2}^n.\end{aligned}$$

After some algebraic manipulation we obtain the matrix

$$M_n = \begin{bmatrix}(-(n+3)f_3^2+4(n+2)f_2f_4)J_1 & f_3J_2 & 4f_4J_3 & (2n+3)f_1f_4J_4\\ (-2f_2f_3+6(n+2)f_1f_4)J_1 & 2f_2J_2 & 3f_3J_3 & (4(2n+1)f_0f_4+(n+2)f_1f_3)J_4\\ ((n-1)f_1f_3+8(n+2)f_0f_4)J_1 & 3f_1J_2 & 2f_2J_3 & (3(2n+1)f_0f_3+f_1f_2)J_4\\ 2nf_0f_3J_1 & 4f_0J_2 & f_1J_3 & (2(2n+1)f_0f_2-nf_1^2)J_4\end{bmatrix},$$

where

$$\begin{aligned}J_1 &= (n+1)(2n+1)f_0,\\ J_2 &= (n+1)(2n+1)(2n+5)f_0f_4,\\ J_3 &= 2(n+1)(n+2)(2n+5)f_0f_4,\\ J_4 &= (n+2)(2n+5)f_4,\end{aligned}$$

and the denominator

$$D_n = 2(n+2)(2n+1)(2n+5)f_0f_4.$$

Recall that $v_n = v_0M_0 \ldots M_{n-1}/(D_0 \ldots D_{n-1})$. For each admissible prime $p = 2n+1 \geqslant 5$, the p-adic valuation of $D_0 \ldots D_{n-1}$ is exactly 1, since p divides $D_{n-2} = 2n(2n-3)(2n+1)f_0f_4$ exactly once, and p does not divide D_j for $j = n-1$ or any $0 \leqslant j \leqslant n-3$. We may thus compute $v_n \bmod p$ as

$$\left(\frac{v_0M_0 \ldots M_{n-1} \bmod p^2}{D_0 \ldots D_{n-1} \bmod p^2}\right) \bmod p.$$

https://doi.org/10.1112/S1461157014000187 Published online by Cambridge University Press

With additional care it is possible to perform the bulk of the computation working modulo p rather than p^2. As noted above, $D_0 \dots D_{n-3}$ is a p-adic unit, and a direct calculation shows that the entries of the last column of $M_{n-2}M_{n-1}$ are divisible by $2n+1$. Let U_n be the last column of $M_{n-2}M_{n-1}/D_{n-2}D_{n-1}$. Then U_n is p-integral for $p = 2n+1$, and we may compute the last entry of $v_n \bmod p$, that is, the lone entry of the Hasse–Witt matrix W_p, as

$$\left(\left(\frac{v_0 M_0 \dots M_{n-3} \bmod p}{D_0 \dots D_{n-3} \bmod p}\right) U_n\right) \bmod p.$$

REMARK 1. Returning briefly to the general case, we can now see why it always suffices to work with moduli $m_n = p^{d+1}$, for sufficiently large admissible p. The denominator D_n always has the form $D_n = C f_0^{\alpha} f_d^{\beta} \prod_{i=1}^{e} (a_i n + b_i)$, where $C, a_i, b_i \in \mathbb{Z}$ and α, β and e are non-negative integers with $\alpha + \beta \leqslant d$ and $e \leqslant d$. We may assume that $(a_i, b_i) = 1$ for all i. If p is larger than every prime divisor of a_i, we see that $a_i n + b_i$ is divisible by p if and only if $n = -b_i/a_i \bmod p$, and this occurs for at most one value of n in the interval $0 \leqslant n < (p-1)/2$. Moreover for large enough p we see that $a_i n + b_i$ cannot be divisible by p^2 for such n. Thus for all sufficiently large admissible primes $p = 2n+1$, we find that $D_0 \dots D_{n-1}$ has p-adic valuation at most d.

REMARK 2. One can make $f_3 = 0$ by replacing x with $x - f_3/(4f_4)$ and y with $y/(16f_4^2)$ and then clearing denominators. This has the advantage that a factor of f_4 cancels in the above formulae for M_n and D_n, but it will also tend to increase the size of the other coefficients. In general, one can always make $f_{d-1} = 0$ with a similar substitution, and when d is even this allows us to remove a power of f_d from D_n and the entries of M_n.

When $f_0 = 0$ one can follow the procedure above, using (3.6) and (3.7) in place of (3.4) and (3.5); alternatively, one may switch to a cubic model via the substitution $x = 1/u$, $y = v/u^2$, which is discussed in the next section. Both methods lead to essentially the same formulae.

3.2. *Genus* 1, *cubic model*

We now consider the case $g = 1$ with $f(x) = f_3x^3 + f_2x^2 + f_1x + f_0$ and $d = 3$. Assuming $f_0 \neq 0$, we obtain the 3×3 transition matrix

$$M_n = \begin{bmatrix} 2(n+1)(2n+1)f_0f_2 & 6(n+1)(n+3)f_0f_3 & (n+3)(n+2)f_1f_3 \\ 4(n+1)(2n+1)f_0f_1 & 4(n+1)(n+3)f_0f_2 & (n+3)(3(2n+1)f_0f_3 + f_1f_2) \\ 6(n+1)(2n+1)f_0^2 & 2(n+1)(n+3)f_0f_1 & (n+3)(2(2n+1)f_0f_2 - nf_1^2) \end{bmatrix}$$

with denominator

$$D_n = 2(n+3)(2n+1)f_0.$$

For all admissible primes $p = 2n+1 \geqslant 5$, the partial product $D_0 \dots D_{n-1}$ is prime to p.

REMARK 3. In the cubic case one can make $f_3 = 1$ and $f_2 = 0$ with a suitable substitution; this simplifies the formulae but may increase the size of f_0 and f_1. If the cubic $f(x)$ has a rational root, one can make $f_0 = 0$ by translating the root to zero (in which case f_2 will typically be nonzero). This is usually well worth doing, since it reduces the dimension of M_n from 3 to 2 (see below). Similar remarks apply whenever d is odd.

When $f_0 = 0$ we have $y^2 = f_3x^3 + f_2x^2 + f_1x$ and the 2×2 transition matrix

$$M_n = \begin{bmatrix} (n+1)f_2 & 2(n+2)f_3 \\ 2(n+1)f_1 & (n+2)f_2 \end{bmatrix}$$

https://doi.org/10.1112/S1461157014000187 Published online by Cambridge University Press

with denominator

$$D_n = n + 2.$$

For all admissible primes $p = 2n + 1$ the partial product $D_0 \dots D_{n-1}$ is prime to p.

3.3. *Genus* 2

The computations in genus 2 are similar, except now each Hasse–Witt matrix has two rows, which we obtain by computing $v_n^{(i)}$ for $i = 1, 2$. For the sake of brevity, we omit the details and list only the denominators $D_n^{(i)}$; a Sage [21] script for generating the transition matrices $M_n^{(i)}$ is available at [14].

For $i = 1$ we get the denominators

$$D_n^{(1)} = \begin{cases} 8(n+2)(2n+1)(2n+3)(4n+7)(4n+9)f_0 f_6^3 & \text{if } d = 6, f_0 \neq 0, \\ 6(n+2)(2n+1)(3n+5)(3n+7)f_0 f_5^2 & \text{if } d = 5, f_0 \neq 0, \\ 3(n+2)(3n+4)(3n+5)f_5^2 & \text{if } d = 5, f_0 = 0. \end{cases}$$

In the case $d = 6$, one verifies that the last two columns of $M_{n-1}^{(1)}/D_{n-1}^{(1)}$ are p-integral for $p = 2n + 1$, and that $D_0^{(1)} \dots D_{n-2}^{(1)}$ is a p-adic unit except possibly for a single factor of p contributed by $4m + 7$ when $m = (n - 3)/2$ or by $4m + 9$ when $m = (n - 4)/2$ (at most one of these occurs for each p). Thus the desired row of the Hasse–Witt matrix W_p may be computed as the last two entries of

$$\left(\left(\frac{v_0 M_0^{(1)} \dots M_{n-2}^{(1)} \bmod p^2}{D_0^{(1)} \dots D_{n-2}^{(1)} \bmod p^2} \right) \frac{M_{n-1}^{(1)}}{D_{n-1}^{(1)}} \right) \bmod p.$$

Similar observations apply to both of the $d = 5$ cases, and again one finds that it suffices to work with the moduli $m_n = p^2$ (we omit the details).

The denominators for $i = 2$ are

$$D_n^{(2)} = \begin{cases} 8(n+3)(2n+1)(2n+5)(4n+3)(4n+5)f_0^3 f_6 & \text{if } d = 6, f_0 \neq 0, \\ 8(n+4)(2n+1)(4n+3)(4n+5)f_0^3 & \text{if } d = 5, f_0 \neq 0, \\ 3(n+3)(3n+2)(3n+4)f_1^2 & \text{if } d = 5, f_0 = 0. \end{cases}$$

As above, in all three cases one can arrange to use the moduli $m_n = p^2$.

3.4. *Genus* 3

For $i = 1$ we get the denominators

$$D_n^{(1)} = \begin{cases} 72(n+2)(2n+1)(2n+3)(3n+4)(3n+5)(6n+11)(6n+13)f_0 f_8^5 & \text{if } d = 8, f_0 \neq 0, \\ 10(n+2)(2n+1)(5n+7)(5n+8)(5n+9)(5n+11)f_0 f_7^4 & \text{if } d = 7, f_0 \neq 0, \\ 5(n+2)(5n+6)(5n+7)(5n+8)(5n+9)f_7^4 & \text{if } d = 7, f_0 = 0. \end{cases}$$

For $i = 2$ the denominators are

$$D_n^{(2)} = \begin{cases} 8(n+2)(2n+1)(2n+5)(4n+3)(4n+5)(4n+7)(4n+9)f_0^3 f_8^3 & \text{if } d = 8, f_0 \neq 0, \\ 24(n+2)(2n+1)(3n+7)(3n+8)(4n+3)(4n+5)f_0^3 f_7^2 & \text{if } d = 7, f_0 \neq 0, \\ 3(n+2)(3n+2)(3n+4)(3n+5)(3n+7)f_1^2 f_7^2 & \text{if } d = 7, f_0 = 0, \end{cases}$$

https://doi.org/10.1112/S1461157014000187 Published online by Cambridge University Press

and for $i = 3$ they are

$$D_n^{(3)} = \begin{cases} 72(n+3)(2n+1)(2n+7)(3n+2)(3n+4)(6n+5)(6n+7)f_0^5 f_8 & \text{if } d = 8, f_0 \neq 0, \\ 72(n+5)(2n+1)(3n+2)(3n+4)(6n+5)(6n+7)f_0^5 & \text{if } d = 7, f_0 \neq 0, \\ 5(n+4)(5n+3)(5n+4)(5n+6)(5n+7)f_1^4 & \text{if } d = 7, f_0 = 0. \end{cases}$$

In all three cases it is not difficult to show that by pulling out at most the last three factors from $D_0 \dots D_{n-1}$, it suffices to compute the partial products modulo $m_n = p^3$, where $p = 2n + 1$.

4. *Accumulating remainder trees*

Given a sequence of $r \times r$ integer matrices $A_0, \dots, A_{b-2}$, an r-dimensional integer row vector V, and a sequence of positive integer moduli $m_1, \dots, m_{b-1}$, we wish to compute the sequence of reduced row vectors $C_1, \dots, C_{b-1}$, where

$$C_n := VA_0 \dots A_{n-1} \bmod m_n.$$

For convenience, we define $m_0 = 1$, so C_0 is the zero vector, and we let A_{b-1} be the identity matrix. We also make the simplifying assumption that the bound $b = 2^\ell$ is a power of two, although this is not necessary. In terms of the prime bound N of the previous sections, we use $b = N/2$, which can be viewed as a bound on $n = (p-1)/2$.

As in [**12**, § 3], we work with complete binary trees of depth ℓ with nodes indexed by pairs (i, j) with $0 \leqslant i \leqslant \ell$ and $0 \leqslant j < 2^i$. For each node we define

$$\begin{aligned} m_{i,j} &:= m_{j2^{\ell-i}} m_{j2^{\ell-i}+1} \dots m_{(j+1)2^{\ell-i}-1}, \\ A_{i,j} &:= A_{j2^{\ell-i}} A_{j2^{\ell-i}+1} \dots A_{(j+1)2^{\ell-i}-1}, \\ C_{i,j} &:= VA_{i,0} \dots A_{i,j-1} \bmod m_{i,j}. \end{aligned} \tag{4.1}$$

The values $m_{i,j}$ and $A_{i,j}$ may be viewed as nodes in a *product tree*, in which each node is the product of its children, with leaves $m_j = m_{\ell,j}$ and $A_j = A_{\ell,j}$, for $0 \leqslant j < b$. Each vector $C_{i,j}$ is the product of V and all the matrices $A_{i,k}$ that are nodes on the same level and to the left of $A_{i,j}$, reduced modulo $m_{i,j}$. To compute the vectors $C_j = C_{\ell,j}$, we use the following algorithm.

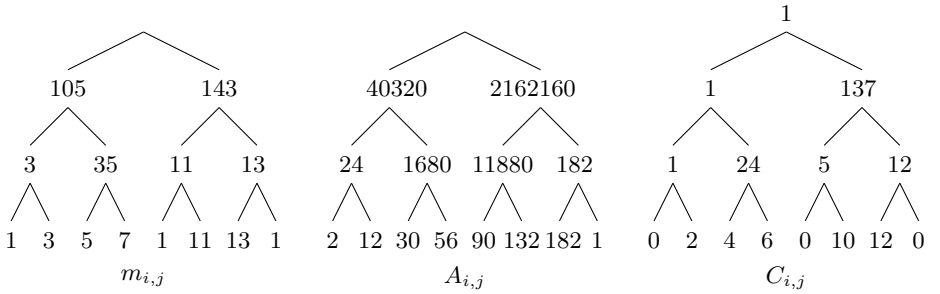
Algorithm RemainderTree

Given $V, A_0, \dots, A_{b-1}$ and $m_0, \dots, m_{b-1}$, with $b = 2^\ell$, compute $m_{i,j}$, $A_{i,j}$, and $C_{i,j}$ as follows.

1. Set $m_{\ell,j} = m_j$ and $A_{\ell,j} = A_j$, for $0 \leqslant j < b$.
2. For i from $\ell - 1$ down to 1:
   For $0 \leqslant j < 2^i$, set $m_{i,j} = m_{i+1,2j} m_{i+1,2j+1}$ and $A_{i,j} = A_{i+1,2j} A_{i+1,2j+1}$.
3. Set $C_{0,0} = V \bmod m_{0,0}$ and then for i from 1 to ℓ:
   For $0 \leqslant j < 2^i$ set $C_{i,j} = \begin{cases} C_{i-1,\lfloor j/2 \rfloor} \bmod m_{i,j} & \text{if } j \text{ is even}, \\ C_{i-1,\lfloor j/2 \rfloor} A_{i,j-1} \bmod m_{i,j} & \text{if } j \text{ is odd}. \end{cases}$

To illustrate the algorithm, let us compute $(p-1)! \bmod p$ for the odd primes $p < 15$; this does not correspond to the computation of a Hasse–Witt matrix, but this makes no difference as far as the RemainderTree algorithm is concerned. We use odd moduli $m_n = 2n + 1$ for $0 \leqslant n < 8$, except that we set the composite moduli m_4 and m_7 to 1, and we use 1×1 matrices $A_n = [(2n+1)(2n+2)]$ for $0 \leqslant n < 7$, and let $A_7 = [1]$ and $V = [1]$. The trees $m_{i,j}$, $A_{i,j}$, and $C_{i,j}$ computed by the RemainderTree algorithm are depicted below.

https://doi.org/10.1112/S1461157014000187 Published online by Cambridge University Press



THEOREM 4.1. *Let B be an upper bound on the bit-size of $\prod_{j=0}^{b-1} m_j$, let B' be an upper bound on the bit-size of any entry of V, let h be an upper bound on the bit-size of any $m_0, \dots, m_{b-1}$ and any entry in $A_0, \dots, A_{b-1}$, and assume that $\log r = O(h)$. The running time of the* RemainderTree *algorithm is*

$$O(r^3 \mathsf{M}(B + bh) \log b + r\mathsf{M}(B')),$$

and its space complexity is

$$O(r^2(B + bh) \log b + rB').$$

Proof. There are $O(B)$ bits at each level of the $m_{i,j}$ tree. For the $A_{i,j}$ tree, observe that the entries of any product $A_{j_1} \dots A_{j_2-1}$ have bit-size $O((j_2-j_1)h+\log r)$; thus there are $O(bh)$ bits at each level of the $A_{i,j}$ tree. These estimates account for the main terms in the time and space bounds; for more details see the proofs of [2, Theorem 1.1] or [12, Proposition 4]. We assume classical matrix multiplication throughout, with complexity $O(r^3)$. The terms involving B' cover any additional cost due to the initial reduction of V modulo $m_{0,0}$. □

4.1. *A fast space-efficient remainder tree algorithm*

The algorithm given in the previous section uses more space than is necessary. We now describe a more space-efficient approach that is also faster by a significant constant factor. As above, we assume $b = 2^\ell$ is a power of two. Our strategy is to pick a parameter k, and rather than computing a single remainder tree, separately compute the 2^k subtrees corresponding to the bottom $\ell - k$ layers of the original tree, each of which has height $\ell - k$ and $t = 2^{\ell-k}$ leaves.

For $0 \leqslant s < 2^k$, we define the sth tree as follows. Let

$$\begin{aligned} m_j^s &:= m_{st+j} \quad (0 \leqslant j < t), \\ A_j^s &:= A_{st+j} \quad (0 \leqslant j < t), \\ V^s &:= VA_0 \dots A_{st-1} \bmod m_{st} \dots m_{b-1}. \end{aligned}$$

For $0 \leqslant i \leqslant \ell - k$ and $0 \leqslant j < 2^i$ we define $m_{i,j}^s$, $A_{i,j}^s$ and $C_{i,j}^s$ in terms of the above data, in direct analogy with (4.1).

We then have $m_{i,j}^s = m_{i+k,j+2^i s}$ and $A_{i,j}^s = A_{i+k,j+2^i s}$; in other words, the $m_{i,j}^s$ and $A_{i,j}^s$ trees are identical to the corresponding subtrees of the original $m_{i,j}$ and $A_{i,j}$ trees rooted at the node (k, s). The same is true for the $C_{i,j}^s$ tree, namely, we have $C_{i,j}^s = C_{i+k,j+2^i s}$. To see this, observe that

$$V^s = VA_{k,0} \dots A_{k,s-1} \bmod m_{k,s} \dots m_{k,2^k-1},$$

and $m_{0,0}^s = m_{k,s}$. Therefore

$$C_{0,0}^s = V^s \bmod m_{0,0}^s = VA_{k,0} \dots A_{k,s-1} \bmod m_{k,s} = C_{k,s}.$$

https://doi.org/10.1112/S1461157014000187 Published online by Cambridge University Press

For the remaining nodes, the claim $C^s_{i,j} = C_{i+k,j+2^i s}$ follows by working downwards from the root of the C^s tree.

The idea of the RemainderForest algorithm below is to compute each subtree separately, allowing us to reuse space, and to keep track of the vector V^s and the moduli product

$$Y^s := m_{st} \dots m_{b-1}$$

as we proceed from one subtree to the next. The RemainderTree algorithm may be viewed as a special case of the RemainderForest algorithm, using $k = 0$.

Algorithm RemainderForest

Given $V, A_0, \dots, A_{b-1}$ and $m_0, \dots, m_{b-1}$, with $b = 2^\ell$, and an integer $k \in [0, \ell]$, compute $C_0, \dots, C_{b-1}$ as follows.

1. Set $Y^0 \leftarrow m_0 \dots m_{b-1}$ and $V^0 \leftarrow V \bmod Y^0$, and let $t = 2^{\ell-k}$.
2. For s from 0 to $2^k - 1$:
   a. Call RemainderTree with inputs V^s, $A_{st}, \dots, A_{(s+1)t-1}$, and $m_{st}, \dots, m_{(s+1)t-1}$ to compute trees m^s, A^s, C^s.
   b. Set $Y^{s+1} \leftarrow Y^s / m^s_{0,0}$ and $V^{s+1} \leftarrow V^s A^s_{0,0} \bmod Y^{s+1}$.
   c. Output the values $C_{st+j} = C^s_{e,j}$ for $0 \leqslant j < t$.
   d. Discard Y^s, V^s, and the trees m^s, A^s, C^s.

We now bound the complexity of the RemainderForest algorithm. We do not include the size of the input in our space bound; in the context of computing Hasse–Witt matrices the input matrices A_j are dynamically computed as they are needed, in blocks of size $2^{\ell-k}$.

Theorem 4.2. *Let B be an upper bound on the bit-size of $\prod_{j=0}^{b-1} m_j$ such that $B/2^k$ is an upper bound on the bit-size of $\prod_{j=st}^{st+t-1} m_j$ for all s. Let B' be an upper bound on the bit-size of any entry of V, let h be an upper bound on the bit-size of any $m_0, \dots, m_{b-1}$ and any entry in $A_0, \dots, A_{b-1}$, and assume that $\log r = O(h)$. The running time of the* RemainderForest *algorithm is*

$$O(r^3 \mathsf{M}(B + bh)(\ell - k) + 2^k r^2 \mathsf{M}(B) + r\mathsf{M}(B')),$$

and its space complexity is

$$O(2^{-k} r^2 (B + bh)(\ell - k) + r(B + B')).$$

Proof. The time complexity of step 1 is $O(\mathsf{M}(B) \log b + r\mathsf{M}(B + B'))$. There are 2^k calls to RemainderTree in step 2, each of which takes time

$$O(r^3 \mathsf{M}(2^{-k}B + 2^{-k}bh)(\ell - k) + r\mathsf{M}(B)),$$

by Theorem 4.1, since the bit-size of any entry of any V^s is bounded by $O(B)$. The cost of step 2b is bounded by $O(\mathsf{M}(B) + r^2 \mathsf{M}(B + 2^{-k}bh))$, thus each invocation of step 2 costs

$$O(r^3 \mathsf{M}(2^{-k}B + 2^{-k}bh)(\ell - k) + r^2 \mathsf{M}(B)).$$

Multiplying by 2^k yields the desired time bound. The first term in the space bound matches the corresponding term in Theorem 4.1; the second term bounds the space needed for step 1 (and the output), and dominates the second term in the space bound of Theorem 4.1. □

With $k = 0$ we have $\ell - k = \ell = \log_2 b$, and the bounds in Theorem 4.2 reduce to those of Theorem 4.1. With $k = \ell$ the RemainderForest algorithm has essentially optimal

https://doi.org/10.1112/S1461157014000187 Published online by Cambridge University Press

space complexity $O(rbh)$ (matching the size of its output), but its time complexity is then quasiquadratic in b, rather than quasilinear. The intermediate choice $k = \log_2 \ell + O(1)$ yields a time complexity that is at least as good as that of the RemainderTree algorithm (and may be smaller by a significant constant factor), but with the space complexity improved by a factor of $\log b$. We will see below that for computing Hasse–Witt matrices, bh is somewhat larger than B, and this implies that an even better choice is $k = 2\log_2 \ell + O(1)$, reducing the space complexity by a further factor of $\log b$. See Table 3 in § 5 for an explicit example.

Remark 4. The space complexity can be further reduced using a time-space trade-off as described in [2, Theorem 1.2]. In practice we find that when computing Hasse–Witt matrices using the RemainderForest approach, for $g \leqslant 3$ and the range of N of interest to us, space is not a limiting factor and no time-space trade-off is necessary. See § 5 for further details.

4.2. *Computing the Hasse–Witt matrix*

We now give a complete algorithm for computing the Hasse–Witt matrix W_p of a hyperelliptic curve at all admissible primes $p \leqslant N$; as noted above, the bound N on p corresponds to a bound of $b = N/2$ on n. While the basic approach has been explained in the previous sections, to achieve the best space complexity we must interleave the RemainderForest computations involving the matrices M_n and denominators D_n, so we use RemainderTree to directly handle each subtree, rather than using RemainderForest as a black box. This also allows us to more carefully control the size of the moduli that we use, as discussed further below.

Algorithm ComputeHasseWittMatrices

Given a hyperelliptic curve $C\colon y^2 = f(x) = \sum_{i=0}^{d} f_i x^i$ of genus g, compute the Hasse–Witt matrices W_p for admissible primes $p \leqslant N$ as follows.

1. Construct a list $\mathcal{P}$ of the admissible primes $p = 2n + 1 \leqslant N$.
2. For i from 1 to g:
   a. Compute $M^{(i)} \in \mathbb{Z}[n]^{r\times r}$ and $D^{(i)} \in \mathbb{Z}[n]$ satisfying $v_{n+1}^{(i)} = v_n^{(i)} M^{(i)}(n)/D^{(i)}(n)$, as in § 3.
   b. Use ComputeHasseWittRows below to compute the ith row of W_p for all $p \in \mathcal{P}$.
3. Output the matrices W_p.

As discussed in § 3, in order to minimize the power of $p = 2n+1$ that we use as our moduli, let e and w be integers such that p^e does not divide $D_0 \dots D_{n-1-w}$ for all sufficiently large admissible p. For $g \leqslant 3$ using $e = g$ and $w \leqslant 3$ suffices; in general e and w are both $O(g)$. Our strategy is to compute the partial products $M_0 \dots M_{n-1-w}$ and $D_0 \dots D_{n-1-w}$ modulo p^e using remainder trees, and to handle the last w values of M_j and D_j separately; this allows us to use a smaller value of e than would otherwise be possible. In the context of the RemainderTree algorithm, this means shifting the moduli m_j by w places to the left, relative to the A_j.

Algorithm ComputeHasseWittRows

Given $i \in [1, g]$, positive integers e, w, a list $\mathcal{P}$ of admissible primes $p \leqslant N = 2^{\ell+1}$, a matrix $M^{(i)} \in \mathbb{Z}[n]^{r\times r}$, and $D^{(i)} \in \mathbb{Z}[n]$, compute the ith row of W_p for all $p \in \mathcal{P}$ as follows.

1. Compute $Y = \prod_{p\in\mathcal{P}} p^g$, let $v = 1$, and let $V \in \mathbb{Z}^r$ be the $(r - i + 1)$th standard basis vector.
2. Fix $k = 2\log_2(\ell\sqrt{g}) + O(1)$, let $t = 2^{\ell-k}$, and for s from 0 to $2^k - 1$:
   a. For $st \leqslant j < (s+1)t$, set $m_j = p^e = (2j + 1 + 2w)^e$ if $p \in \mathcal{P}$ and 1 otherwise.
   b. Compute $M_j = M(j)$ and $D_j = D(j)$ for $st \leqslant j < (s+1)t + w - 1$.
   c. Call RemainderTree with inputs V, M_j, m_j to compute $C_j = V \prod_{u=0}^{j-1} M_u \bmod m_j$, $m^s = \prod m_j$, and $M^s = \prod M_j$, where j ranges over integers from st to $st+t-1$.

https://doi.org/10.1112/S1461157014000187 Published online by Cambridge University Press

d. Call RemainderTree with inputs v, D_j, m_j to compute $c_j = v \prod_{u=0}^{j-1} D_u \bmod m_j$ and $D^s = \prod D_j$, where j ranges over integers from st to $st + t - 1$.
e. Set $Y \leftarrow Y/m^s$, $V \leftarrow VM^s \bmod Y$, and $v \leftarrow vD^s \bmod Y$.
f. Compute $v_j = C_j M_j \dots M_{j+w-1}/(c_j D_j \dots D_{j+w-1}) \bmod p$ for $st \leqslant j < (s+1)t$ such that $p = 2j + 1 + 2w \in \mathcal{P}$, and extract the ith row of W_p as the last g entries of v_j.

3. Output the ith row of each of the matrices W_p for $p \in \mathcal{P}$.

We now prove the main result announced in the introduction, which bounds the time and space complexity of ComputeHasseWittMatrices by $O(g^5 \mathsf{M}(N \log(\|f\|N)) \log N)$ and $O(g^2 N(1 + \log \|f\|/\log N))$, respectively, assuming $g = O(\log N)$.

Proof of Theorem 1.1. The time and space needed to enumerate the primes in $[1, N]$ may be bounded by $O(N \log^{2+\epsilon} N)$ and $O(N)$, respectively, via [2, Proposition 2.3], by dividing the interval $[1, N]$ into $O(\log^3 N)$ subintervals. It follows from Chebyshev's bound that $\mathcal{P}$ uses $O(N)$ space. The complexity of ComputeHasseWittRows may be bounded as in the proof of Theorem 4.2; the only new elements are steps 2a and 2b, which have a total time complexity of $O(g^4 N \mathsf{M}(\log(\|f\|N)))$, and step 2f, whose complexity is lower. This is within our desired time bound, and the space complexity of these steps is dominated by the size of the output.

We now proceed as in the proof of Theorem 4.2. We have $B = O(gN)$, since $\sum_{p \leqslant N} \log p \sim N$, and we note that the requirement that $B/2^k$ bound the bit-size of the product $m_{st} \dots m_{st+t-1}$ is satisfied for any $k = O(\log \log N)$; these facts follow from the prime number theorem. Further, $b = N/2$, $B' = O(1)$, $\ell = \log_2 N - 1$, $k = 2 \log_2(\ell\sqrt{g}) + O(1)$, and $h = O(g \log(\|f\|N))$, since the polynomials in $M(n)$ and $D(n)$ all have degree $O(g)$ and coefficients of bit-size $O(g \log \|f\|)$, and the moduli have bit-size $O(g \log N)$. This yields the time bound

$$O(g^3 \mathsf{M}(gN + hN) \log N + g^3 \mathsf{M}(gN) \log^2 N + g),$$

and the space bound

$$O(g^2(gN + hN) \log N/(g \log^2 N) + g^2 N).$$

This yields $O(g^4 \mathsf{M}(N \log(\|f\|N)) \log N)$ time and $O(g^2 N(1 + \log \|f\|/\log N))$ space bounds for ComputeHasseWittRows, which is called g times. □

5. *Implementation details and performance results*

We implemented the ComputeHasseWittMatrices algorithm in C, using the `gcc` compiler [4] and the GNU multiple-precision arithmetic library (GMP) [8]. For the crucial operation of multiplying matrices with very large integer entries, we used a customized FFT implementation as described below.

5.1. *Customized FFT*

The customized FFT uses the standard 'small primes' approach, as outlined in [26, Chapter 8]. To compute a product uv, where $u, v \in \mathbb{Z}$, we choose a parameter $c \geqslant 1$ and write $u = F(2^c)$ and $v = G(2^c)$, where $F, G \in \mathbb{Z}[x]$ have coefficients bounded by 2^c. We then compute the polynomial product $FG \in \mathbb{Z}[x]$ and obtain uv as $(FG)(2^c)$. To compute FG, we choose four suitable 62-bit primes $p_1, \dots, p_4$ and compute $FG \bmod p_i$ in $(\mathbb{Z}/p_i\mathbb{Z})[x]$ for each i, and then reconstruct FG via the Chinese remainder theorem. The parameter c is chosen as large as possible so that the coefficients of FG remain bounded by $p_1 \dots p_4$. Multiplication in $(\mathbb{Z}/p_i\mathbb{Z})[x]$ is achieved by using Fourier transforms (number-theoretic transforms) over $\mathbb{Z}/p_i\mathbb{Z}$. This requires $p_i = 1 \bmod 2^a$, where 2^a is the transform length. Our implementation uses optimized modular arithmetic as

https://doi.org/10.1112/S1461157014000187 Published online by Cambridge University Press

in [13], truncated Fourier transforms to avoid power-of-two jumps in running times [24, 25], and ideas from [11] to improve locality.

To multiply matrices we use the same strategy. If u and v are $r \times r$ integer matrices (recall that $r = d$ or $d-1$, where d is the degree of the polynomial f in the curve equation $y^2 = f(x)$), we write $u = F(2^c)$ and $v = G(2^c)$ where now F and G are matrices of polynomials with small coefficients, or equivalently polynomials with matrix coefficients. We then perform $2r^2$ forward transforms, multiply the resulting Fourier coefficients (each coefficient is an $r \times r$ matrix over $\mathbb{Z}/p_i\mathbb{Z}$), and perform r^2 inverse transforms, with a final linear-time substitution generating the desired product uv. Our implementation allows the polynomial entries to have signed coefficients, so that we can directly handle matrices u and v containing a mixture of positive and negative entries. Matrix-vector products are handled similarly.

The main advantage of this approach over a straightforward GMP implementation is that we require only $O(r^2)$ transforms rather than $O(r^3)$. In our computations the Fourier transforms make up the bulk of the time spent on matrix multiplication.

5.2. *Timings*

The timings listed in this section were obtained using an 8-core Intel Xeon E5-2670 CPU running at 2.60 GHz, with 20 MB of cache and 32 GB of RAM; in each case we list the total CPU time, in seconds, for a single-threaded implementation. Table 1 lists timings for increasing values of N with $g = 1, 2, 3$ and each of the three possible values of r; as in § 3 we have

$$r = \begin{cases} 2g & \text{when } d = 2g+1 \text{ and } f_0 = 0, \\ 2g+1 & \text{when } d = 2g+1 \text{ and } f_0 \neq 0, \\ 2g+2 & \text{when } d = 2g+2 \text{ and } f_0 \neq 0. \end{cases}$$

Table 2 gives the corresponding memory consumption for each case.

The impact of varying the parameter k, which determines the number 2^k of subtrees used in the RemainderForest algorithm, is illustrated for a particular example with $g = 3$ and $N = 20$ in Table 3. In all of our other tests the parameter k was chosen to optimize time; the

Table 1. *Time (CPU seconds) for Hasse–Witt matrix computations for the curve* $y^2 = 2x^d + 3x^{d-1} + \ldots + p_{d+1}$, *where* p_n *is the* n*th prime* ($f_0 = 0$ *for* $r = 2g$).

	$g=1$			$g=2$			$g=3$		
N	$r=2$	$r=3$	$r=4$	$r=4$	$r=5$	$r=6$	$r=6$	$r=7$	$r=8$
2^{14}	< 1	< 1	< 1	< 1	< 1	1	1	2	3
2^{15}	< 1	< 1	< 1	1	1	2	3	6	9
2^{16}	< 1	< 1	1	2	3	5	8	14	21
2^{17}	< 1	1	1	4	7	12	20	34	52
2^{18}	1	2	4	9	17	29	49	81	123
2^{19}	1	4	8	22	40	69	116	192	294
2^{20}	3	9	20	50	94	166	282	459	694
2^{21}	7	21	47	123	227	398	667	1085	1633
2^{22}	17	49	114	287	534	946	1560	2540	3810
2^{23}	38	115	268	645	1240	2230	3660	5940	9100
2^{24}	89	271	641	1510	2920	5260	8490	13 800	20 600
2^{25}	202	628	1470	3430	6740	11 800	19 600	31 800	47 200
2^{26}	470	1475	3390	7930	15 800	27 400	44 700	72 900	107 000

https://doi.org/10.1112/S1461157014000187 Published online by Cambridge University Press

optimal choice of k varies with both N and r and in our tests ranged from 4 to 8. As can be seen in Table 3, the value of k that optimizes time also yields a space utilization that is much better than would be achieved by the original RemainderTree algorithm (the case $k = 0$). Even in our largest tests, the time-optimal value of k yielded a space utilization under 20 GB, well within the 32 GB available on our test system. By contrast, the original RemainderTree algorithm would have required more than 1 TB of memory in our larger tests.

Tables 4 compares the performance of the new algorithm (in the column labelled `hassewitt`) to the `smalljac` implementation described in [15]. In genus 2 the `smalljac` implementation relies primarily on group computations in the Jacobian of the curve, as described in [15], and the current version [22] includes additional improvements from [23]. As can be seen in the table, the new algorithm surpasses the performance of `smalljac` when N is between 2^{18} and 2^{19} and is more than 12 times faster for $N = 2^{26}$.

As noted in [15], for genus 3 curves, using an optimized version of Kedlaya's algorithm [10] is faster than using group computations in the Jacobian for $N \geqslant 2^{16}$. Table 5 compares the performance of the new algorithm to that of the `hypellfrob` library [9], which implements the algorithm of [10], using one digit of p-adic precision (sufficient to compute the Hasse–Witt matrix). In genus 3 the new algorithm is substantially faster than `hypellfrob` for all the values of N that we tested, and more than 20 times faster for $N = 2^{26}$. We do not include a column for the case $r = 8$ in Table 5 because the `hypellfrob` library requires d to be odd.

Table 2. *Space (MB) for Hasse–Witt matrix computations for the curve* $y^2 = 2x^d + 3x^{d-1} + \ldots + p_{d+1}$, *where* p_n *is the* n*th prime* ($f_0 = 0$ *for* $r = 2g$).

	$g = 1$			$g = 2$			$g = 3$		
N	$r = 2$	$r = 3$	$r = 4$	$r = 4$	$r = 5$	$r = 6$	$r = 6$	$r = 7$	$r = 8$
2^{14}	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1	1
2^{15}	< 1	< 1	1	1	1	1	4	6	8
2^{16}	1	1	1	3	5	7	9	12	16
2^{17}	2	2	4	6	10	14	18	25	33
2^{18}	5	5	8	13	20	29	38	51	69
2^{19}	11	11	17	27	41	59	79	106	144
2^{20}	16	21	35	53	83	121	162	220	295
2^{21}	32	42	71	108	169	249	332	450	610
2^{22}	63	84	145	218	346	517	682	942	1258
2^{23}	124	170	307	444	716	1064	1396	1940	2614
2^{24}	247	634	634	920	1467	2195	2869	3980	5385
2^{25}	498	708	1300	1890	3014	3398	5865	8231	11 162
2^{26}	1002	1440	2679	3843	6478	6950	12 134	12 925	17 137

Table 3. *Time (CPU seconds) and space (MB) for Hasse–Witt matrix computations for the curve* $y^2 = 2x^7 + 3x^6 + 5x^5 + 7x^4 + 11x^3 + 13x^2 + 17x + 19$ *with* $N = 20$ *and varying* k.

	k											
	0	1	2	3	4	5	6	7	8	9	10	11
Time (s)	750	718	661	602	535	483	**459**	466	540	736	1145	2055
Space (MB)	8529	4416	2215	1089	533	311	**220**	178	162	153	149	147

https://doi.org/10.1112/S1461157014000187 Published online by Cambridge University Press

TABLE 4. *Performance comparison with* `smalljac` *in genus* 2. *Times in CPU seconds.*

	$r = 4$		$r = 5$		$r = 6$	
N	`hassewitt`	`smalljac`	`hassewitt`	`smalljac`	`hassewitt`	`smalljac`
2^{14}	0.2	0.2	0.4	0.2	0.7	0.3
2^{15}	0.6	0.5	1.1	0.6	1.9	0.7
2^{16}	1.4	1.7	2.8	1.7	4.9	2.0
2^{17}	3.5	5.6	6.8	5.6	11.9	6.4
2^{18}	8.6	19.9	16.8	20.2	29.0	22.1
2^{19}	20.6	76.0	39.7	76.4	69.1	83.4
2^{20}	48.9	257	94.4	257	166	284
2^{21}	123	828	227	828	398	914
2^{22}	287	2630	534	2630	946	2900
2^{23}	645	8560	1240	8570	2230	9520
2^{24}	1510	28000	2920	28 000	5260	31 100
2^{25}	3430	92200	6740	92 300	11 800	102 000
2^{26}	7930	314 000	15 800	316 000	27 400	349 000

TABLE 5. *Performance comparison with* `hypellfrob` *in genus* 3. *Times in CPU seconds.*

	$r = 6$		$r = 7$	
N	`hassewitt`	`hypellfrob`	`hassewitt`	`hypellfrob`
2^{14}	1.3	6.7	2.0	6.8
2^{15}	3.4	15.5	5.5	15.6
2^{16}	8.3	37.4	13.6	37.6
2^{17}	20.2	95.1	33.3	95.0
2^{18}	48.6	249	80.4	250
2^{19}	116	680	192	681
2^{20}	282	1910	459	1920
2^{21}	667	5450	1090	5460
2^{22}	1560	16 200	2540	16 300
2^{23}	3660	49 400	5940	49 400
2^{24}	8490	152 000	13 800	152 000
2^{25}	19600	467 000	31 800	467 000
2^{26}	44 700	1490 000	72 900	1490 000

References

1. A. BOSTAN, P. GAUDRY and É. SCHOST, 'Linear recurrences with polynomial coefficients and application to integer factorization and Cartier–Manin operator', *SIAM J. Comput.* 36 (2007) no. 6, 1777–1806; MR 2299425 (2008a:11156).
2. E. COSTA, R. GERBICZ and D. HARVEY, 'A search for Wilson primes', *Math. Comp.* (2014) to appear.
3. F. FITÉ, K. S. KEDLAYA, V. ROTGER and A. V. SUTHERLAND, 'Sato–Tate distributions and Galois endomorphism modules in genus 2', *Compos. Math.* 148 (2012) no. 5, 1390–1442; MR 2982436.
4. Free Software Foundation, *GNU compiler collection*, version 4.8, 2013, http://gcc.gnu.org/.
5. P. GAUDRY, D. KOHEL and B. SMITH, 'Counting points on genus 2 curves with real multiplication', *Advances in cryptology—ASIACRYPT 2011*, Lecture Notes in Computer Science 7073 (Springer, Heidelberg, 2011) 504–519; MR 2935020.
6. P. GAUDRY and É. SCHOST, 'Genus 2 point counting over prime fields', *J. Symbolic Comput.* 47 (2012) no. 4, 368–400; MR 2890878.
7. JOSEP GONZÁLEZ, 'Hasse–Witt matrices for the Fermat curves of prime degree', *Tohoku Math. J.* (2) 49 (1997) no. 2, 149–163; MR 1447179 (98b:11064).

https://doi.org/10.1112/S1461157014000187 Published online by Cambridge University Press

8. T. Granlund and the GMP development team, GNU multiple precision arithmetic library, version 5.1, 2013, http://gmplib.org/.

9. D. Harvey, `hypellfrob` software library, version 2.1.1, 2008, http://web.maths.unsw.edu.au/∼davidharvey/code/hypellfrob/hypellfrob-2.1.1.tar.gz.

10. D. Harvey, 'Kedlaya's algorithm in larger characteristic', *Int. Math. Res. Not. IMRN* (2007) no. 22, doi:10.1093/imrn/rnm095.

11. D. Harvey, 'A cache-friendly truncated FFT', *Theoret. Comput. Sci.* 410 (2009) no. 27–29, 2649–2658; MR 2531107 (2010g:68327).

12. D. Harvey, 'Counting points on hyperelliptic curves in average polynomial time', *Ann. of Math.* (2) 179 (2014) no. 2, 783–803.

13. D. Harvey, 'Faster arithmetic for number-theoretic transforms', *J. Symbolic Comput.* 60 (2014) 113–119; MR 3131382.

14. D. Harvey and A. V. Sutherland, Sage worksheet for computing transition matrices, 2014, http://math.mit.edu/∼drew/Hasse-Witt-transition-matrices.sws.

15. K. S. Kedlaya and A. V. Sutherland, 'Computing L-series of hyperelliptic curves', *Algorithmic Number Theory Eighth International Symposium (ANTS VIII)*, Lecture Notes in Computer Science 5011 (Springer, Berlin, 2008) 312–326; MR 2467855 (2010d:11070).

16. K. S. Kedlaya and A. V. Sutherland, 'Hyperelliptic curves, L-polynomials, and random matrices', *Arithmetic, geometry, cryptography and coding theory*, Contemporary Mathematics 487 (American Mathematical Society, Providence, RI, 2009) 119–162; MR 2555991 (2011d:11154).

17. J. I. Manin, 'The Hasse–Witt matrix of an algebraic curve', *AMS Trans. Series 2* 45 (1965) 245–264; (originally published in Izv. Akad. Nauk SSSR Ser. Mat. **25** (1961) 153–172); MR 0124324 (23 #A1638).

18. J. Pila, 'Frobenius maps of abelian varieties and finding roots of unity in finite fields', *Math. Comp.* 55 (1990) no. 192, 745–763; MR 1035941 (91a:11071).

19. A. Schönhage and V. Strassen, 'Schnelle multiplikation grosser zahlen', *Computing (Arch. Elektron. Rechnen)* 7 (1971) 281–292; MR 0292344 (45 #1431).

20. R. Schoof, 'Elliptic curves over finite fields and the computation of square roots mod p', *Math. Comp.* 44 no. 170, 483–494; MR 777280 (86e:11122).

21. W. A. Stein *et al.*, Sage mathematics software (version 6.0), The Sage Development Team, 2013, http://www.sagemath.org.

22. A. V. Sutherland, `smalljac` software library, version 4.0.23, 2013, http://math.mit.edu/∼drew/smalljac_v4.0.23.tar.

23. A. V. Sutherland, 'Structure computation and discrete logarithms in finite abelian p-groups', *Math. Comp.* 80 (2011) no. 273, 477–500; MR 2728991 (2012d:20112).

24. J. van der Hoeven, 'The truncated Fourier transform and applications', *ISSAC 2004* (ACM, New York, 2004) 290–296; MR 2126956.

25. J. van der Hoeven, 'Notes on the truncated Fourier transform', Technical Report 2005-5, Université Paris-Sud, Orsay, France, 2005, available at http://www.texmacs.org/joris/tft/tft-abs.html.

26. J. von zur Gathen and J. Gerhard, *Modern computer algebra*, 3rd edn (Cambridge University Press, Cambridge, 2013) MR 3087522.

27. N. Yui, 'On the Jacobian varieties of hyperelliptic curves over fields of characteristic $p > 2$', *J. Algebra* 52 (1978) no. 2, 378–410; MR 0491717 (58 #10920).

David Harvey
School of Mathematics and Statistics
University of New South Wales
Sydney, NSW 2052
Australia
d.harvey@unsw.edu.au

Andrew V. Sutherland
Department of Mathematics
Massachusetts Institute of Technology
Cambridge, MA 02139
USA
drew@math.mit.edu

https://doi.org/10.1112/S1461157014000187 Published online by Cambridge University Press