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

\[
w_{ij} = f_p^{(p-1)/2} \mod p \quad (1 \leq i, j \leq g),
\]

where \( f_p^k \) denotes the coefficient of \( x^k \) in \( f(x)^p \); see [7, 27]. We have the identity

\[
\chi(\lambda) \equiv (-1)^g \lambda^g \det(W_p - \lambda I) \mod p,
\]

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 \( p + 1 - \text{tr}(\text{Frob}_p) \) 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 \( \chi(\lambda) \mod p \)) for all primes \( p \leq N \) 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 \( p \leq N \), this is \( O(g^{8+\epsilon} \log^{4+\epsilon} p) \), the first
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\(^1\). 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 \( g \leq 3 \), where knowledge of \( \chi(\lambda) \mod p \) 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 \( \chi(\lambda) \mod p \) is \( O(p^{1/4+\epsilon}) \); while this is exponential in \( \log p \), within the feasible range of \( p \leq N \) (say \( N \leq 2^{32} \)), the time to derive \( \chi(\lambda) \) from \( \chi(\lambda) \mod p \) is actually negligible compared to the average time to compute \( \chi(\lambda) \mod p \). 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 \textsc{ComputeHasseWittMatrices}, which computes \( W_p \) for all admissible \( p \leq N \) (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 \( M(n) \) the time to multiply two \( n \)-bit integers. We may take \( M(n) = O(n \log n \log \log n) \), via [19].

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

\[
O(g^5 M(N \log(\|f\| N)) \log N),
\]

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 \textsc{hypellfrob} [9] and \textsc{smalljac} [22] libraries. In genus 2 the new algorithm outperforms both libraries for \( N \geq 2^{19} \), 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 \( W_p \)

\(^1\)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 \( p \leq N \) in any case.
for \( p \leq N \) 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 \textsc{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)

\[ C_n := VA_0 \cdots A_{n-1} \mod m_n, \]

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

We may apply \textsc{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 \( D_0 \cdots D_{n-1} \) is at most \( d \). Thus to obtain

\[ v_n = v_0 M_0 \cdots M_{n-1} / D_0 \cdots D_{n-1} \]

modulo \( p \), it suffices to compute

\[ v_0 M_0 \cdots M_{n-1} \mod p^{d+1} \quad \text{and} \quad D_0 \cdots D_{n-1} \mod p^{d+1}. \]

We run \textsc{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 \( 1 \times 1 \) 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 \( g \leq 3 \), 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 \( 1 \leq i \leq g \), consider the sequence of vectors

\[ v_n^{(i)} = [f_n^{2i+n+1}, \ldots, f_n^{2i+n-1}] \in \mathbb{Z}^r \quad (n \geq 0). \]
For each admissible prime $p = 2n + 1$, the last $g$ entries of $v_n^{(i)}$ are, modulo $p$, precisely the entries of the $i$th 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 $n \geq 0$, 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 $n \geq 0$, the identities

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

imply the relations

$$f_k^{n+1} = \sum_{j=0}^d f_j f_k^{-j}, \quad (3.1)$$

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

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

$$\sum_{j=0}^d (nj - k + j) f_j f_k^{-j} = 0 \quad (3.3)$$

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

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

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

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

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

$$v_n^{(i)} = [f_0^n, f_1^n, \ldots, f_{2in+i-1}^n].$$

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

$$v_{n+1}^{(i)} = [f_0^{n+1}, f_1^{n+1}, \ldots, f_{2in+3i-1}^{n+1}].$$
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

\[(n - k)f_i f_k^n = -\sum_{j=1}^{d-1} (n(j+1) - k + j)f_{j+1} f_{k-j}^n, \quad (3.6)\]

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

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

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

we use these relations to extend $v_n^{(i)}$ to the vector $[f_{2n+3i-2d+1}^n, \ldots, f_{2n+3i-1}^n]$ 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 $g \leq 3$.

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 $f(x) = f_4 x^4 + f_3 x^3 + f_2 x^2 + f_1 x + f_0$ with $f_0 f_1 \neq 0$; 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

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

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 $4 \times 4$ 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 $f_{2n}^n \mod p$, which is the last entry of $v_n \mod p$.

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

\[(2n + 1) f_0 f_{2n+1}^n = (2n + 3) f_4 f_{2n-3}^n + (n + 2) f_3 f_{2n-2}^n + f_2 f_{2n-1}^n - n f_1 f_{2n}^n.\]

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

\[(2n + 2) f_0 f_{2n+2}^n = (2n + 2) f_4 f_{2n-2}^n + (n + 1) f_3 f_{2n-1}^n - (n + 1) f_1 f_{2n+1}^n.\]
Combining these equations yields

$$2(2n + 1)\frac{d^2}{d^2} f_n^{2n+2} = -(2n + 3)f_1 f_4 f_n^{2n-3} + (2(2n + 1) - (n + 2)f_1 f_3) f_n^{2n-2} + ((2n + 1)f_0 f_3 - f_1 f_2) f_n^{2n-1} + nf_1^2 f_n.$$ 

Next we extend \( v_n \) ‘leftwards’ by applying (3.5) with \( k = 2n - 4 \), obtaining

$$(2n + 4)f_4 f_n^{2n-4} = -(n + 3)f_3 f_n^{2n-3} - 2f_2 f_n^{2n-2} + (n - 1)f_1 f_n^{2n-1} + 2nf_0 f_n.$$ 

With \( k = 2n - 5 \) we get

$$(2n + 5)f_4 f_n^{2n-5} = -(n + 4)f_3 f_n^{2n-4} - 3f_2 f_n^{2n-3} + (n - 2)f_1 f_n^{2n-2} + 2nf_0 f_n,$$

and therefore

$$(2n + 5)(2n + 4)f_4^2 f_n^{2n-5} = ((n + 3)(n + 4)f_3^2 - 3(2n + 4)f_2 f_4) f_n^{2n-3} + (2n + 4)f_1 f_4 f_n^{2n-2} + (-n - 1)(n + 4)f_0 f_3 + (2n - 1)(2n + 4)f_0 f_4) f_n^{2n-1} - 2n(n + 4)f_0 f_1 f_n^{2n}.$$ 

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

$$f_n^{2n-1} = f_4 f_n^{2n-5} + \ldots + f_0 f_n^{2n-1},$$

$$f_n^{2n+1} = f_4 f_n^{2n-2} + \ldots + f_0 f_n^{2n+2}.$$ 

After some algebraic manipulation we obtain the matrix

$$M_n = \begin{bmatrix} (-n + 3)f_3^2 + 4(n + 2)f_2 f_4 & J_1 & J_2 & J_3 & J_4 \\ -2f_2 f_3 + 6(n + 2)f_1 f_4 & J_1 & J_2 & J_3 & J_4 \\ (n - 1)f_1 f_3 + 8(n + 2)f_0 f_4 & J_1 & J_2 & J_3 & J_4 \end{bmatrix},$$

where

$$J_1 = (n + 1)(2n + 1)f_0,$$

$$J_2 = (n + 1)(2n + 1)(2n + 5)f_0 f_4,$$

$$J_3 = 2(n + 1)(n + 2)(2n + 5)f_0 f_4,$$

$$J_4 = (n + 2)(2n + 5)f_4,$$

and the denominator

$$D_n = 2n(2n + 1)(2n + 5)f_0 f_4.$$ 

Recall that \( v_n = v_0 M_0 \ldots M_{n-1}/(D_0 \ldots D_{n-1}) \). For each admissible prime \( p = 2n + 1 \geq 5 \), the \( p \)-adic valuation of \( D_0 \ldots D_{n-1} \) is exactly 1, since \( p \) divides \( D_{n-2} = 2n(2n-3)(2n+1)f_0 f_4 \) exactly once, and \( p \) does not divide \( D_j \) for \( j = n - 1 \) or any \( 0 \leq j \leq n - 3 \). We may thus compute \( v_n \mod p \) as

$$\left( \frac{v_0 M_0 \ldots M_{n-1} \mod p^2}{D_0 \ldots D_{n-1} \mod p^2} \right) \mod p.$$
With additional care it is possible to perform the bulk of the computation working modulo \( p \) rather than \( p^2 \). As noted above, \( D_0 \ldots D_{n-3} \) 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 \( v_n \mod p \), that is, the lone entry of the Hasse–Witt matrix \( W_p \), as

\[
\left( \frac{v_0M_0 \ldots M_{n-3} \mod p}{D_0 \ldots D_{n-3} \mod p} \right) U_n \mod p.
\]

**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 \( D_n = C f_0 \beta_d \prod_{i=1}^{e} (a_i n + b_i) \), where \( C, a_i, b_i \in \mathbb{Z} \) and \( \alpha, \beta \) and \( e \) are non-negative integers with \( \alpha + \beta \leq d \) and \( e \leq d \). 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 \( n \equiv -b_i/a_i \mod p \), and this occurs for at most one value of \( n \) in the interval \( 0 \leq n < (p-1)/2 \). 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 \( D_0 \ldots D_{n-1} \) 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 \( f_d = 0 \) 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 \( f(x) = f_3 x^3 + f_2 x^2 + f_1 x + f_0 \) and \( d = 3 \). Assuming \( f_0 \neq 0 \), we obtain the 3 \times 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 \( p = 2n + 1 \geq 5 \), the partial product \( D_0 \ldots D_{n-1} \) 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 \( y^2 = f_3 x^3 + f_2 x^2 + f_1 x \) and the 2 \times 2 transition matrix

\[
M_n = \begin{bmatrix}
(n+1)f_2 & 2(n+2)f_3 \\
2(n+1)f_1 & (n+2)f_2
\end{bmatrix}
\]
with denominator
\[ D_n = n + 2. \]
For all admissible primes \( p = 2n + 1 \) the partial product \( D_0 \cdots D_{n-1} \) 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 \( v^{(i)}_n \) for \( i = 1, 2 \). For the sake of brevity, we omit the details and list only the denominators \( D^{(i)}_n \); a Sage [21] script for generating the transition matrices \( M^{(i)}_n \) is available at [14].

For \( i = 1 \) we get the denominators
\[
D^{(1)}_n = \begin{cases} 
8(n + 2)(2n + 1)(2n + 3)(4n + 7)(4n + 9)f_0f_6^3 & \text{if } d = 6, f_0 \neq 0, \\
6(n + 2)(2n + 1)(3n + 5)(3n + 7)f_0f_2^5 & \text{if } d = 5, f_0 \neq 0, \\
3(n + 2)(3n + 4)(3n + 5)f_2^5 & \text{if } d = 5, f_0 = 0.
\end{cases}
\]

In the case \( d = 6 \), one verifies that the last two columns of \( M^{(1)}_{n-1}/D^{(1)}_{n-1} \) are \( p \)-integral for \( p = 2n + 1 \), and that \( D^{(1)}_0 \cdots D^{(1)}_{n-2} \) 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
\[
\left( \left( \frac{v_0M^{(1)}_0 \cdots M^{(1)}_{n-2} \mod p^2}{D^{(1)}_0 \cdots D^{(1)}_{n-2} \mod p^2} \right) \frac{M^{(1)}_{n-1}}{D^{(1)}_{n-1}} \right) \mod p.
\]

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
\[
D^{(2)}_n = \begin{cases} 
8(n + 3)(2n + 1)(2n + 5)(4n + 3)(4n + 5)f_0f_6^3 & \text{if } d = 6, f_0 \neq 0, \\
8(n + 4)(2n + 1)(4n + 3)(4n + 5)f_0^3f_2 & \text{if } d = 5, f_0 \neq 0, \\
3(n + 3)(3n + 2)(3n + 4)f_2^5 & \text{if } d = 5, f_0 = 0.
\end{cases}
\]

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
\[
D^{(1)}_n = \begin{cases} 
72(n + 2)(2n + 1)(2n + 3)(3n + 4)(3n + 5)(6n + 11)(6n + 13)f_0f_8^5 & \text{if } d = 8, f_0 \neq 0, \\
10(n + 2)(2n + 1)(5n + 7)(5n + 9)(5n + 11)f_2^3f_8 & \text{if } d = 7, f_0 \neq 0, \\
5(n + 2)(5n + 6)(5n + 7)(5n + 8)(5n + 9)f_2^5 & \text{if } d = 7, f_0 = 0.
\end{cases}
\]

For \( i = 2 \) the denominators are
\[
D^{(2)}_n = \begin{cases} 
8(n + 2)(2n + 1)(2n + 5)(4n + 3)(4n + 5)(4n + 7)(4n + 9)f_0f_8^3 & \text{if } d = 8, f_0 \neq 0, \\
24(n + 2)(2n + 1)(3n + 7)(3n + 8)(4n + 3)(4n + 5)f_0^3f_2^5 & \text{if } d = 7, f_0 \neq 0, \\
3(n + 2)(3n + 2)(3n + 4)(3n + 5)(3n + 7)f_2^3f_2^5 & \text{if } d = 7, f_0 = 0,
\end{cases}
\]
and for \( i = 3 \) they are

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

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

4. **Accumulating remainder trees**

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

\[
C_n := VA_0 \ldots A_{n-1} \mod m_n.
\]

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 \( \ell \) with nodes indexed by pairs \((i, j)\) with \( 0 \leq i \leq \ell \) and \( 0 \leq j < 2^i \). For each node we define

\[
m_{i,j} := m_{j2^\ell-i}m_{j2^\ell-i+1} \ldots m_{(j+1)2^\ell-i-1},
A_{i,j} := A_{j2^\ell-i}A_{j2^\ell-i+1} \ldots A_{(j+1)2^\ell-i-1},
C_{i,j} := VA_{i,0} \ldots A_{i,j-1} \mod m_{i,j}.
\]

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 \( 0 \leq j < b \). 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** \textsc{RemainderTree}

Given \( V, A_0, \ldots, A_{b-1} \) and \( m_0, \ldots, m_{b-1} \), 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 \( 0 \leq j < b \).
2. For \( i \) from \( \ell - 1 \) down to 1:
   - For \( 0 \leq j < 2^i \), 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 \( C_{0,0} = V \mod m_{0,0} \) and then for \( i \) from 1 to \( \ell \):
   - For \( 0 \leq j < 2^i \) set \( C_{i,j} = \begin{cases} C_{i-1,\lfloor j/2 \rfloor} \mod m_{i,j} & \text{if } j \text{ is even}, \\
C_{i-1,\lfloor j/2 \rfloor}A_{i,j-1} \mod m_{i,j} & \text{if } j \text{ is odd}.
\end{cases}\
\)

To illustrate the algorithm, let us compute \( (p-1)! \mod p \) 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 \textsc{RemainderTree} algorithm is concerned. We use odd moduli \( m_n = 2n + 1 \) for \( 0 \leq n < 8 \), except that we set the composite moduli \( m_4 \) and \( m_7 \) to 1, and we use \( 1 \times 1 \) matrices \( A_n = [(2n+1)(2n+2)] \) for \( 0 \leq n < 7 \), and let \( A_2 = [1] \) and \( V = [1] \). The trees \( m_{i,j}, A_{i,j}, \) and \( C_{i,j} \) computed by the \textsc{RemainderTree} algorithm are depicted below.
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 $m_0, \ldots, m_{b-1}$ and any entry in $A_0, \ldots, A_{b-1}$, and assume that $\log r = O(h)$. The running time of the RemainderTree algorithm is

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

and its space complexity is

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

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 $A_{j_1} \ldots A_{j_{t-1}}$ have bit-size $O((j_{t-1} - j_1) h + \log r)$; 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 $m_{0,0}$. \hfill $\Box$

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 $b = 2^k$ 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 $0 \leq s < 2^k$, we define the $s$th tree as follows. Let

$$m_{s}^j := m_{s+t}^j \quad (0 \leq j < t),$$

$$A_{s}^j := A_{s+t}^j \quad (0 \leq j < t),$$

$$V s := V A_{0} \cdots A_{t-1} \mod m_{s} \cdots m_{b-1}. $$

For $0 \leq i \leq \ell - k$ and $0 \leq j < 2^i$ 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

$$V s := V A_{k,0} \cdots A_{k,s-1} \mod m_{k,s} \cdots m_{k,2^k-1},$$

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

$$C_{0,0}^s = V s \mod m_{0,0}^s = V A_{k,0} \cdots A_{k,s-1} \mod m_{k,s} = C_{k,s}. $$
For the remaining nodes, the claim \( C_{i,j}^s = C_{i+k,j+2^s} \) follows by working downwards from the root of the \( C^s \) tree.

The idea of the \textsc{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

\[
Y^s := m_{s1} \ldots m_{sb-1}
\]
as we proceed from one subtree to the next. The \textsc{RemainderTree} algorithm is in step 2, each of which takes time \( O(B^2) \), and the output), and dominates the second term in the space bound of Theorem 4.1.

**Algorithm \textsc{RemainderForest}**

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

1. Set \( Y^0 \leftarrow m_0 \ldots m_{b-1} \) and \( V^0 \leftarrow V \) mod \( Y^0 \), and let \( t = 2^{\ell-k} \).

2. For \( s \) from 0 to \( 2^k - 1 \):
   a. Call \textsc{RemainderTree} with inputs \( V^s, A_{st}, \ldots, A_{(s+1)t-1} \), and \( m_{st}, \ldots, m_{(s+1)t-1} \)
      to compute trees \( m^s, A^s, C^s \).
   b. Set \( Y^{s+1} \leftarrow Y^s/m_{0,0}^s \) and \( V^{s+1} \leftarrow V^s A_{0,0}^s \) mod \( Y^{s+1} \).
   c. Output the values \( C_{st+j} = C_{st,j}^s \) for \( 0 \leq j < t \).
   d. Discard \( Y^s, V^s \), and the trees \( m^s, A^s, C^s \).

We now bound the complexity of the \textsc{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 \( \prod_{j=1}^{b-1} m_j \) such that \( B/2^k \) is an upper bound on the bit-size of \( \prod_{j=st}^{st+1-1} m_j \) 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 \( m_0, \ldots, m_{b-1} \) and any entry in \( A_0, \ldots, A_{b-1} \), and assume that \( \log r = O(h) \). The running time of the \textsc{RemainderForest} algorithm is

\[
O(r^3M(B + bh)(\ell - k) + 2^kr^2M(B) + rM(B'))
\]

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 \( O(M(B) \log b + rM(B + B')) \). There are \( 2^k \) calls to \textsc{RemainderTree} in step 2, each of which takes time

\[
O(r^3M(2^{-k} B + 2^{-k} bh)(\ell - k) + rM(B)),
\]

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 \( O(M(B) + r^2M(B + 2^{-k}bh)) \), thus each invocation of step 2 costs

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

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 \textsc{RemainderForest} algorithm has essentially optimal
Given a hyperelliptic curve \( C : y^2 = f(x) = \sum_{i=0}^{d} f_i x^i \) of genus \( g \), compute the Hasse–Witt matrices \( W_p \) for admissible primes \( p \leq N \) as follows.

1. For \( i \) from 1 to \( g \):
   a. Compute \( M^{(i)} \in \mathbb{Z}[n]^{r \times r} \), \( D^{(i)} \in \mathbb{Z}[n] \) satisfying \( v^{(i)}_{n+1} = v^{(i)}_n M^{(i)}(n)/D^{(i)}(n) \), as in \( \S \, 3 \).
   b. Use \texttt{ComputeHasseWittRows} below to compute the \( i \)th row of \( W_p \) for all \( p \in \mathcal{P} \).

2. Output the matrices \( W_p \).

As discussed in \( \S \, 3 \), 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 \( D_0 \ldots D_{n-1-w} \) for all sufficiently large admissible \( p \). For \( g \leq 3 \) using \( e = g \) and \( w \leq 3 \) suffices; in general \( e \) and \( w \) are both \( O(g) \). Our strategy is to compute the partial products \( M_0 \ldots M_{n-1-w} \) and \( D_0 \ldots D_{n-1-w} \) 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 \texttt{RemainderTree} algorithm, this means shifting the moduli \( m_j \) by \( w \) places to the left, relative to the \( A_j \).

### Algorithm \texttt{ComputeHasseWittMatrices}

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

1. Compute \( Y = \prod_{p \in \mathcal{P}} p^q \), let \( v_1 = 1 \), and let \( V \in \mathbb{Z}^r \) be the \((r - i + 1)\)th standard basis vector.
2. Fix \( k = 2 \log_2(e \sqrt{g}) + O(1) \), let \( t = 2^{e-k} \), and for \( s \) from 0 to \( 2^k - 1 \):
   a. For \( st \leq j < (s+1)t \), 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 \( st \leq j < (s+1)t + w - 1 \).
   c. Call \texttt{RemainderTree} with inputs \( V, M_j, m_j \) to compute \( C_j = V \prod_{u=0}^{j-1} M_u \mod m_j, m^s = \prod m_j, \) and \( M^s = \prod M_j \), where \( j \) ranges over integers from \( st \) to \( st + t - 1 \).
To compute a product
Customized FFT
implementation as described below. For the crucial
operation of multiplying matrices with very large integer entries, we used a customized FFT
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. 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 ∈ ℤ, we choose a parameter c ≥ 1 and write u = F(2^c) and
v = G(2^c), where F, G ∈ ℤ[x] have coefficients bounded by 2^c. We then compute the polynomial
product FG ∈ ℤ[x] and obtain uv as (FG)(2^c). To compute FG, we choose four suitable 62-bit
primes p_1, . . . , p_4 and compute FG mod p_i in (ℤ/p_iℤ)[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 . . . p_4. Multiplication in (ℤ/p_iℤ)[x] is achieved by using
Fourier transforms (number-theoretic transforms) over ℤ/p_iℤ. This requires p_i = 1 mod 2^{2^c},
where 2^{2^c} is the transform length. Our implementation uses optimized modular arithmetic as
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

| $g$ | 1 | 2 | 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 | 2 | 3 |
| 2^15 | 1 | 1 | 1 | 1 | 1 | 2 | 3 | 6 |
| 2^16 | 1 | 1 | 1 | 1 | 2 | 3 | 5 | 8 |
| 2^17 | 1 | 1 | 1 | 1 | 2 | 3 | 5 | 8 |
| 2^18 | 1 | 2 | 4 | 9 | 17 | 29 | 66 | 123 |
| 2^19 | 1 | 4 | 8 | 22 | 40 | 69 | 116 | 192 |
| 2^20 | 3 | 9 | 20 | 50 | 94 | 166 | 282 | 459 |
| 2^21 | 7 | 21 | 47 | 123 | 227 | 398 | 667 | 1085 |
| 2^22 | 17 | 49 | 114 | 287 | 534 | 946 | 1560 | 2540 |
| 2^23 | 38 | 115 | 268 | 645 | 1240 | 2230 | 3660 | 5940 |
| 2^24 | 89 | 271 | 641 | 1510 | 2920 | 5260 | 8490 | 13800 |
| 2^25 | 202 | 628 | 1470 | 3430 | 6740 | 11800 | 19600 | 31800 |
| 2^26 | 470 | 1475 | 3390 | 7930 | 15800 | 27400 | 44700 | 72900 | 107000 |
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 \textsc{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 \textsc{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 \textsc{hassewitt}) to the \textsc{smalljac} implementation described in [15]. In genus 2 the \textsc{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 \textsc{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 \( N \geq 2^{16} \). Table 5 compares the performance of the new algorithm to that of the \textsc{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 \textsc{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 \textsc{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 \( r = 2g \)).

| \( N \) | \( r = 2 \) | \( r = 3 \) | \( r = 4 \) | \( r = 4 \) | \( r = 5 \) | \( r = 6 \) | \( g = 3 \) |
|---|---|---|---|---|---|---|---|
| \( 2^{14} \) | <1 | <1 | <1 | <1 | <1 | <1 | 1 |
| \( 2^{15} \) | <1 | <1 | 1 | 1 | 1 | 4 | 6 |
| \( 2^{16} \) | 1 | 1 | 1 | 3 | 5 | 7 | 9 |
| \( 2^{17} \) | 3 | 6 | 10 | 3 | 4 | 18 | 25 |
| \( 2^{18} \) | 5 | 13 | 20 | 29 | 38 | 51 | 69 |
| \( 2^{19} \) | 11 | 27 | 41 | 59 | 79 | 106 | 144 |
| \( 2^{20} \) | 16 | 53 | 83 | 121 | 162 | 220 | 295 |
| \( 2^{21} \) | 32 | 108 | 169 | 249 | 332 | 450 | 610 |
| \( 2^{22} \) | 63 | 218 | 346 | 517 | 682 | 942 | 1258 |
| \( 2^{23} \) | 124 | 444 | 716 | 1064 | 1396 | 1940 | 2614 |
| \( 2^{24} \) | 247 | 920 | 1467 | 2195 | 2869 | 3980 | 5385 |
| \( 2^{25} \) | 498 | 1890 | 3014 | 3398 | 5865 | 8231 | 11162 |
| \( 2^{26} \) | 1002 | 3843 | 6478 | 6950 | 12134 | 12925 | 17137 |

### 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 | 468 | 540 | 736 | 1145 | 2055 |
| Space (MB) | 8529 | 4416 | 2215 | 1089 | 533 | 311 | 220 | 178 | 162 | 153 | 149 | 147 |
Table 4. Performance comparison with smalljac in genus 2. Times in CPU seconds.

| 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      | 28000    | 5260      | 31100    |
| 2^25| 3430      | 92200    | 6740      | 92300    | 11800     | 102000   |
| 2^26| 7930      | 314000   | 15800     | 316000   | 27400     | 349000   |

Table 5. Performance comparison with hypellfrob in genus 3. Times in CPU seconds.

| 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      | 16200      | 2540      | 16300      |
| 2^23| 3660      | 49400      | 5940      | 49400      |
| 2^24| 8490      | 152000     | 13800     | 152000     |
| 2^25| 19600     | 467000     | 31800     | 467000     |
| 2^26| 44700     | 1490000    | 72900     | 1490000    |

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).
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. A. SCHÖNHAGE and V. STRASSEN, ‘Schnelle multiplikation grosser zahlen’, Computing (Arch. Elektron. Rechnen) 7 (1971) 281–292; MR 0292344 (45 #1431).

19. 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).

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

21. A. V. SUTHERLAND, smalljac software library, version 4.0.23, 2013, http://math.mit.edu/~drew/smalljac-4.0.23.tar.

22. A. V. SUTHERLAND, ‘Structure computation and discrete logarithms in finite abelian $p$-groups’, Math. Comp. 80 (2011) no. 273, 477–500; MR 2728991 (2012d:11012).

23. 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.

24. J. VON ZUR GATHEN and J. GERHARD, Modern computer algebra, 3rd edn (Cambridge University Press, Cambridge, 2013) MR 3087522.

25. 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).