Computer Generation of Explicit Formulas for Hyperelliptic Curve Divisor Arithmetic

Amir Abbas Asgari, Haiyang He, Michael J. Jacobson Jr.(✉), and Renate Scheidler

University of Calgary, Calgary, AB, Canada
{amirabbas.asgari,jacobs,rscheidl}@ucalgary.ca

Abstract. The Jacobian of a hyperelliptic curve is the subject of many conjectures and open problems in algebraic geometry and number theory. Efficient Jacobian arithmetic plays a crucial role in the experimental investigation of these problems. For curves of low genus, explicit formulas effectively realize this arithmetic, but manually developing such formulas is not feasible in higher genera. We introduce the first software package for automating the process of generating and verifying explicit formulas for curves of arbitrary genus. Our formulas outperform generic algorithms for Jacobian arithmetic and exhibit comparable operation counts against manually fine-tuned formulas for small genera.

Keywords: Hyperelliptic curves · Jacobian arithmetic · Explicit formulas

1 Introduction

Since the introduction of elliptic and hyperelliptic curves for cryptographic purposes [13,14,17], significant effort has been extended to speed up the underlying group arithmetic via highly optimized formulas. The empirical investigation of prominent open problems and conjectures in algebraic geometry and number theory has further motivated improvements to hyperelliptic curve Jacobian arithmetic. This approach involves gathering extensive numerical data, aimed at offering evidence and insights to help formulate a proof of (or less likely, find a direct counterexample to disprove) these problems. Examples include questions pertaining to L-functions of abelian varieties over number fields, including hyperelliptic variants of the Birch and Swinnerton-Dyer conjecture, the Koblitz-Zywina conjecture, the Lang-Trotter conjecture, and the Sato-Tate conjecture [23] for elliptic curves. Other interesting questions include analogues to the Cohen-Lenstra heuristics [6] that describe the frequency of a given abelian group appearing as the ideal class group of a quadratic imaginary number field. Here, the function field version is the Friedman-Washington heuristics, which aim to characterize analogous behavior for quadratic function fields [1,7].

F. Boulier et al. (Eds.): CASC 2026, LNCS 16844, pp. 33–53, 2027.
https://doi.org/10.1007/978-3-032-34586-8_2

The generic arithmetic of the Jacobian of a ramified hyperelliptic curve of arbitrary genus over a base field of odd characteristic is realized by Cantor's algorithm [5], which is expressed in terms of polynomial arithmetic thanks to the Mumford representation [18] of group elements. This representation consists of a pair of univariate polynomials of bounded degree over the base field, which uniquely represents semi-reduced divisors on the curve.

Although there have been no asymptotic improvements to Cantor's algorithm since it was first proposed, significant practical improvements have been achieved, and our work continues in this direction. One important improvement is an adaptation of the NUCOMP algorithm [11], originally developed by Shanks [21] for composing binary quadratic forms. The core idea was to embed part of the *reduction* process of Cantor's algorithm within its *composition* phase and work with smaller intermediate polynomials and partial information (instead of full Mumford representations of auxiliary divisors) to obtain the final reduced divisor, the same as the output of Cantor's algorithm. The advantage of performing divisor arithmetic via NUCOMP was experimentally confirmed, especially for curves of higher genera, with a genus threshold around 7 or 8 [12] that was later improved to 6 [10].

Further improvements to Jacobian arithmetic were achieved via an approach known as *explicit formulas*. The idea is to transform the arithmetic on Mumford polynomials to operations on their coefficients in the base field, which opens the door to substantial computational improvements. Beginning with Harley's first practical set of such formulas [9] for hyperelliptic curves of genus 2, extensive effort has been extended to derive explicit formulas for curves of genera up to 4. These include highly optimized formulas in genera 2 and 3 with fewer advancements in the case of genus 4 due to the increased complexity of deriving explicit formulas as the genus grows. See Lindner [15] and the citations therein for a comprehensive list of sources.

The work of Nagao [19] is also significant because instead of providing straight-line formulas fine-tuned for one or a few particular genera, three improvements were proposed that are applicable to divisor arithmetic in arbitrary genus (although some of them are only effective under specific conditions on the genus). However, the operation counts reported in that work are based on numerical experiments over a limited number of prime fields, not a rigorous analysis of the formulas as found in the other aforementioned sources.

Continuing the process of manually deriving explicit formulas for curves of higher genera is simply infeasible. In particular, aiming for fully optimized explicit formulas covering all the possible cases (rather than only the most common case usually addressed in the literature), as undertaken by Lindner [15] for genera 2 and 3, is out of the question; the number of cases and the size of the formulas simply grow far too rapidly as the genus increases. This suggests a new and entirely different approach to generate such formulas via automation.

In this paper, we present a novel comprehensive algorithmic methodology for developing computer-generated efficient explicit formulas for Jacobian arithmetic on ramified hyperelliptic curves of arbitrary genus, including software that

implements this approach. This represents the very first foray into automation of explicit formula generation. We focus on affine formulas requiring a single field inversion as this is typically the preferred model for numerical conjecture testing [23], but our methods are certainly extendable to any other desirable model including split hyperelliptic curves and projective (inversion-free) models. We validate our formulas via extensive test suites and report in detail on our field operation counts and practical performance. Our automatically generated explicit formulas predominantly outperform the generic algorithms of Cantor's addition and NUCOMP in terms of runtime for curves of genera 2 to 10 over base fields of various sizes. Our operation counts are better than those reported by Nagao [19] in the case that a generic algorithm for Jacobian arithmetic is employed. Moreover, our operation counts remain within a reasonably close range when compared to hand-crafted explicit formulas, manually fine-tuned for limited cases of low-genus curves. The results of our white-box and black-box test suites (also produced automatically by the software based on its input parameters) ensure an extremely high level of reliability of the automated formulas, thanks to verifying each set of formulas against a vast number of concrete test cases.

2 Hyperelliptic Jacobians

For the mathematical background and further details, the reader is referred to [8,15,16] and [3, Chapter 2]. Throughout, let k be a perfect field, $\overline{k}$ its algebraic closure, and $k[x]$ the univariate polynomial ring over k. The leading coefficient of a polynomial $P(x) \in k[x]$ is denoted by $\mathrm{lcf}(P)$. A *(ramified) hyperelliptic curve C of genus g* defined over k is represented by a hyperelliptic equation

$$C : y^2 + \mathrm{h(x)}y = f(x) \tag{1}$$

that is irreducible in $k(x, y)$ and non-singular, with $f(x),\ \mathrm{h(x)} \in k[x]$, $\deg(f) = 2g + 1$ and $\deg(h) \leq g$.

A *divisor* D on C is a formal sum of points on C, written as $D = \sum_P n_P P$, where the sum runs over all points on C, including those with coordinates in $\overline{k}$ and the unique point at infinity, denoted ∞. Here, $n_P \in \mathbb{Z}$, and only finitely many n_P are non-zero. The *degree* of D is the integer $\deg(D) = \sum_P n_P$. Thus, every degree zero divisor D has the form $D = D_a - \deg(D_a)\infty$, where the sum of the *affine* divisor D_a runs over the points $P \neq \infty$ on C. For simplicity, we henceforth identify D with its affine part D_a and its degree with $\deg(D_a)$ (although the actual degree of D is of course zero).

A *principal* divisor is the divisor of a rational function $R(x, y)$ on C, where each n_P is the order of P as a pole or a zero of $R(x, y)$. Two divisors D_1, D_2 on C are said to be *(linearly) equivalent*, written $D_1 \equiv D_2$, if $D_1 - D_2$ is principal. We will henceforth only consider divisors *defined over k*, i.e., divisors that are fixed by the Galois group of $\overline{k}/k$. The *Jacobian* or *(degree zero) divisor class group* of C is the finite abelian group of linear equivalence classes of degree zero divisors defined over k under formal divisor addition. It generalizes the

notion of the group of points on an elliptic curve, and although its mathematical construction is somewhat involved, we will see that its elements have a simple representation as pairs of polynomials in $k[x]$.

In practice, Jacobian arithmetic involves two special types of divisors. For a *semi-reduced* divisor D with $D_a = \sum_P n_P P$, all n_P are non-negative and no subsum of D_a produces a principal divisor. In addition, D is *reduced* if $\deg(D_a) \leq g$. Every divisor class in the Jacobian of C contains a unique reduced divisor that is efficiently computable. Jacobian arithmetic can thus be performed on reduced divisors: to add two degree zero divisor classes, given by their respective unique reduced representatives D_1 and D_2, find the unique reduced divisor in the class of $D_1 + D_2$.

Every semi-reduced divisor D with $D_a = \sum_P n_P P$ can be uniquely represented by a pair of polynomials $u, v \in k[x]$, referred to as the *Mumford representation* of D and written as $[D] = [u, v]$. Here, the roots of u are exactly the x-coordinates of the points P appearing in D_a, with respective multiplicities n_P, and v Hermite-interpolates the points in D_a up to order n_P. In particular, $\deg(v) < \deg(u) = \deg(D_a)$. By way of the Mumford representation of reduced divisors, every degree zero divisor class is thus uniquely represented by a pair of polynomials $u, v \in k[x]$ with $\deg(v) < \deg(u) \leq g$, and Jacobian arithmetic can be realized via arithmetic on these polynomials. For example, the Jacobian of the genus 2 curve $C : y^2 + xy = x^5 + 2x + 1$ over $k = \mathbb{F}_3$ is a cyclic group of order 10, generated by the class of the reduced divisor $D = (-1 + i, 1) + (-1 - i, 1)$, where $i^2 = 2$. The Mumford representation of D is $D = [x^2 + 2x + 2, 1]$.

Cantor's algorithm [5] performs Jacobian arithmetic on Mumford representations in two stages. Given two reduced input divisors $[D_1] = [u_1, v_1]$ and $[D_2] = [u_2, v_2]$, *composition* produces a semi-reduced divisor $D = [u, v] \equiv D_1 + D_2$, which entails two (and generically only one) extended GCD computations on u_1, u_2 and $v_1 + v_2 + h$, with $h(x)$ given in (1). This is followed by *reduction* to obtain the unique reduced divisor equivalent to D, computed via a series of intermediate semi-reduced divisors from the continued fraction expansion of the function $(u + y)/v$ on C. Each continued fraction step decreases $\deg(u)$ by at least 2, so the reduced divisor equivalent to $D_1 + D_2$ is found after at most $\lceil g/2 \rceil$ reduction steps.

In [11], Shanks' NUCOMP algorithm [21] was introduced into Jacobian arithmetic to replace Cantor's arithmetic. It interleaves the composition and reduction stages, thereby avoiding the costly computation of $\lceil g/2 \rceil$ intermediate Mumford polynomials and operating on polynomials of smaller degree throughout. Algorithm 1 provides pseudocode of the NUCOMP variant used as the basis for the automated explicit formulas presented herein.

3 Explicit Formulas for Jacobian Arithmetic

Instead of working with Mumford polynomials, *explicit formulas* describe Jacobian arithmetic in terms of operations (i.e., additions, subtractions, multiplications and inversions) in the base field k, performed on the coefficients of

Algorithm 1: NUCOMP [15, Algorithm 19 with minor adaptations]

Input: $[u_1, v_1], [u_2, v_2], f, h$
Output: $[u, v] \equiv [u_1, v_1] + [u_2, v_2]$

if $\deg(u_1) < \deg(u_2)$ **then**
  Swap $[u_1, v_1]$ and $[u_2, v_2]$.
$t_1 = v_1 + h,\ t_2 = v_2 - v_1$.
$w_1 = (f - v_1(v_1 + h))\,/u_1$ (exact division).
$(S, a_1, b_1) = \mathrm{XGCD}(u_1, u_2)$ (only S and a_1 are needed).
$K = a_1 t_2 \pmod{u_2}$.
if $S \neq 1$ **then**
  $(S', a_2, b_2) = \mathrm{XGCD}(S, v_2 + t_1)$.
  $S = S'$.
  $K = a_2 K + b_2 w_1$.
  if $S \neq 1$ **then**
    $u_1 = u_1/S,\ u_2 = u_2/S$ (exact division).
    $w_1 = w_1 S$.
  $K = K \pmod{u_2}$.
if $\deg(u_1) + \deg(u_2) \leq g$ **then**
  $u = u_1 u_2.\ v = v_1 + u_1 K \pmod{u}$.
else
  $r = K,\ r' = u_2,\ c' = 0,\ c = -1,\ l = -1$.
  while $\deg(r) > (\deg(u_2) - \deg(u_1) + g)\,/2$ **do**
    $(q, r_n) = \mathrm{DivRem}(r', r)$.
    $r' = r,\ r = r_n,\ c_n = c' - qc,\ c' = c,\ c = c_n,\ l = -l$.
  $t_3 = u_1 r$.
  $M_1 = (t_3 + t_2 c)\,/u_2$ (exact division).
  $M_2 = (r(v_2 + t_1) + w_1 c)\,/u_2$ (exact division).
  $u' = l(rM_1 - cM_2)$.
  $z = (t_3 + c'u')/c$ (exact division).
  $v = z - t_1 \pmod{u'}$.
  $u = u'$, and then make u monic.
return [u, v].

these polynomials. Although they do not decrease the asymptotic complexity of Jacobian arithmetic, explicit formulas lead to significant practical improvements, in terms of both field operation counts and empirical runtime. They eliminate redundant calculations, re-use computations and effect other optimizations by utilizing an efficient combination of techniques to perform each block of polynomial-level calculations (see [15] for a survey of these techniques). Explicit formulas are the state-of-the-art methods for genus 2, 3 and 4, but producing them for higher genera is too labour-intensive to do by hand. In this section, we give an overview of the key techniques we use to generate explicit formulas for arbitrary genera using our automated method.

Given its advantages over Cantor's method, we chose NUCOMP as the base generic algorithm from which to derive our explicit formulas. This choice is

further supported by Lindner's work [15] that derived state-of-the-art explicit formulas for curves of genera 2 and 3 from NUCOMP. The overall idea is to transform Algorithm 1 into a loop-free set of instructions at the level of field operations. Following prior literature, we denote (non-constant) multiplication, addition, squaring, constant multiplication, and inversion by M, A, S, C, and I, respectively. Note that by constant multiplication we mean that at least one of the multiplicands is a coefficient of one of the defining polynomials f or h of the curve Eq. (1) or a fixed element of the base field k.

When converting the polynomial arithmetic operations in NUCOMP such as multiplication and division to explicit formulas, there are several choices of which algorithm to use for a given operation. Our software is designed to determine and use the optimal algorithms in terms of minimizing field operations subject to the genus of the curve under consideration. For polynomial multiplication and modular reduction, our software includes both the schoolbook and Karatsuba algorithms, but it is easily extendable to include others. The software also performs optimizations for exact divisions (when the remainder is known to be zero) and for reducing the number of required field inversions to just one inversion (at the expense of extra multiplications) via Montgomery's trick for simultaneous inversions.

Optimizing polynomial arithmetic described in terms of base field operations requires taking the comparative cost of different field operations into account. A performance trade-off between pairs of field operations (e.g., multiplication and addition) can be described via a threshold of the relative cost of one operation to the other. For instance, a threshold of 3 for the relative cost of a multiplication to an addition means that it is only beneficial to save 1 multiplication at the cost of at most 3 extra additions. The example below demonstrates how to compare and pick the best choice between the Karatsuba and schoolbook algorithms to perform the multiplication of two input polynomials of degree two.

Example 1. Let $A(x) = a_2x^2 + a_1x + a_0, B(x) = b_2x^2 + b_1x + b_0 \in k[x]$, and put $C(x) = A(x)B(x)$. Schoolbook multiplication computes the coefficients of $C(x) = c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0$ as

$$\begin{aligned} &c_4 = a_2b_2, \quad c_3 = a_2b_1 + a_1b_2, \\ &c_2 = a_2b_0 + a_1b_1 + a_0b_2, \quad c_1 = a_1b_0 + a_0b_1, \quad c_0 = a_0b_0, \end{aligned}$$

for a cost of $9M + 4A$. In contrast, employing Karatsuba multiplication, we first rewrite $A(x) = (a_2x + a_1)x + a_0$ and $B(x) = (b_2x + b_1)x + b_0$, so

$$\begin{aligned} C(x) = (a_2x + a_1)(b_2x + b_1)x^2 + ((a_2x + a_1 + a_0)(b_2x + b_1 + b_0) \\ - (a_2x + a_1)(b_2x + b_1) - a_0b_0)x + a_0b_0. \end{aligned}$$

Executing recursion once again for the intermediate subexpressions, we find that

$$\begin{aligned}
G(x) &:= (a_2x + a_1)(b_2x + b_1) \\
&= a_2b_2x^2 + ((a_2 + a_1)(b_2 + b_1) - a_2b_2 - a_1b_1)x + a_1b_1, \\
H(x) &:= (a_2x + a_1 + a_0)(b_2x + b_1 + b_0) \\
&= a_2b_2x^2 + ((a_2 + a_1 + a_0)(b_2 + b_1 + b_0) - a_2b_2 \\
&\quad - (a_1 + a_0)(b_1 + b_0))x + (a_1 + a_0)(b_1 + b_0), \\
F(x) &:= a_0b_0,
\end{aligned}$$

Writing $G(x) = g_2x^2 + g_1x + g_0$ and $H(x) = h_2x^2 + h_1x + h_0$, we obtain

$$c_4 = g_2, \quad c_3 = g_1 + h_2 - g_2, \quad c_2 = g_0 + h_1 - g_1, \quad c_1 = h_0 - g_0 - f_0, \quad c_0 = f_0.$$

This alternate method costs $7M + 16A$, so the cost difference between the two methods is $2M - 12A$. Thus, as long as $1M$ is more expensive than $6A$ in the base field, it is worth using Karatsuba's technique instead of the schoolbook method.

By an *exact division*, we mean that when dividing $A(x)$ by $B(x)$ we know *a priori* that the remainder is zero. The quotient $Q(x) = A(x)/B(x)$ is of degree $\deg(A) - \deg(B)$ and thus has $\deg(A) - \deg(B) + 1$ coefficients. The core idea to optimize an exact division (referred to as an *efficient* exact division) compared to a generic division with remainder is that only the top-most $\deg(A) - \deg(B) + 1$ coefficients of $A(x)$ are needed in the computation. The improvement lies in pruning unnecessary calculations *prior to* the point where the exact division is carried out. A concrete example is given by the frequent case in divisor composition, namely when $S = \gcd(u_1, u_2) = 1$ in Step 5 of Algorithm 1. Here, the polynomial w_1 computed in Step 4 is only used to obtain M_2 in Step 24. So the expected degrees of the dividend $r(v_2 + t_1) + w_1c$ and divisor u_2 in the exact quotient M_2 determine how many of the top-most coefficients of $f - v_1(v_1 + h)$, i.e., the dividend of w_1, must be computed.

Field inversions are especially expensive in finite fields, so Montgomery's simultaneous inversion trick effects substantial cost savings by replacing n inversions with one inversion and approximately $3n$ multiplications. Given n variables $x_1, \ldots, x_n$ that must be inverted, a stack is formed whose i-th entry is $y_i = x_iy_{i-1}$ (with $y_0 = 1$). After the n-th iteration, y_n is popped and the inversion y_n^{-1} is performed. Then the individual inverses are obtained as $x_i^{-1} = y_n^{-1}y_{i-1}z_i$ for $n \geq i \geq 1$ where y_{i-1} is popped and $z_i = x_{i+1}z_{i+1}$ (with $z_n = 1$). Incorporating Montgomery's trick into explicit formulas requires storing auxiliary information about which variables need to be inverted, so all such inversions can be carried out at once. This is referred to as a *weight* variable in the literature. The weight variable assigned to a polynomial indicates that all the coefficients of the polynomial must ultimately be multiplied by the inverse of that weight. Note, however, that weights of polynomials may vary through the calculations involving that polynomial and before arriving at the point of invoking Montgomery's trick for each execution path. Details about this behavior and the role of weights

in producing explicit formulas are illustrated in the example of schoolbook multiplication in Sect. 4.2.

In modular reduction, both the schoolbook and the Karatsuba algorithms offer the option of indicating whether to compute both the quotient and the remainder of the division or just the latter. Our implementation of Karatsuba modular reduction is based on Algorithm 4 in [24, Section 4.7]. However, this original algorithm only applies to reduction involving monic divisors/moduli and does not perform any weight handling. In our implementation, these limitations are addressed. Taking advantage of an idea similar to Karatsuba multiplication, our algorithm iteratively carries out consecutive divisions in which two coefficients of the final quotient are computed at each step, where the remainder of the last division is the desired remainder of the original division. Comprehensive technical details can be found in [3, Sections 3.4.3 and 5.2.4].

4 Software Design

Our software uses a templated description of NUCOMP (Algorithm 1) to generate source code in a specified programming language that implements explicit formulas for Jacobian arithmetic on an arbitrary ramified hyperelliptic curve of a specified genus. We refer to our main software as the *meta-program* to distinguish it from the code instances it generates. The source code, documentation, and some sample outputs of our software are available at https://csgit.ucalgary.ca/jacobs/explicit-formula-autogeneration. A second software package used for benchmarking the C++ implementation of the generated explicit formulas against generic algorithms, can be found at https://csgit.ucalgary.ca/jacobs/hyperelliptic-curve-arithmetic-library.git.

The inputs to our software, which we refer to as *hyperparameters*, currently include the target genus g, the relative cost η of base field multiplication to addition, the target programming language in which the final explicit formulas are produced (current options are Magma and C++), and a specification for the characteristic of the base field (2, odd, or arbitrary). The software compares different techniques to carry out the same polynomial-level primitive (e.g., multiplication of two polynomials), selects the best technique (i.e., the one with the least overall computational cost) based on the given ratio η and genus g, and generates the resulting explicit formulas accordingly.

The ultimate output of our software for a given genus g is an implementation of two functions, one to perform addition of Jacobian elements and a second for doubling. The inputs to these functions include the defining polynomials $f(x)$ and $h(x)$ of a hyperelliptic curve of genus g as in (1) and Mumford representations of two reduced divisors D_1 and D_2 (or just that of D_1 for doubling). Given these inputs, the automatically-generated explicit formulas output the reduced divisor equivalent to $D_1 + D_2$ for addition, and to $2D_1 = D_1 + D_1$ for doubling.

Each of these functions is in fact a director function that finds the degrees of the input divisors (i.e., $\deg(u_1)$ and, if applicable, $\deg(u_2)$) and invokes the appropriate subroutine for that combination of degrees. There is a total of

$g(g+1)/2$ and g subroutines in the addition and doubling formulas, respectively. Formula subroutines are named according to the operation they perform and the degrees of their input polynomials. For example, Deg21Add performs addition of a degree 2 and a degree 1 divisor on a genus 2 curve (see also Table 4). As per Step 1 of Algorithm 1, our top-level director function swaps the input divisors if needed, to ensure $1 \leq \deg(u_2) \leq \deg(u_1) \leq g$, before passing $[u_1, v_1]$ and $[u_2, v_2]$ on to the appropriate subroutine.

In the remainder of this section we describe how our software produces code implementing these functions.

4.1 Functionality

Our software produces language-independent representations of explicit formulas for each subroutine (corresponding to the degrees of the input divisors) by converting each step of NUCOMP (Algorithm 1), one by one, according to the hyperparameters g and η and the input divisor degrees. Separate functions depending on the degrees of the inputs (which can easily be determined as a function of the genus and input divisor degrees) for converting primitive polynomial operations including multiplication, efficient exact division, division with remainder, etc., are invoked as required during this process. These functions are realized as member functions of classes for each operation, thereby allowing the use of inheritance and polymorphism to readily incorporate additional algorithms for any given operation. Once explicit formulas for all required subroutines have been generated, the internal representations are converted to code in the target programming language; this design feature facilitates the easy inclusion of further target languages in the future.

Some of the main challenges in developing our software include managing conditional logic (which occurs both explicitly and implicitly; for example, in Steps 7 and 5 of Algorithm 1, respectively), weight handling of polynomials to ensure a single field inversion throughout each execution path, enumerating all possible scenarios that may occur through at most two XGCD calculations, and efficient exact division that requires back-propagation of information to avoid redundant calculations. We briefly explain the key parts of our contributions to address these and other problems when converting Algorithm 1 beyond Steps 1 and 2 into explicit formulas.

Prior to the explicit formula conversion process, the software performs a preprocessing step that determines, for every polynomial multiplication algorithm implemented in our software, the number of field operations required to multiply two polynomials of any degree potentially encountered during NUCOMP. In this way, whenever a polynomial multiplication occurs during the formula generation process, the optimal algorithm choice can be determined via table look-ups as opposed to generating formulas each time, thereby greatly improving the efficiency of the overall process. Furthermore, our software saves this data to a file, so subsequent invocations can simply read the data and augment it as necessary for input degrees that have not previously been processed. Note that only the raw operation counts are saved, allowing this data to be re-used even if η

(the relative cost of multiplication relative to addition in the base field) changes. The same methodology was applied to other polynomial operations supported by multiple algorithms, including variants of division with remainder that do or do not compute the quotient.

After pre-processing, all the formula subroutines are converted one by one. For each subroutine, the process begins by converting Step 3 of Algorithm 1, which involves a polynomial addition and a subtraction, by simply expanding these into operations on the coefficients of v_1, v_2 and h. Furthermore, an auxiliary polynomial $v_2 + t_1$ is computed, which is used in many execution paths at two later points in Algorithm 1.

Extended Euclidean Algorithm. After Step 3, NUCOMP requires one or two instances of the extended Euclidean algorithm (XGCD), depending on whether the first gcd value is 1. At this state, the degrees of the input polynomials are known. Thus, as the consecutive divisions with remainder (DivRem operations) are processed, appropriate conditional logic is created based on the degrees of the intermediate remainders. The chain of consecutive divisions terminates when the zero remainder is reached, making the second-to-last remainder the gcd of the input polynomials. Our approach is to track the degrees of the intermediate remainders and produce separate (possibly nested) conditional blocks for each case until arriving at the termination point.

The connection between the two XGCD instances is also taken into account. The explicit formulas to perform the second XGCD in Step 8 of Algorithm 1, whose first input polynomial is the result of the first XGCD (i.e., S in Step 5) and whose second input polynomial is $v_2 + t_1$, are generated only for the execution paths in which $S \neq 1$. Since the gcd of two polynomials is always monic, checking the latter condition is equivalent to checking $\deg(S) \neq 0$, which in turn can be determined by ascertaining whether all the consecutive coefficients of S except the constant coefficient, from highest to lowest, vanish.

The main loop of the XGCD algorithm is made explicit by enumerating all combinations of degrees of intermediate remainders that may arise throughout each XGCD calculation. Moreover, the relevant function also calls itself to perform the second XGCD if the first is ascertained to be nontrivial. Once the remainder in the XGCD calculation evaluates to zero, the gcd and Bézout coefficients are determined, and the relevant generated formulas are appended to the corresponding formula body.

Expansion of NUCOMP After XGCD. After performing the needed XGCD instances, the software must produce explicit formulas for the rest of NUCOMP, specifically Steps 4–29 of Algorithm 1 except for Steps 5, 7, 8, and 11. However, in the implementation, we do not always comply with the exact order of these steps to avoid as many unnecessary calculations as possible and improve the performance of the formulas. For example, instead of computing w_1 in Step 4, we postpone its partial calculation until execution paths are visited that actually use this polynomial. This happens when Steps 10 and/or 24 are taken, where

some (possibly all) of the top-most coefficients of w_1 are needed to perform the appropriate efficient exact division(s). A more detailed discussion of further such instances of re-ordering can be found in [3, Section 5.1].

A major challenge in this phase is handling the separate possible scenarios arising from the conditional logic of NUCOMP, which in particular includes the *if* block starting at Step 15 and the *while* block starting at Step 19, referred to as *the NUCOMP loop* from here on. Generating formulas for each scenario entails determining which variables need to be inverted at the appropriate point via Montgomery's trick, so we can ultimately produce the target polynomials (either the monic or the original weightless form) using their weighted versions and the calculated inverses. The remaining steps are mainly given as polynomial-level instructions (such as those beginning at Step 22), which are easier to convert to explicit formulas.

There are three top-level scenarios, according to whether the *if* condition in Step 15 holds, and if not, whether the NUCOMP loop is entered. In each case, appropriate conditional logic is generated, and the required polynomial arithmetic is made explicit as mentioned above. The NUCOMP loop is a truncated version of the extended Euclidean algorithm, and the same considerations as described above for making the XGCD computation explicit apply. For a comprehensive treatment of these three scenarios, their subcases, and the relevant calculations involved in each of them, the reader is referred to [3, Section 5.1.2].

As pointed out earlier, a significant challenge is to determine the single point at which Montgomery's trick of simultaneous inversions is invoked to obtain the weightless or monic versions of the required intermediate and target polynomials. The meta-program stores for each polynomial a (possibly empty) weight involving both a variable name and a vector of multiplicands with exponents constituting that weight. While performing each type of polynomial arithmetic (addition, multiplication, modular reduction, etc.), the weight of the resulting polynomial is obtained from the weight of the input polynomials (see [3, Section 3.4.5] for a thorough discussion on the relevant procedure for each operation). Assigning weights to polynomials instead of individual field elements is done for simplicity, increasing scalability, and reducing the amount of memory needed for handling weights, although more granular weight handling was previously used in manual approaches of developing explicit formulas such as [15].

A core idea of polynomial-level weight handling is that whenever coefficients of a polynomial $A(x)$ must be divided by a field element $b \in k$, we avoid immediately computing the inverse b^{-1} and instead append b to the current weight $w(A)$ of A. In this way, all the inversions are postponed until a later point at which they can all be performed at once using Montgomery's trick. For example, for obtaining the monic version of the polynomial S in Step 5 of Algorithm 1 (recall that polynomial gcd's are monic), we have $b = \mathrm{lcf}(S)$, and if S has a prior weight of $w_0(S)$, then its weight is updated to $w(S) = w_0(S)\mathrm{lcf}(S)$.

We have included mechanisms (through a hierarchy of classes and methods) that take as input coefficient representation of polynomial operands (also realized as specific objects in the design) and produce explicit formulas that per-

form schoolbook and Karatsuba algorithms for both multiplication and modular reduction. Each of the relevant functions is able to omit some of the coefficients that are no longer needed later during the overall computations of NUCOMP; this extra functionality is mainly provided to enable efficient exact division. For addition and subtraction, the schoolbook method is sufficient since it is optimal. Montgomery's trick is also implemented as a stand-alone generic function that takes as input an arbitrary number of field element variables and generates formulas that compute all their inverses.

4.2 An Example of Explicit Formula Generation

We illustrate how our software produces a loop-free explicit formula from a high-level description of a polynomial-level algorithm, using the example of schoolbook multiplication. Given two polynomials

$$A(x) = \sum_{i=0}^{n} a_i x^i, \quad B(x) = \sum_{j=0}^{m} b_j x^j,$$

we wish to compute

$$C(x) = \sum_{k=0}^{n+m} c_k x^k = A(x)B(x).$$

The idea is to first obtain all the terms $a_i b_j$ for $0 \le i \le n$ and $0 \le j \le m$, and store them in a two-dimensional array v where row v_k consists of all the terms $a_i b_j$ such that $i + j = k$. This implies

$$c_k = \sum_{l=0}^{|v_k|-1} v_{k,l},$$

where $|v_k|$ denotes the number of entries in v_k. After collecting all these terms into v_k, it only remains to add, for each $0 \le k \le n+m$, all the summands of c_k.

If $|v_k| = 0$ or 1, we assign $c_k = 0$ or $c_k = v_{k,0}$, respectively. Even if $|v_k| = 2$, we can still compute c_k with the single instruction $c_k = v_{k,0} + v_{k,1}$. But as soon as $|v_k| > 2$, we need to build up the formulas by accumulating the summands that ultimately produce c_k. Denoting by λ the global counter *label* of the formulas and by t a local counter that we further need, we have

$$\begin{aligned}
temp_{\lambda,t} &= v_{k,0} + v_{k,1} \\
temp_{\lambda,t+1} &= temp_{\lambda,t} + v_{k,2} \\
&\vdots \\
temp_{\lambda,t+|v_k|-3} &= temp_{\lambda,t+|v_k|-4} + v_{k,|v_k|-2} \\
c_k &= temp_{\lambda,t+|v_k|-3} + v_{k,|v_k|-1}.
\end{aligned}$$

We denote the weighted versions of A, B and C by $\hat{A}$, $\hat{B}$ and $\hat{C}$, respectively. Next, another function is invoked to assign the correct weight to $\hat{C}$. This function is independent of the specific algorithm used to carry out the multiplication. In Algorithm 2, we describe how weight handling is done to obtain the weight $w(\hat{C})$. The collection of variables that constitute the product of each weight variable is stored as a set of pairs whose first entry determines the variable and whose second entry is its exponent in the *weight factorization* of the weight variable. In the example of computing $\hat{A}(x)\hat{B}(x)$, if

$$w(\hat{A}) = \prod_{i=1}^{d} \alpha_i^{s_i}, \quad w(\hat{B}) = \prod_{j=1}^{e} \beta_j^{t_j},$$

then we have weight variable $w(\hat{A})$ and weight map $\{(\alpha_1, s_1), \ldots, (\alpha_d, s_d)\}$ for $\hat{A}$, and similarly $w(\hat{B})$ and $\{(\beta_1, t_1), \ldots, (\beta_e, t_e)\}$ for $\hat{B}$.

Algorithm 2: Weight handling in multiplication

Input: $w(\hat{A})$, weightMap$(\hat{A}) = \{(\alpha_1, s_1), \ldots, (\alpha_d, s_d)\}$, and $w(\hat{B})$, weightMap$(\hat{B}) = \{(\beta_1, t_1), \ldots, (\beta_e, t_e)\}$
Output: Create $w(\hat{C})$, weightMap$(\hat{C})$ for $\hat{C}(x) = \hat{A}(x)\hat{B}(x)$

```
1  weightMap(Ĉ) = {(α_1, s_1), ..., (α_d, s_d)}
2  for j = 1 to e do
3      if β_j == α_i for some 1 ≤ i ≤ d then
4          weightMap(Ĉ)[α_i] = s_i + t_j
5      else
6          weightMap(Ĉ)[β_j] = t_j
7  if w(Â) is empty then
8      if w(B̂) is empty then
9          w(Ĉ) = empty
10     else
11         w(Ĉ) = w(B̂)
12 else
13     if w(B̂) is empty then
14         w(Ĉ) = w(Â)
15     else
16         w(Ĉ) = w(Â)w(B̂)
```

In Step 1 of Algorithm 2, the weight map of $\hat{C}$ is initially populated with the weight map of $\hat{A}$. Depending on whether or not an entry in the weight map of $\hat{B}$ shares a common variable with some entry in the weight map of $\hat{A}$, the exponent of the corresponding entry in the weight map of $\hat{C}$ is increased accordingly (Step 4), or a new entry is created with the exponent coming only from $\hat{B}$ (Step 6).

Steps 7–16 handle the assignment of the weight variable $w(\hat{C})$ based on the weights of $\hat{A}$ and $\hat{B}$. The purpose is to avoid redundant multiplications in the formulas if either of the weight variables $w(\hat{A})$ or $w(\hat{B})$ is trivial (i.e., the corresponding weight map is empty).

5 Results

In this section, we present our results on the evaluation metrics of our software, including the field operation counts and empirical runtime, as well as our findings regarding black-box and white-box tests.

The experiments of this work were performed on an OptiPlex 7060 (085A) Dell computer with 16 GB of DDR4 memory (RAM), 384 KB of L1, 1536 KB of L2, and 12 MB of L3 caches, and the Intel Core i7-8700 CPU with frequency 3.20 GHz running on an Ubuntu 20.04.6 LTS Linux operating system. Moreover, version 11.5.1 of the Number Theory Library (NTL) [22] for C++ was used to implement our two polynomial-based algorithms (Cantor and NUCOMP), and for finite field arithmetic appearing in both these algorithms as well as in our automated explicit formulas. Version 6.2.0 of the GNU Multiple Precision Arithmetic Library (GMP) was used as the long integer package by NTL. We used the C++17 standard (indicated via `-std=C++17` among the compiler flags) and the GNU GCC compiler for C++ (aka g++) version 9.4.0 to compile our code.

5.1 Validation of Formulas

To provide the strongest possible assurance of correctness, we devoted substantial effort to developing test suites via both black-box and white-box testing.

White-Box Test Results. After producing explicit formulas, the meta-program creates test generators which are later used by a Python test driver to generate white-box test suites. Each test suite is a source file in the target output language consisting of multiple test cases, each of which contains the following:

- The base field $\mathbb{F}_q$, with q a prime power;
- The defining polynomials $f(x), h(x) \in \mathbb{F}_q[x]$ as in (1);
- Mumford representations $[u_1, v_1]$ and $[u_2, v_2]$ of two reduced input divisors D_1 and D_2 in case of addition (or just D_1 in case of doubling);
- A statement to assert that the result of our explicit addition/doubling algorithm with the given inputs matches the one produced by Cantor's algorithm.

The test generators attempt to find at least one test case covering every possible distinct execution path through the explicit formulas by producing random input divisors. We imposed a bound on the number of consecutive trials to avoid falling into an infinite loop, which could occur if there are execution paths that are in fact completely unreachable. This is to ensure that if no new execution paths are visited after a certain number of trials, the harness terminates the process and reports the achieved coverage ratio. The maximum number of trials was set to 25 in the experiments presented here.

The measure of success for our white-box testing is given by the ratio of visited execution paths to all potential such paths. We achieved full coverage of

Table 1. Coverage of white-box tests for addition and doubling formulas.

Genus	2	3	4	5	6
Addition	100%	100%	79.87%	74.68%	54.81%
Doubling	100%	100%	76.47%	82.73%	52.72%

Table 2. Number of potential execution paths for addition and doubling formulas.

Genus	2	3	4	5	6
Addition	14	59	298	1177	5696
Doubling	8	26	102	278	1286

Table 3. Timings for generating explicit formulas and the output file sizes for genera 2 to 8.

		CPU time to produce		Size of formulas	
Genus	char(k)	Addition	Doubling	Addition	Doubling
2	arbitrary	38.196 ms	19.776 ms	26.259 KB	15.911 KB
	2	35.019 ms	19.139 ms	25.169 KB	15.194 KB
	$\neq 2$	34.522 ms	19.063 ms	23.872 KB	14.202 KB
3	arbitrary	244.918 ms	103.494 ms	231.941 KB	108.319 KB
	2	243.868 ms	117.692 ms	226.833 KB	105.988 KB
	$\neq 2$	232.144 ms	98.726 ms	214.224 KB	98.906 KB
4	arbitrary	1.519 s	460.044 ms	1.671 MB	527.277 KB
	2	1.515 s	456.667 ms	1.654 MB	521.375 KB
	$\neq 2$	1.569 s	466.914 ms	1.625 MB	508.466 KB
5	arbitrary	7.914 s	1.832 s	9.546 MB	2.231 MB
	2	7.896 s	1.825 s	9.470 MB	2.212 MB
	$\neq 2$	7.738 s	1.758 s	9.164 MB	2.115 MB
6	arbitrary	45.623 s	9.236 s	55.482 MB	11.130 MB
	2	45.055 s	9.071 s	55.226 MB	11.082 MB
	$\neq 2$	42.011 s	8.541 s	51.386 MB	10.312 MB
7	arbitrary	3.449 min	32.825 s	262.755 MB	40.930 MB
	2	3.393 min	32.154 s	261.780 MB	40.771 MB
	$\neq 2$	3.255 min	30.571 s	244.220 MB	37.620 MB
8	arbitrary	17.320 min	2.278 min	1.196 GB	176.216 MB
	2	17.428 min	2.295 min	1.192 GB	175.737 MB
	$\neq 2$	15.912 min	2.230 min	1.148 GB	168.775 MB

all potential execution paths for both the addition and the doubling formulas in genera 2 and 3. Although we were unable to achieve the coveted 100% coverage for genera exceeding 3, our coverage percentages are still acceptable; our numerical data are provided in Tables 1 and 2. Not surprisingly, the data in Table 2 indicate that the total number of potential execution paths grows exponentially in the genus, as do the generation time and file size as observed in Table 3.

Performing white-box tests in this manner and analyzing the results helped us detect and exclude some theoretically unreachable paths from the explicit formulas. In addition to improving coverage ratios achieved in white-box tests, eliminating these redundant paths also leads to smaller file sizes for the final formulas. We were not able to eliminate all unreachable code with this preliminary attempt at static analysis, but two of the discovered unreachable code blocks (branches that can never be taken) are eliminated in the formula generation. To systematically address this problem, we augmented each object of field element variable with auxiliary information used to track the local state of that variable in different scopes of the formulas. Numerical results demonstrating our success at pruning unreachable execution paths can be found at [3, Section 6.2.1].

Black-Box Test Results. For our black-box tests, we ran extensive randomly-generated test cases for addition and doubling formulas for genera 2 to 8 (14 formula sets in total), all of which were successfully completed with no reported errors. Each test suite consisted of 50 rounds. Within a round, each of the 12 field sizes $q \in \{2, 3, 4, 5, 7, 8, 9, 11, 13, 16, 17, 19\}$ was selected once and a random hyperelliptic curve C of the given genus over the finite field $\mathbb{F}_q$ was chosen. For each curve C, 1000 random divisor additions and doublings in the Jacobian of C were performed and the results were verified against Magma's [4] built-in divisor arithmetic. For addition, the input divisors were explicitly chosen to be distinct in order to avoid doublings within the addition tests. All black-box tests passed without any errors. In total, our black-box test suite executed 600,000 different random test cases error-free, thus providing a very high degree of confidence in the correctness of the generated formulas.

5.2 Operation Counts

Table 4 lists the operation counts for the formulas generated by our software for various genera using the value of $\eta = 3$ for the multiplication/addition cost ratio, which is the threshold observed in prior works such as [23] and [15]. For simplicity, the M counts in the table represent the combined number of multiplications and squarings.

Nagao [19] reports operation counts of a straight-forward implementation of Cantor's addition, obtained as the maximal number of times some low-level multiplication and inversion functions are called within executions of their program for prime field sizes in a given range. In contrast, all our counts are based on automatic reports acquired directly from the symbolic representation of the generated formulas. Unlike Nagao's results, our operation counts are independent of the base field and only depend on the genus of the curve. The data in Table 4 compare our operation counts against Nagao's counts using Cantor's algorithm, the only source of operation counts for a wide range of genera. In all cases, the counts for our explicit formulas were significantly lower than Nagao's. This also agrees with our findings that empirically, our formulas outperform both generic algorithms (Cantor and NUCOMP), as reported in Sect. 5.3.

Table 4. Operation counts for divisor addition with char $k \neq 2$.

Input degrees	Deg22Add		Deg33Add		Deg44Add		Deg55Add	
Operation	M	I	M	I	M	I	M	I
Cantor's addition [19]	70	3	200	4	386	6	694	7
This work	35	1	115	1	195	1	344	1
Improvement	50.00%	66.67%	42.50%	75.00%	49.48%	83.33%	50.43%	85.71%
Best explicit	23	1	62	1	164	2	-	-

Deg66Add		Deg77Add		Deg88Add	
M	I	M	I	M	I
1054	9	1604	10	2186	12
475	1	708	1	914	1
54.93%	88.89%	55.86%	90.00%	58.19%	91.67%

For hyperelliptic curves of low genera, the best operation counts in prior literature are superior to Nagao's [19], due to comprehensive optimization of explicit formulas by hand (see [15] for a survey). The best known operation counts for genus 2 [15], genus 3 [2], and genus 4 [20] are also listed in Table 4. Note that the operation counts for genus 4 from [20] are for arbitrary characteristic fields, as that work did not consider the char $k \neq 2$ case. Although our counts are not as low as those for these manually fine-tuned formulas, they still lie in a comparable range. Our explicit formulas for genera $g \geq 5$, along with the methodology of automatic generation, are entirely new. For a more thorough comparison between the operation counts of our automated formulas and those of manually developed formulas in prior works, see [3, Section 6.3].

5.3 Empirical Benchmarking

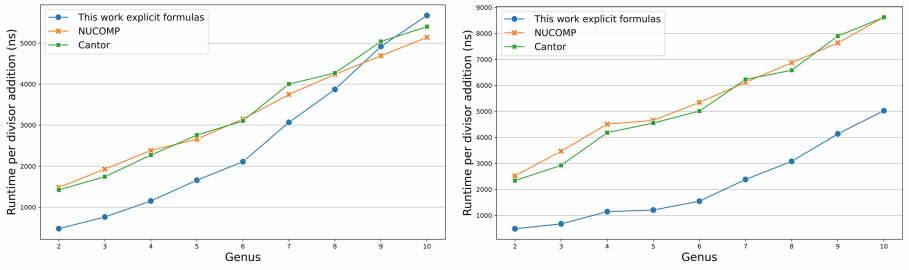
Lastly, we report on our experiments comparing the performance of the C++ implementation of our explicit formulas to implementations of Cantor and NUCOMP within the same environment and framework (same base libraries for finite field arithmetic, etc.). We tried multiple combinations of compiler flags, among which the set

```
-O3 -std=C++17 -mtune=native -march=native -ftls-model=initial-exec
```

yielded the best results. Note that although our software is capable of generating explicit formulas covering all cases for divisor arithmetic of each genus, the results provided in this section were obtained with a variant of our formulas that considers only the most frequent case within each subroutine ($S = 1$ in Algorithm 1 and no "unexpected cancellations" occur in intermediate calculations). Specifically, this modified implementation does not include explicit formulas for divisors not

conforming to these assumptions, and if such a special case is encountered, Cantor's addition is invoked as the baseline generic algorithm. The main reason is that the size of the full explicit formulas that include all the special cases grows very rapidly with the genus (as evident in Table 3), so compiling and running them becomes cumbersome, if not infeasible, for large genera. The methodology of restricting explicit formulas to the frequent case and simply calling Cantor's algorithm in the special cases only impacts performance for the smallest finite fields, as the probability of special cases occurring rapidly becomes negligible as q grows. Some partial results illustrating performance with full explicit formulas can be found in [3, Table 6.14].

Our numerical data, shown in Fig. 1, were obtained by comparing the performance of our automated explicit formulas against implementations of Cantor's addition and NUCOMP in the same experimental environment. For all three approaches, we performed additions in the Jacobian of curves defined over 5-bit and 32-bit finite fields, and obtained the same chain of divisors. For each bit length $l = 5$ or $l = 32$ of field sizes, $N_F = 10$ primes (not necessarily distinct) in the range $[2^{l-1}, 2^l - 1]$ were generated. Then $N_C = 100$ random curves over each corresponding prime-order finite field and $N_D = 10$ divisors per each curve were produced. Each consecutive pair of such divisors initiates a Fibonacci-style chain of divisor additions, with each chain having a predetermined length of 1000. In practice, $N_F N_C N_D \times 1000 = 10^7$ individual divisor additions take place via each approach and the average times per Jacobian operation are reported in Fig. 1.



(a) 5-bit field size

(b) 32-bit field size

Fig. 1. Average runtime of a divisor addition on two fields k of different sizes with $\mathrm{char}(k) \neq 2$ for the frequent case of each combination of input divisor degrees.

For curves over large finite fields, our automatically generated explicit formulas always outperformed the generic algorithms. For small finite fields, the performance of our formulas began to deteriorate past genus 8, due to the fact that the special divisor addition cases that resort to generic arithmetic are encountered with increased frequency, so the advantage gained by employing explicit formulas in the frequent cases diminishes. This can be alleviated by either handling all cases with explicit formulas or possibly reusing values already computed as opposed to restarting the generic algorithms from scratch.

6 Conclusion

This work introduces, for the first time, software based on a novel approach that fully automates the generation and verification of explicit formulas for Jacobian arithmetic on ramified hyperelliptic curves of arbitrary genus. The software was designed with extensibility in mind, enabling the integration of additional techniques to further optimize computational building blocks. Natural opportunities for improvement include employing faster multiplication algorithms based on Fast Fourier Transforms or the Toom-Cook divide-and-conquer approach, and using Newton iteration for modular reduction. Many hand-generated explicit formulas (see [15], for example) use resultants instead of the XGCD computations in Algorithm 1 (NUCOMP), and our software could be extended to do the same. Methods for detecting re-used expressions can also be incorporated to further decrease operation counts. Our framework also allows extensions to projective models that yield inversion-free formulas, split-model hyperelliptic curves and, in principle, Jacobians of non-hyperelliptic curves. Inversion-free formulas in particular should be easy to handle with the existing weight-handling mechanism, requiring very little additional computational cost if any.

Further analysis of conditional logic in NUCOMP, for example via static analysis or symbolic verification, may lead to a more complete characterization of unreachable execution paths in formulas, which could increase white-box test coverage toward 100% and strengthen the evidence supporting the correctness of the automatically generated formulas. Empirical data from the current implementation may provide valuable insights for this analysis. Eliminating more unreachable code is an important step in allowing us to scale the results to higher genera, as currently the sheer size of the formulas, not the runtime to produce them, is a significant limiting factor.

Acknowledgments. The first author acknowledges the support of an Alberta Graduate Excellence Scholarship (AGES), awarded during his studies at the University of Calgary. The third and fourth authors acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC). The authors also wish to acknowledge contributions of Jonah Richards toward development of a software package used for benchmarking the results of this work, as well as the anonymous referees.

Disclosure of Interests. The authors have no competing interests.

References

1. Achter, J.D.: Results of Cohen-Lenstra type for quadratic function fields. In: Computational Arithmetic Geometry, Contemporary Mathematics, vol. 463, pp. 1–7. Amer. Math. Soc., Providence (2008). https://doi.org/10.1090/conm/463/09041
2. Apperley, R.: Unpublished work on explicit formulas for genus 3 ramified hyperelliptic curves. undergraduate research project at the University of Calgary (2019)
3. Asgari, A.A.: Automated Explicit Formula Generation for Hyperelliptic Curve Jacobian Arithmetic, University of Calgary. Master's thesis (2025)

4. Bosma, W., Cannon, J., Playoust, C.: The Magma algebra system. I. The user language. J. Symb. Comput. **24**(3-4), 235–265 (1997). https://doi.org/10.1006/jsco.1996.0125
5. Cantor, D.G.: Computing in the Jacobian of a hyperelliptic curve. Math. Comput. **48**(177), 95–101 (1987). https://doi.org/10.2307/2007876
6. Cohen, H., Lenstra, H.W.: Heuristics on class groups of number fields. In: Jager, H. (ed.) Number Theory Noordwijkerhout 1983. LNM, vol. 1068, pp. 33–62. Springer, Heidelberg (1984). https://doi.org/10.1007/BFb0099440
7. Friedman, E., Washington, L.C.: On the distribution of divisor class groups of curves over a finite field. In: Théorie des nombres (Quebec. PQ, 1987), pp. 227–239. de Gruyter, Berlin (1989)
8. Galbraith, S.D.: Mathematics of Public Key Cryptography. Cambridge University Press, Cambridge (2012). https://doi.org/10.1017/CBO9781139012843
9. Gaudry, P., Harley, R.: Counting points on hyperelliptic curves over finite fields. In: Bosma, W. (ed.) ANTS 2000. LNCS, vol. 1838, pp. 313–332. Springer, Heidelberg (2000). https://doi.org/10.1007/10722028_18
10. Imbert, L., Jacobson, M.J., Jr.: Empirical optimization of divisor arithmetic on hyperelliptic curves over $\mathbb{F}_{2^m}$. Adv. Math. Commun. **7**(4), 485–502 (2013). https://doi.org/10.3934/amc.2013.7.485
11. Jacobson, M.J., van der Poorten, A.J.: Computational aspects of NUCOMP. In: Fieker, C., Kohel, D.R. (eds.) ANTS 2002. LNCS, vol. 2369, pp. 120–133. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45455-1_10
12. Jacobson, Jr., M.J., Scheidler, R., Stein, A.: Fast arithmetic on hyperelliptic curves via continued fraction expansions. In: Advances in Coding Theory and Cryptography, Ser. Coding Theory Cryptology, vol. 3, pp. 200–243. World Sci. Publ., Hackensack (2007). https://doi.org/10.1142/9789812772022_0013
13. Koblitz, N.: Elliptic curve cryptosystems. Math. Comput. **48**(177), 203–209 (1987). https://doi.org/10.2307/2007884
14. Koblitz, N.: Hyperelliptic cryptosystems. J. Cryptol. **1**(3), 139–150 (1989). https://doi.org/10.1007/BF02252872
15. Lindner, S.A.: Improvements to Divisor Class Arithmetic on Hyperelliptic Curves, University of Calgary. Ph.D. thesis (2020)
16. Menezes, A., Zuccherato, R., Wu, Y.H.: An elementary introduction to hyperelliptic curves (1996). https://www.uwaterloo.ca/scholar/sites/ca.scholar/files/ajmeneze/files/hyperelliptic.pdf
17. Miller, V.S.: Use of elliptic curves in cryptography. In: Williams, H.C. (ed.) CRYPTO 1985. LNCS, vol. 218, pp. 417–426. Springer, Heidelberg (1986). https://doi.org/10.1007/3-540-39799-X_31
18. Mumford, D.: Tata lectures on theta. II, Progress in Mathematics, vol. 43. Birkhäuser Boston, Inc., Boston (1984). https://doi.org/10.1007/978-0-8176-4578-6
19. Nagao, K.: Improving group law algorithms for jacobians of hyperelliptic curves. In: Bosma, W. (ed.) ANTS 2000. LNCS, vol. 1838, pp. 439–447. Springer, Heidelberg (2000). https://doi.org/10.1007/10722028_28
20. Pelzl, J., Wollinger, T., Paar, C.: Low cost security: explicit formulae for genus-4 hyperelliptic curves. In: Matsui, M., Zuccherato, R.J. (eds.) SAC 2003. LNCS, vol. 3006, pp. 1–16. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24654-1_1
21. Shanks, D.: On Gauss and composition. I, II. In: Number Theory and Applications (Banff, AB, 1988), NATO Advanced Science Institutes Series C: Mathematical and

Physical Sciences, vol. 265, pp. 163–178, 179–204. Kluwer Acad. Publ., Dordrecht (1989)
22. Shoup, V.: Number Theory Library (NTL) (2021). https://github.com/libntl/ntl
23. Sutherland, A.V.: Fast Jacobian arithmetic for hyperelliptic curves of genus 3. In: Proceedings of the Thirteenth Algorithmic Number Theory Symposium. Open Book Series, vol. 2, pp. 425–442. Math. Sci. Publ., Berkeley (2019)
24. Wollinger, T.J., Pelzl, J., Paar, C.: Cantor versus Harley: optimization and analysis of explicit formulae for hyperelliptic curve cryptosystems. IEEE Trans. Comput. **54**(7), 861–872 (2005). https://doi.org/10.1109/TC.2005.109