IAS/Park City Mathematics Series
Volume 00, Pages 000–000
S 1079-5634(XX)0000-0

Counting points on hyperelliptic curves over finite fields

David Harvey

Abstract. This course is a gentle introduction to my paper "Computing zeta functions of arithmetic schemes" (2015). We walk through the algorithms of that paper in detail for the special case of a hyperelliptic curve over a prime field. In particular, we explain how to obtain "square-root time" and "average polynomial time" point counting algorithms for such a curve. The presentation is "elementary" in the sense that it does not rely on any cohomology.

Contents

1 Introduction 1
2 A crash course on fast arithmetic 6
3 Hyperelliptic curves and zeta functions 11
4 A modulo p trace formula 17
5 Recurrences for polynomial powers 22
6 A square-root time algorithm 28
7 An average polynomial time algorithm 35
8 A modulo p^λ trace formula 47

1. Introduction

1.1. What this course is about This course has two main aims. The first is to provide a gentle introduction to some of the ideas behind my paper "Computing zeta functions of arithmetic schemes" [23]. That paper presented a number of algorithms for counting points on algebraic varieties over finite fields, or in other words, counting the number of solutions to systems of polynomial equations over finite fields. In this course we will specialise the methods of [23] to the case of a *hyperelliptic curve over a prime field*, i.e., a curve defined by an equation of the form $y^2 = f(x)$ over a field $\mathbf{F}_p$ where p is an odd prime. (Here "hyperelliptic" includes the case of elliptic curves; see Remark 3.1.2.) Here is a baby example of the kind of problem we consider in this course:

2010 *Mathematics Subject Classification.* Primary 14Dxx; Secondary 14Dxx.
Key words and phrases. Park City Mathematics Institute.

©0000 (copyright holder)

Problem 1.1.1. Count (by hand) the number of solutions $(x, y) \in \mathbf{F}_3 \times \mathbf{F}_3$ to the equation $y^2 = x^3 + x$.

The second aim of this course is to serve as a jumping-off point for further research. Sprinkled throughout these notes are sketches of a number of unpublished point counting algorithms, and also a few other algorithms unrelated to point counting. My hope is that someone reading these notes will be sufficiently intrigued by one of these sketches to take up the challenge of fleshing out the details, writing an efficient implementation, and publishing the results.

1.2. What this course is not about In this course I will have almost nothing to say about the many other approaches to point counting that have been developed over the last few decades. These include algorithms based on analysing the group structure of the Jacobian (see for example [15, §2] or [51]) and algorithms based on ℓ-adic cohomology [43, 46]. There are some links between the approach taken in this course and algorithms based on p-adic cohomology, such as [31, 53], but I will make only a few brief comments on the latter. One should keep in mind that it is often possible to combine information obtained from these different classes of algorithms, and the compound algorithm may well be more efficient than any of its components considered separately. Finally, I will also have nothing to say about point counting algorithms designed to run on quantum computers [32].

This course is also not about *applications* of point counting. There are plenty of applications in cryptography, coding theory and number theory — see for example [10], [47], and [17] respectively.

The literature on the above topics is vast and I cannot even begin to offer a survey here.

1.3. Outline of the course Section 2 is a "cheat sheet" that summarises the tools we need from computer algebra, such as fast algorithms for integer and matrix multiplication. You may wish to skip this section on a first reading, and refer back occasionally when needed.

Section 3 introduces the main objects of study in this course — hyperelliptic curves and zeta functions — and works out some of their basic properties.

Section 4 develops a "trace formula" that is central to all algorithms discussed subsequently. This formula yields a congruence modulo p for the number of $\mathbf{F}_{p^r}$-rational points on a given hyperelliptic curve $y^2 = f(x)$ over $\mathbf{F}_p$, in terms of certain selected coefficients of $f^{(p-1)/2}$. Already in this section we will be able to construct point counting algorithms that significantly outperform the naive "enumeration" method.

In Section 5 we establish a *recurrence* for the coefficients of $f^{(p-1)/2}$. This recurrence feeds into the following two sections to obtain even faster point counting algorithms: in Section 6 we see how to obtain a "square-root time" point counting algorithm, whose complexity is roughly linear in $p^{1/2}$, and in Section 7 we obtain

an "average polynomial time" algorithm, which reduces the complexity further to $O(\log^4 p)$, when averaged over many primes.

All of the algorithms discussed up to this point in this course have only computed the point counts modulo p. In Section 8 we develop a more powerful trace formula that yields the point counts modulo *powers* of p. Combined with the methods of Sections 5 and 6, this leads to various algorithms for computing the entire zeta function, including algorithms that run in "square-root time" and "average polynomial time". These algorithms are not worked out in detail, but are instead left as challenging exercises for the reader.

1.4. Problems These course notes include many problems of various sorts.

A large fraction of the problems ask you to supply proofs of the various Theorems, Propositions and Lemmas. These are an integral part of the course and I strongly recommend that you attempt as many as possible.

A problem marked with the game controller icon (🎮) is an implementation problem, to be carried out in your *favourite computer algebra system* (FCAS). Examples include Sage, Magma, Maple, Mathematica and Minecraft. I try to use Sage where possible because I like to support free software. Personally, I find that one of the best ways to understand an algorithm deeply is to implement it — the machine is unforgiving and will not let you cheat. If you don't know how to use these systems, or have never done any programming before, now is an excellent time to begin learning! Some of the implementation problems are fairly substantial projects, so you might consider working in a team.

The lightning bolt icon (🗲) indicates a problem that I don't know how to solve. This does not necessarily mean that the problem is difficult or unsolved — it might only reveal some ignorance on my part, or that I haven't spent much time thinking about it yet. Two lightning bolts (🗲🗲) mean that I do not know how to solve the problem, despite having devoted considerable thought and attention to it. Again, this *may* mean that it is difficult or even impossible, but not necessarily!

1.5. Prerequisites and notation I assume that you are familiar with basic algebraic notions such as groups, rings, fields, polynomials, the Chinese remainder theorem, and the rudiments of linear algebra over a field. We write $\#S$ for the cardinality of a set S, and $\overline{\mathbb{K}}$ for the algebraic closure of a field $\mathbb{K}$. For a ring R, we write $R[\![x]\!]$ for the ring of (formal) power series with coefficients in R, and $\mathrm{Mat}_d(R)$ for the ring of $d \times d$ matrices with entries in R.

If F is a polynomial in one variable, we write F_i for the coefficient of x^i in F. By convention $F_i = 0$ if $i < 0$ or $i > \deg F$.

The symbol p always denotes a prime. I assume that you know elementary number-theoretic facts such as Fermat's little theorem, the fact that $(x+y)^p = x^p + y^p$ in characteristic p, the structure of finite fields, and how to solve congruences modulo powers of a prime. (We will not actually use p-adic numbers anywhere, but it won't hurt to know about them!)

We write $\mathbf{F}_q$ for the field with q elements. Assuming that q is a power of an *odd* prime, we define

$$\chi_q : \mathbf{F}_q \to \{0, 1, -1\} \subset \mathbf{Z}$$

to be the quadratic character on $\mathbf{F}_q$, i.e.,

$$\chi_q(\beta) \coloneqq \begin{cases} 0 & \beta = 0, \\ 1 & \beta \text{ is a square in } \mathbf{F}_q^*, \\ -1 & \beta \text{ is a non-square in } \mathbf{F}_q^*. \end{cases}$$

In particular, $\chi_p(\beta)$ is the same as the Legendre symbol (β/p). The identity

$$\beta^{(q-1)/2} \equiv \chi_q(\beta) \pmod{p} \qquad \text{for any } \beta \in \mathbf{F}_q \tag{1.5.1}$$

plays a central role throughout the course.

The following result is equivalent to the Prime Number Theorem; see for instance [40, Thm. 6.9].

Lemma 1.5.2. *For $N \geqslant 2$ we have*

$$\sum_{p \leqslant N} \log p = N + o(N).$$

(Whenever we take sums or products over the symbol p, we always mean over primes. The symbol $o(N)$ means a function $f(N)$ such that $\lim_{N\to\infty} f(N)/N = 0$.)

We will need only the most basic notions from algebraic geometry. We write $\mathbf{A}^n_{\mathbb{K}}$ and $\mathbf{P}^n_{\mathbb{K}}$ for the affine and projective spaces of dimension n over a field $\mathbb{K}$. If X is a variety over $\mathbb{K}$ and $\mathbb{L}$ is an extension of $\mathbb{K}$, we write $X(\mathbb{L})$ for the set of $\mathbb{L}$-rational points on X.

I assume that you are comfortable with big-O notation. As a brief reminder, we write $f(n) = O(g(n))$, or occasionally $f(n) \ll g(n)$, to mean that there exists an absolute constant $C > 0$ such that $f(n) \leqslant Cg(n)$ for all n in the relevant domain. If $g(n)$ is negative or undefined for small values of n, for example, $g(n) = n \log n \log\log n$, these values of n are tacitly excluded from the domain. When we write $f(n) = O(n^{1+\varepsilon})$, this means that the bound holds for all $\varepsilon > 0$, where the implied constant may depend on ε.

1.6. Algorithms and computational complexity I will use the terms "running time" and "complexity" interchangeably. By the running time of an algorithm, I mean the number of "bit operations" that it performs. I assume that you have some intuitive understanding of what this means, but in order to prove *theorems* about the complexity of algorithms, we need to settle on a precise computational model. In this course, the model I will always have in the back of my mind is the *deterministic multitape Turing model*. Briefly, this means a Turing machine with a fixed, finite number of tapes, and the complexity refers to the number of steps executed by the machine over the course of a computation. I do not want to get sidetracked by the technical details of this; for a formal presentation, see [42].

I will however mention one "gotcha" for those experienced programmers who may not have studied formal computational models before. In the multitape Turing model, one does not have available constant-time random access to elements of an array. In fact, accessing the n-th element of an array of b-bit elements requires time $O(nb)$, as the tape head has to travel all the way to the target slot.

In this course we will mainly focus on time complexity, and I will not give formal estimates for space complexity, i.e., the amount of memory used by a computation, due to lack of space. However, for certain algorithms the space complexity becomes the main bottleneck in practice, and I will point this out when relevant. We will also completely ignore the issue of parallelisation, which again is very important in practical computations.

1.7. Some thoughts about the big picture Before going on, I would like to offer some personal reflections concerning the status of the paper [23] on which this course is based.

My own point counting education began in the p-adic cohomology world, mainly in the vicinity of Kedlaya's algorithm [31]. The development of the algorithms in [23] was heavily influenced by this background. Over the years, I have formed the definite impression that the approach to point counting in [23] is in some deep way *the same* as the p-adic cohomological approach. This is a very vague gut feeling and I do not know how to make it precise. I certainly do not know how to take an algorithm expressed in terms of p-adic cohomology and "translate" it into the setting of [23], or vice versa. And I do not know any cohomological interpretation of the "trace formula" [23, Theorem 3.1] that lies at the heart of the method. (In these notes, the closest analogue of this formula is Theorem 8.4.1.) Yet the feeling persists that the two approaches are somehow the same thing in disguise.

What is most frustrating is that each approach seems capable of doing things that the other cannot. I will give a few examples.

In one direction, the methods of [23] seem to lead to much more general results than the p-adic cohomology approach. For example, [23] presents "square root time" and "average polynomial time" point counting algorithms for completely arbitrary varieties. (For the special case of hyperelliptic curves, these results are worked out in detail in §6 and §7 of these notes.) We do not know any other way of proving these extremely general results; in the p-adic cohomology context, we can achieve results of a similar strength for only a rather limited class of varieties. Another example is the recent work of my student Madeleine Kyng, who has designed an algorithm that uses the setup in [23], together with some integral closure calculations, to count points on completely arbitrary curves [35]. Her experiments confirm that this algorithm can confidently handle curves that are apparently intractable by other known methods, including Tuitman's algorithm [53], which is currently the most general variant of Kedlaya's algorithm known.

In the other direction, the p-adic cohomology approach seems capable of producing better complexity bounds than [23] in many cases. For example, for a curve of large genus g over a small field $\mathbf{F}_p$, algorithms based on [23] tend to get bogged down when computing powers of the unreasonably large matrices appearing in the "trace formula". (For the corresponding phenomenon in these notes, consider the terms with large values of ℓ in (8.4.2).) Both approaches yield complexity bounds polynomial in g, but the exponent of g is usually somewhat smaller for the p-adic cohomology approach.

The complexity advantage for the p-adic cohomology algorithms is even more dramatic in higher dimensions, at least in the cases that these algorithms are applicable at all. For example, when using [23] to handle a quartic surface in $\mathbf{P}^3$ (a K3 surface), one needs to run the algorithm with a very high "target precision" (in these notes, the parameter λ in §8), making it essentially infeasible in practice. On the other hand, the algorithm in [1] needs only one or two p-adic digits, and is eminently feasible. The difference here seems to be that the p-adic cohomology algorithms have more direct access to arithmetic properties of the Frobenius action on cohomology. Another example is Lauder's deformation algorithm [36], which for fixed p can handle a smooth projective hypersurface of degree d in n variables in time polynomial in d^n. Using [23], one achieves complexity only polynomial in the rather larger quantity d^{n^2}. Again, Lauder is using properties of Frobenius that are invisible from the perspective of [23].

I am hopeful that one day the two approaches will be unified, and that we will have available a conceptual framework that allows us to design algorithms achieving the best of both worlds: the generality and flexibility of [23], and the better complexity bounds coming from p-adic cohomology.

Acknowledgments Many thanks to Alex Best for his comments on a draft of these notes.

2. A crash course on fast arithmetic

Many of the algorithms presented in this course rely on fast algorithms for arithmetic on objects such as integers, polynomials and matrices. Your computer can add, subtract and multiply integers up to about 64 bits long in constant time; these operations are hard-wired into the silicon. But if you ask your FCAS to perform arithmetic on integers larger than this, it has to be done in software, and it becomes crucial to have available algorithms whose complexity scales well for large inputs. The study of such algorithms belongs to a field known as *symbolic computation* or *computer algebra*.

This section summarises the complexity results that we need, without going into the details of the underlying algorithms. The details can be found in many sources, such as [5, 8, 34, 54].

2.1. Integer arithmetic We assume that integers are represented in the usual binary notation, with perhaps an extra bit to indicate the sign. (These sorts of implementation details do not affect the complexity results, but it is good to get into the habit of thinking about how to represent various mathematical objects explicitly on a Turing machine tape.)

Addition and subtraction. We can add and subtract n-bit integers in time $O(n)$, by working from the least significant bit to the most significant, propagating carries (or borrows) as we go.

Multiplication. We write $M_{\mathrm{int}}(n)$ for the cost of multiplying n-bit integers. Sometimes we will write expressions like $M_{\mathrm{int}}(\log p)$, where $\log p$ might not be an integer. For convenience, we interpret this to mean $M_{\mathrm{int}}(\lceil \log p \rceil)$.

The classical long multiplication algorithm runs in time $O(n^2)$. Currently, the best known asymptotic bound for $M_{\mathrm{int}}(n)$ is

$$M_{\mathrm{int}}(n) = O(n \log n).$$

This bound is achieved via a much more complicated algorithm involving the fast Fourier transform [28], and is widely believed to be optimal.

Problem 2.1.1. (🎮) Investigate how long it takes your FCAS to multiply random n-bit integers, for $n = 10^3, 10^4, \ldots, 10^9$. How does the performance scale as a function of n?

Remark 2.1.2. All of the computer algebra systems mentioned earlier (except possibly Minecraft) use the GMP library [20] under the bonnet[1] for their integer arithmetic needs. The $n \log n$ algorithm of [28] is probably not practical, but GMP does implement an algorithm whose theoretical complexity comes very close to $O(n \log n)$. For small n, maybe up to a few hundred bits, the running time of GMP's multiplication routine behaves like $O(n^2)$, but as n increases, the software works very hard to push the exponent down towards 1.

Division and remainder. Let u and $v > 0$ be integers with at most n bits. Suppose that we want to compute the quotient $q := \lfloor u/v \rfloor$ and corresponding remainder $r := u - qv$. The classical long division algorithm performs these tasks in time $O(n^2)$. Using a division algorithm that combines fast multiplication with Newton's method, this may be improved to

$$O(M_{\mathrm{int}}(n)) = O(n \log n).$$

GCD and extended GCD. Let $u, v > 0$ be integers with at most n bits. Let g be their greatest common divisor, and let a and b be the Bezout cofactors, i.e., so that $g = au + bv$. The classical version of Euclid's algorithm computes g, a and b in time $O(n^2)$. This may be improved to

$$O(M_{\mathrm{int}}(n) \log n) = O(n \log^2 n),$$

[1] bonnet (*noun*): metal sheet covering the engine. *U.S. English:* hood.

using any of several variants of the recursive "half-GCD" algorithm. In particular, we may test whether u is invertible modulo v, if and so, compute the modular inverse, in time $O(n \log^2 n)$.

Problem 2.1.3. (🕶) For the same values of n as in Problem 2.1.1, investigate how long it takes your FCAS to compute the quotient (and/or the corresponding remainder) of a random $2n$-bit integer by a random n-bit integer, and how long it takes to compute the GCD (and corresponding cofactors) of two random n-bit integers. How do these running times compare to the time required to multiply a pair of n-bit integers?

Modular arithmetic. Let $m \geqslant 2$. Elements of the ring $\mathbf{Z}/m\mathbf{Z}$ may be represented by residues in the interval $0 \leqslant x < m$, and so occupy $O(\log m)$ bits of space.

We can add elements of $\mathbf{Z}/m\mathbf{Z}$ by simply adding the residues and then subtracting m if necessary. The complexity is $O(\log m)$. Similar comments apply for subtraction in $\mathbf{Z}/m\mathbf{Z}$.

To multiply two elements of $\mathbf{Z}/m\mathbf{Z}$, we simply multiply the residues, divide by m, and keep the remainder. Using the results for integer multiplication and division mentioned above, the cost of this is

$$O(M_{\text{int}}(\log m)) = O(\log m \log\log m).$$

To compute a quotient in $\mathbf{Z}/m\mathbf{Z}$, we can use the extended GCD algorithm to invert the divisor (if possible) and then multiply by the dividend. The cost is

$$O(M_{\text{int}}(\log m)\log\log m) = O(\log m\,(\log\log m)^2).$$

In some algorithms we will simply count the number of "arithmetic operations" in $\mathbf{Z}/m\mathbf{Z}$, not bothering to distinguish between additions, multiplications, and divisions. In this situation we will usually simplify matters by estimating the cost of each operation to be $O(\log^{1+\varepsilon} m)$.

Remark 2.1.4. Again, the bound $O(\log^{1+\varepsilon} m)$ is not very realistic when $\log m$ is small; the quasi-linear behaviour only really kicks in when $\log m$ is quite large. Nevertheless, for the purpose of writing down mathematical theorems, it is convenient and traditional to express things this way. With some experience, one learns how to interpret these sorts of complexity bounds.

2.2. Polynomial arithmetic We will mainly be interested in the case of polynomials over $\mathbf{Z}/m\mathbf{Z}$, for an integer $m \geqslant 2$. A polynomial of degree $< n$ in $(\mathbf{Z}/m\mathbf{Z})[x]$ is represented by a sequence of n coefficients, and thus occupies space $O(n \log m)$.

Multiplication. Let $M_m(n)$ denote the cost of multiplying two polynomials $f, g \in (\mathbf{Z}/m\mathbf{Z})[x]$ of degree $< n$. One may reduce this problem to *integer* multiplication via the technique of *Kronecker substitution*, i.e., lift the polynomials to $F, G \in \mathbf{Z}[x]$, pack the coefficients of F and G into large integers (by evaluating at 2^c for a suitable integer $c \geqslant 1$), multiply the large integers, unpack the coefficients of $FG \in \mathbf{Z}[x]$ from the resulting integer product, and finally reduce modulo m to

obtain the coefficients of $fg \in (\mathbf{Z}/m\mathbf{Z})[x]$. This leads to the estimate

$$M_m(n) = O(M_{\text{int}}(n \log(nm))),$$

where the $\log(nm)$ term arises from estimating the size of the coefficients of FG.

If n is not too large compared to m, say $n \ll m^{O(1)}$, then this simplifies to

$$M_m(n) = O(M_{\text{int}}(n \log m)) = O(n \log m \log(n \log m)).$$

In other words, the cost is $O(b \log b)$ where b is the total bit size of the inputs.

Remark 2.2.1. Kronecker substitution is inefficient when m is fixed and n is large: the cost becomes $O(M_{\text{int}}(n \log n)) = O(n \log^2 n)$. Joris van der Hoeven and I recently presented an algorithm that achieves $M_m(n) = O(n \log n)$ (for fixed m) under some plausible but unproved number-theoretic hypotheses [29].

Problem 2.2.2. (🗲) Can you design an algorithm that *provably* multiplies polynomials of degree $< n$ in $(\mathbf{Z}/m\mathbf{Z})[x]$ (for fixed m) in time $O(n \log n)$?

Division. Let $f, g \in (\mathbf{Z}/m\mathbf{Z})[x]$ have degree $< n$, with g monic. Using Newton's method combined with fast multiplication, we may compute polynomials $q, r \in (\mathbf{Z}/m\mathbf{Z})[x]$ such that $f = qg + r$ and $\deg r < \deg g$ (the quotient and remainder) in time

$$O(M_m(n)).$$

2.3. Computing large powers Let R be a commutative ring, let $u \in R$ and let $k \geqslant 1$ be an integer. To compute u^k, we will generally use the "repeated squaring" method: first recursively compute $u^{\lfloor k/2 \rfloor}$, and then recover u^k as either $(u^{\lfloor k/2 \rfloor})^2$ or $(u^{\lfloor k/2 \rfloor})^2 \cdot u$, depending on whether k is even or odd.

The total number of multiplications in R is $O(\log k)$, but the complexity analysis depends on the ring. We will need the following two cases.

Proposition 2.3.1. *Let $k \geqslant 1$ and let $u \in \mathbf{Z}/m\mathbf{Z}$. Then we may compute u^k in time*

$$O(M_{\text{int}}(\log m) \log k).$$

Proposition 2.3.2. *Let $k \geqslant 1$ and let $f \in (\mathbf{Z}/m\mathbf{Z})[x]$ have degree $n \geqslant 1$. Then we may compute f^k in time*

$$O(M_m(nk)).$$

Notice that in the polynomial case, the cost of computing f^k is the same (up to a constant) as the cost of a *single* polynomial multiplication of degree $\deg(f^k) = nk$. This occurs because the cost of the polynomial multiplications increases according to a geometric progression as the algorithm proceeds.

2.4. Product trees A *product tree* on a sequence of positive integers $u_1, \ldots, u_N$ is a binary tree with N leaves, whose leaves are assigned the values $u_1, \ldots, u_N$, and whose non-leaf nodes are recursively assigned the product of the values of their children. There are various ways to handle the case that N is not a power of two,

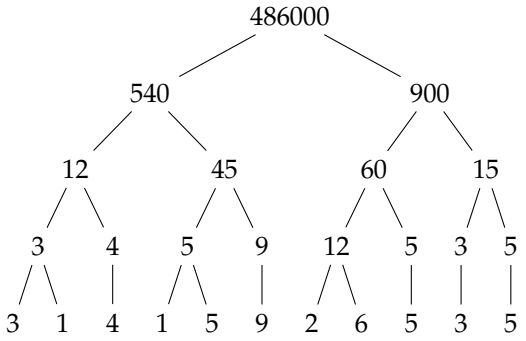

Figure 2.4.1. A product tree on the first 11 digits of π.

and any of these are reasonable as long as the height of the tree is $O(\log N)$. An example with $N = 11$ is illustrated in Figure 2.4.1.

Starting with $u_1, \dots, u_N$, we may compute all nodes in the product tree by repeatedly multiplying together values in adjacent nodes.

Proposition 2.4.2. *The cost of computing all nodes in the product tree is*

$$O(M_{\mathrm{int}}(\log B) \log N), \qquad \textit{where } B \coloneqq \prod_{j=1}^{N} u_j.$$

This follows from the fact that the total bit size of all integers in each level of the tree is $O(\log B)$, and that $M_{\mathrm{int}}(n)$ may be assumed to be super-linear, i.e., $\sum_j M_{\mathrm{int}}(n_j) \leqslant M_{\mathrm{int}}(\sum_j n_j)$.

Remark 2.4.3. Strictly speaking, one has to be careful with the complexity bounds if many of the leaf nodes have the value 1. We will ignore this minor technicality.

The same construction may also be carried out for polynomials. The following application plays an important role in §6.

Proposition 2.4.4 (Fast multipoint evaluation)**.** *Let $m, n \geqslant 2$ be integers. Suppose that we are given as input a polynomial $Q \in (\mathbf{Z}/m\mathbf{Z})[x]$ of degree at most n, and a sequence of evaluation points $\alpha_1, \dots, \alpha_n \in \mathbf{Z}/m\mathbf{Z}$. Then we may compute the values*

$$Q(\alpha_1), \dots, Q(\alpha_n) \in \mathbf{Z}/m\mathbf{Z}$$

in time

$$O(M_m(n) \log n).$$

Briefly, the algorithm implementing Proposition 2.4.4 runs as follows. We first use a polynomial analogue of Proposition 2.4.2 to compute a product tree on the linear polynomials

$$x - \alpha_1, \dots, x - \alpha_n \in (\mathbf{Z}/m\mathbf{Z})[x].$$

We then compute the reduction of $Q(x)$ modulo each node in the product tree, by working recursively from the top of the tree down to the leaves, i.e., given $Q(x)$

$(\mathrm{mod}\ g_1(x)g_2(x))$, we may compute $Q(x)\ (\mathrm{mod}\ g_1(x))$ and $Q(x)\ (\mathrm{mod}\ g_2(x))$ by simply dividing by each of g_1 and g_2. At the end we have $Q(x)\ (\mathrm{mod}\ x-\alpha_j)$, or equvialently $Q(\alpha_j)$, for all $j=1,\ldots,n$.

2.5. Linear algebra Let R be a commutative ring and let $d \geqslant 1$. Consider the problem of multiplying two $d \times d$ matrices over R. The obvious algorithm for this task requires $O(d^3)$ ring operations (multiplications and additions) in R.

One of the biggest surprises in computer algebra was Strassen's discovery [48] that this bound can be improved to $O(d^{\log 7/\log 2})$, where $\log 7/\log 2 \approx 2.81$.

I will denote by ω any feasible *exponent of matrix multiplication*, i.e., so that $d \times d$ matrices over R may be multiplied using $O(d^\omega)$ ring operations in R. Strassen's result implies that we may take $\omega = \log 7/\log 2$.

Remark 2.5.1. The best known exponent is currently $\omega = 2.3728596$ [3]. However, the result in [3] is stated for the case that R is a *field*, and counting field operations instead of ring operations. I suspect that this value of ω also works for an arbitrary commutative ring, but I am not an expert on these matters.

The actual running time of matrix multiplication depends on the cost of arithmetic in R. This is easy if R is a finite ring; for example, the cost of multiplication in $\mathrm{Mat}_d(\mathbf{F}_p)$ is $O(d^\omega\, M_{\mathrm{int}}(\log p))$. But if the matrix entries are allowed to grow, such as in the case $R = \mathbf{Z}$, then the running time also depends on the size of the entries. We will defer this issue for now and revisit it in §7.

The other result we need is an estimate for the cost of computing the *inverse characteristic polynomial* of a matrix over a field.

Proposition 2.5.2. *Given as input* $A \in \mathrm{Mat}_d(\mathbb{K})$*, we may compute the polynomial* $\det(I - TA) \in \mathbb{K}[T]$ *using* $O(d^\omega)$ *field operations in* $\mathbb{K}$.

This is actually quite a recent result; see [41].

2.6. Enumerating primes The usual sieve of Eratosthenes is tricky to implement on a multitape Turing machine due to the lack of fast random array access. Schönhage, Grotefeld and Vetter [45] showed how to list the primes $p \leqslant N$ in time $O(N \log^2 N \log\log N)$ by implementing the Eratosthenes sieve using fast sorting algorithms instead. The following slightly better bound (see [16]) is achieved by replacing the Eratosthenes sieve with the Atkin sieve.

Proposition 2.6.1. *The primes* $p \leqslant N$ *may be enumerated in time*

$$O(N \log^2 N/\log\log N).$$

3. Hyperelliptic curves and zeta functions

3.1. Hyperelliptic curves

Definition 3.1.1. Let $\mathbb{K}$ be a field of characteristic $\neq 2$ and let $g \geqslant 1$ be an integer. A *hyperelliptic curve* of genus g over $\mathbb{K}$ is the smooth algebraic curve $C/\mathbb{K}$ associated to an affine equation

$$y^2 = f(x),$$

where $f \in \mathbb{K}[x]$ is a squarefree polynomial of degree $2g+1$ or $2g+2$.

From now on, whenever we give an equation $y^2 = f(x)$ for a hyperelliptic curve of genus g, it is tacitly assumed that $f(x)$ satisfies the requirements above, i.e., it is squarefree and has degree $2g+1$ or $2g+2$.

Remark 3.1.2. Definition 3.1.1 is slightly nonstandard when $g = 1$. A curve of genus 1 is usually not called hyperelliptic, but rather *elliptic* (assuming that it has a distinguished $\mathbb{K}$-rational point). The algorithms discussed in this course work perfectly well when $g = 1$, so it will be convenient to abuse terminology and call such a curve a "hyperelliptic curve of genus 1".

Remark 3.1.3. When $\operatorname{char}\mathbb{K} = 2$, the general equation for a hyperelliptic curve has the form $y^2 + yh(x) = f(x)$. We will never need to worry about fields of characteristic two in this course.

What is the best way to think about C? A rookie mistake is to homogenise the equation $y^2 = f(x)$ in the usual way to obtain a projective curve in $\mathbf{P}^2$. The problem is that this curve is usually not smooth. (To get a smooth projective model, in general one needs to embed the curve in a higher-dimensional projective space.)

A much more useful way to think about C is as a *two-to-one cover of* $\mathbf{P}^1$. Let us describe more explicitly what this means.

Suppose that

$$f(x) = f_0 + f_1 x + \cdots + f_{2g+2} x^{2g+2}, \qquad f_i \in \mathbb{K}.$$

Note that $f_{2g+2} = 0$ is allowed; this happens exactly when $\deg f = 2g+1$.

The rational function x may be thought of as a map from C to $\mathbf{P}^1 = \mathbf{A}^1 \cup \{\infty\}$. The original equation

$$y^2 = f_0 + f_1 x + \cdots + f_{2g+2} x^{2g+2} \tag{3.1.4}$$

is really an equation for the affine patch of C consisting of those points whose x-coordinate lies in $\mathbf{A}^1$. But this patch misses all "points at infinity", i.e., the points on C whose x-coordinate is ∞. To see these points, we need to switch to a different model. Put $x = 1/u$ and $y = v/u^{g+1}$; then the equation becomes

$$v^2 = f_0 u^{2g+2} + f_1 u^{2g+1} + \cdots + f_{2g+2}. \tag{3.1.5}$$

This is an equation for a different affine patch of C, consisting of those points whose x-coordinate is not zero. In particular, the points where $x = \infty$ correspond exactly to the points on (3.1.5) where $u = 0$.

In summary, C may be written as a union of the two affine pieces given by (3.1.4) and (3.1.5), glued together by the relations $x = 1/u$, $y = v/u^{g+1}$.

Armed with this description, we can now say concretely what the points of $C(\mathbb{L})$ look like, for any field extension $\mathbb{L}$ of $\mathbb{K}$. Each point in $P \in C(\mathbb{L})$ has a well-defined x-coordinate $x(P) \in \mathbf{P}^1(\mathbb{L}) = \mathbb{L} \cup \{\infty\}$. The number of points in $C(\mathbb{L})$ with a specified x-coordinate is given by the following lemma. For this statement it is convenient to define

$$f(\infty) \coloneqq f_{2g+2}.$$

Lemma 3.1.6. *For any $\alpha \in \mathbf{P}^1(\mathbb{L}) = \mathbb{L} \cup \{\infty\}$, the number of points $P \in C(\mathbb{L})$ with $x(P) = \alpha$ is either one, two, or zero, according to whether $f(\alpha)$ is zero, a square in $\mathbb{L}^*$, or a non-square in $\mathbb{L}^*$.*

In particular:

- The points $P \in C(\mathbb{L})$ with $x(P) = 0$ correspond to solutions in $\mathbb{L}$ of
$$y^2 = f_0.$$
- The points $P \in C(\mathbb{L})$ with $x(P) = \infty$ correspond to solutions in $\mathbb{L}$ of
$$v^2 = f_{2g+2}.$$
- For any $\alpha \in \mathbb{L}^*$, the points $P \in C(\mathbb{L})$ with $x(P) = \alpha$ correspond to solutions in $\mathbb{L}$ of
$$y^2 = f(\alpha).$$
(Equivalently, these points correspond to solutions in $\mathbb{L}$ of $v^2 = \bar{f}(1/\alpha)$, where $\bar{f}(u) \coloneqq u^{2g+2} f(1/u)$ is the "reversal" of $f(x)$.)

Problem 3.1.7. Let C be the curve from Problem 1.1.1, now including the point(s) at infinity. Calculate $\#C(\mathbf{F}_3)$ and $\#C(\mathbf{F}_{3^2})$ (again by hand).

The covering map $x\colon C \to \mathbf{P}^1$ is ramified at exactly $2g+2$ points, the *Weierstrass points* of C. These are exactly the points in $C(\overline{\mathbb{K}})$ whose x-coordinates satisfy $f(\alpha) = 0$. In particular, there is a Weierstrass point at $x = \infty$ if and only if $\deg f = 2g+1$.

Finally, we make some brief comments about the squarefreeness condition in Definition 3.1.1. This condition is equivalent to requiring that $f(x)$ has no repeated roots, or alternatively that the discriminant of $f(x)$ is nonzero. (The condition "$\deg f = 2g+1$ or $2g+2$" may be interpreted as saying that $f(x)$ does not have repeated roots at ∞.) The purpose of the squarefreeness condition is to ensure that the curve is nonsingular. If it is not satisfied, then the curve defined by $y^2 = f(x)$ will have lower genus than expected. Nevertheless, many of the results in this course work perfectly well without this condition.

3.2. The number of points on the curve Now suppose that $\mathbb{K}$ is a *finite* field, say $\mathbb{K} = \mathbf{F}_q$ where $q = p^a$, for p an odd prime and $a \geqslant 1$. Let $C/\mathbf{F}_q$ be a hyperelliptic curve of genus $g \geqslant 1$. In this case $C(\mathbf{F}_q)$ is obviously a *finite* set. In fact, Lemma 3.1.6 implies that $\#C(\mathbf{F}_q) \leqslant 2(q+1)$, since there are at most two points in $C(\mathbf{F}_q)$ for each of the $q+1$ possible values of x in $\mathbf{P}^1(\mathbf{F}_q)$. But we can do much better:

Theorem 3.2.1 (Hasse–Weil bound)**.** *Let $C/\mathbf{F}_q$ be a hyperelliptic curve of genus $g \geqslant 1$. Then*

$$\bigl|\#C(\mathbf{F}_q) - (q+1)\bigr| \leqslant 2g \cdot q^{1/2}.$$

A heuristic argument for Theorem 3.2.1 might go like this: for each $\alpha \in \mathbf{P}^1(\mathbf{F}_q)$, the value of $f(\alpha)$ is "equally likely" to be a square or a non-square in $\mathbf{F}_q$, so the total number of points should behave like a random variable with mean $q+1$ and variance proportional to q. This is probably the wrong way to think about it, but it does give roughly the right answer!

Theorem 3.2.1 also implies that

$$\bigl|\#C(\mathbf{F}_{q^r}) - (q^r+1)\bigr| \leqslant 2g \cdot q^{r/2} \tag{3.2.2}$$

for any $r \geqslant 1$. This follows by interpreting the equation $y^2 = f(x)$ defining C as an equation over $\mathbf{F}_{q^r}$ instead of over $\mathbf{F}_q$.

We will not give a proof of Theorem 3.2.1 in this course. The result actually holds for any curve of genus g, not just for hyperelliptic curves. The theorem follows from a more general statement about the *zeta function* of the curve, discussed in the next section; see in particular Problem 3.3.8.

The main thrust of this course is to develop efficient algorithms for calculating the numbers $\#C(\mathbf{F}_{q^r})$ for $r = 1, 2, \ldots$, given as input the polynomial $f \in \mathbf{F}_q[x]$. The simplest possible algorithm for computing $\#C(\mathbf{F}_{q^r})$ is straightforward *enumeration*: loop through all $\alpha \in \mathbf{P}^1(\mathbf{F}_{q^r}) = \mathbf{F}_{q^r} \cup \{\infty\}$, and for each α compute $f(\alpha)$ and test whether it is a square in $\mathbf{F}_{q^r}$ (see Lemma 3.1.6). We will not analyse the complexity of this algorithm in detail. The main point is that the complexity grows at least as rapidly as q^r, since this is the number of x-values that must be inspected. In particular, the running time is *exponential in* r.

The enumeration algorithm can actually be a reasonable choice when the parameters are small. Various implementation tricks are available, such as using difference tables to speed up the computation of the values of $f(\alpha)$. For details see [33, §3] and Sutherland's `smalljac` library [50].

Problem 3.2.3. (🥽) Let $p = 101$ and let $C/\mathbf{F}_p$ be the genus 3 curve given by

$$y^2 = x^7 - x^6 + 6x^5 - 7x^4 + 5x^3 + x^2 - x + 1.$$

Implement the enumeration algorithm in your FCAS and use it to show that

$$\#C(\mathbf{F}_p) = 153, \qquad \#C(\mathbf{F}_{p^2}) = 9935, \qquad \#C(\mathbf{F}_{p^3}) = 1029891.$$

Check that (3.2.2) holds for these values.

Problem 3.2.4. (🥽) Investigate whether your FCAS has built-in functionality for computing $\#C(\mathbf{F}_{p^r})$. Do you know what algorithm it is using under the bonnet? Experimentally, how does its running time vary as a function of p, g and r?

3.3. The zeta function Let $C/\mathbf{F}_q$ be a hyperelliptic curve of genus $g \geqslant 1$. The sequence of point counts $\#C(\mathbf{F}_{q^r})$, for $r = 1, 2, \ldots$, may be encoded into a generating

function (formal power series) called the the *zeta function* of C. This is defined by the formula

$$Z_C(T) \coloneqq \exp\left(\sum_{r=1}^{\infty} \frac{\#C(\mathbf{F}_{q^r})}{r} T^r\right) \in \mathbf{Q}[\![T]\!]. \tag{3.3.1}$$

Here exp denotes the usual exponential of power series, i.e.,

$$\exp U(T) \coloneqq 1 + U(T) + \frac{U(T)^2}{2!} + \cdots.$$

Theorem 3.3.2 (Weil Conjectures)**.** *Let $C/\mathbf{F}_q$ be a hyperelliptic curve of genus $g \geqslant 1$. Then the zeta function of C is a rational function of the form*

$$Z_C(T) = \frac{L_C(T)}{(1-T)(1-qT)},$$

where $L_C(T) \in 1 + T\mathbf{Z}[T]$ is a polynomial of degree $2g$ with the following properties:

- *(Functional equation) We have*
$$L_C\left(\frac{1}{qT}\right) = (qT^2)^{-g} L_C(T).$$
- *(Riemann Hypothesis) The roots of $L_C(T)$ (in the field of complex numbers) have absolute value $q^{-1/2}$.*

Despite traditionally being known as the "Weil Conjectures", the statements in Theorem 3.3.2 are most definitely theorems! In fact, the theorem holds verbatim for an arbitrary nonsingular curve of genus g over $\mathbf{F}_q$. For a proof, see for instance [37] or [47]. (The Weil Conjectures may also be generalised far beyond the case of curves; this provided the motivating force for much of the development of 20th century algebraic geometry.)

One important implication of Theorem 3.3.2 is that knowledge of the finite object $L_C(T)$ is equivalent to knowledge of the entire infinite sequence $\#C(\mathbf{F}_{q^r})$ for $r \geqslant 1$. So another way to express the goal of this course is that we want to develop efficient algorithms for computing $L_C(T)$, given the equation $y^2 = f(x)$ as input.

Remark 3.3.3. There is no known algorithm for computing $L_C(T)$ in polynomial time, i.e., polynomial in the amount of data needed to represent the input polynomial $f \in \mathbf{F}_q[x]$, which is $O(g \log q) = O(ag \log p)$ bits. When g is fixed, there are ℓ-adic algorithms that can compute $L_C(T)$ in time polynomial in $\log q$, but the exponent grows with g. On the other hand, when p is fixed, there are p-adic algorithms that can compute $L_C(T)$ in time polynomial in g and a. (The algorithms suggested in §8.5 also have this property.)

The following problem works out more explicitly the relationship between the point counts and the coefficients of $L_C(T)$ and $Z_C(T)$.

Problem 3.3.4. Let $N_r \coloneqq \#C(\mathbf{F}_{q^r})$, and put

$$Z_C(T) = 1 + \sum_{i \geqslant 1} c_i T^i, \qquad L_C(T) = 1 + \sum_{i \geqslant 1} a_i T^i,$$

where we tacitly write $a_i = 0$ for $i > 2g$.

a) Find formulas for c_1 and c_2 in terms of N_1 and N_2. Conversely, find formulas for N_1 and N_2 in terms of c_1 and c_2.
b) Same as (a) for (a_1, a_2) and (c_1, c_2).
c) More generally, show that for any $t \geqslant 1$, there exist formulas expressing the vectors $(N_1, \ldots, N_t)$, $(a_1, \ldots, a_t)$ and $(c_1, \ldots, c_t)$ in terms of each other.

Problem 3.3.4(c) implies that $L_C(T)$ is determined by the values $(N_1, \ldots, N_{2g})$. The next problem shows that we can do a little better by taking advantage of the functional equation.

Problem 3.3.5. Let

$$L_C(T) = a_0 + a_1 T + \cdots + a_{2g} T^{2g},$$

where $a_0 = 1$. Show that the functional equation in Theorem 3.3.2 is equivalent to the statement that

$$a_{2g-i} = q^{g-i} a_i, \qquad 0 \leqslant i \leqslant 2g. \tag{3.3.6}$$

Conclude that knowledge of $N_1, \ldots, N_g$ (where $N_r \coloneqq \#C(\mathbf{F}_{q^r})$) is enough to deduce $L_C(T)$.

Problem 3.3.5 implies that if we want to use the enumeration algorithm to compute $L_C(T)$, then we should expect the complexity to grow at least as rapidly as q^g, since this is how much work we need to do to compute $\#C(\mathbf{F}_{q^g})$. In particular, the complexity is *exponential in* g.

Problem 3.3.7. (🎮) Consider again the genus 3 curve from Problem 3.2.3.

a) Using the point counts found in Problem 3.2.3, compute the coefficients of T, T^2 and T^3 in $L_C(T)$.
b) Use (3.3.6) to recover the full polynomial $L_C(T)$.
c) Use $L_C(T)$ to deduce the value of $\#C(\mathbf{F}_{p^4})$.
d) Compute $\#C(\mathbf{F}_{p^4})$ directly using the enumeration algorithm, i.e., the same method as in Problem 3.2.3, and compare the results. (There are about 100 million points to count here. You will need a fairly tight implementation to finish in a reasonable amount of time.)

Problem 3.3.8. Prove Theorem 3.2.1 (the Hasse–Weil bound) from Theorem 3.3.2 (the Weil Conjectures) as follows.

a) Use the Riemann Hypothesis in Theorem 3.3.2 to show that $L_C(T)$ admits a factorisation

$$L_C(T) = \prod_{i=1}^{2g} (1 - \omega_i T)$$

where $\omega_1, \ldots, \omega_{2g}$ are complex numbers with $|\omega_i| = q^{1/2}$.

b) By taking the formal logarithm of $Z_C(T)$, show that

$$\#C(\mathbf{F}_{q^r}) = q^r + 1 - \sum_{i=1}^{2g} \omega_i^r$$

for all $r \geqslant 1$. Hence deduce Theorem 3.2.1.

4. A modulo p trace formula

To keep things simple, for the rest of the course we will assume that $q = p$, i.e., we will only consider curves defined over the prime field $\mathbf{F}_p$. Recall also that p is always assumed to be odd.

The main result of this section is Theorem 4.2.7, which gives a congruence modulo p for the number of points on a hyperelliptic curve $C/\mathbf{F}_p$ with coordinates in $\mathbf{F}_{p^r}$, for any $r \geqslant 1$. This "trace formula" will play a central role in the point counting algorithms discussed later in the course.

4.1. Counting points with coordinates in $\mathbf{F}_p$ We begin with a baby version that only counts points defined over the prime field.

Proposition 4.1.1. *Let $C/\mathbf{F}_p$ be a hyperelliptic curve of genus $g \geqslant 1$ defined by $y^2 = f(x)$ for some $f \in \mathbf{F}_p[x]$. Then*

$$\#C(\mathbf{F}_p) \equiv 1 - \sum_{j=1}^{g} h_{j(p-1)} \pmod{p}, \tag{4.1.2}$$

where

$$h \coloneqq f^{(p-1)/2} \in \mathbf{F}_p[x].$$

Recall that $h_{j(p-1)}$ means the coefficient of $x^{j(p-1)}$ in $h(x)$. For the proof of Proposition 4.1.1, it is helpful to introduce a certain subvariety of C.

Definition 4.1.3. Let $C/\mathbf{F}_p$ be a hyperelliptic curve given by $y^2 = f(x)$. We denote by $\widetilde{C}$ the subvariety of C consisting of those points for which $x \neq 0, \infty$.

Problem 4.1.4. Prove Proposition 4.1.1 as follows.

a) Using (1.5.1) and Lemma 3.1.6, show that

$$\#\widetilde{C}(\mathbf{F}_p) \equiv \sum_{\alpha \in \mathbf{F}_p^*} \left(f(\alpha)^{(p-1)/2} + 1\right) \pmod{p}.$$

b) For any integer $i \geqslant 0$, show that

$$\sum_{\alpha \in \mathbf{F}_p^*} \alpha^i = \begin{cases} -1 & \text{if } i \text{ is divisible by } p-1, \\ 0 & \text{otherwise.} \end{cases}$$

c) Combining (a) and (b), show that

$$\#\widetilde{C}(\mathbf{F}_p) \equiv -1 - \sum_{j=0}^{g+1} h_{j(p-1)} \pmod{p}.$$

d) Show that

$$\#\{P \in C(\mathbf{F}_p) : x(P) = 0\} \equiv h_0 + 1 \pmod{p},$$
$$\#\{P \in C(\mathbf{F}_p) : x(P) = \infty\} \equiv h_{(g+1)(p-1)} + 1 \pmod{p}.$$

e) Deduce (4.1.2) by adding together the results of (c) and (d).

Problem 4.1.5. (🕹) Compute $\#C(\mathbf{F}_p) \pmod{p}$ for the curve in Problem 3.2.3, by using your FCAS to explicitly compute $h = f^{(p-1)/2}$, and then applying Proposition 4.1.1. Notice that h is a polynomial of degree 350 over $\mathbf{F}_p$, but only three of its coefficients contribute in (4.1.2). Check that your result is consistent with the value for $\#C(\mathbf{F}_p)$ found in Problem 3.2.3.

Problem 4.1.6.

a) Use the Hasse–Weil bound (Theorem 3.2.1) to show that if $p > 16g^2$, then knowledge of $\#C(\mathbf{F}_p) \pmod{p}$ determines $\#C(\mathbf{F}_p) \in \mathbf{Z}$ unambiguously.

b) For the curve in Problem 3.2.3, show that $\#C(\mathbf{F}_p) \pmod{p}$ does not provide enough information to pin down $\#C(\mathbf{F}_p)$. What possible values for $\#C(\mathbf{F}_p)$ are compatible with the Hasse–Weil bound for this curve?

4.2. Counting points with coordinates in $\mathbf{F}_{p^r}$ We now investigate how to generalise Proposition 4.1.1 to obtain a formula for $\#C(\mathbf{F}_{p^r}) \pmod{p}$ for any $r \geqslant 1$. The following problem carries out the most obvious line of attack, which is to replace the sum over $\mathbf{F}_p^*$ by a sum over $\mathbf{F}_{p^r}^*$.

Problem 4.2.1. Let $C/\mathbf{F}_p$ be a hyperelliptic curve of genus $g \geqslant 1$ defined by $y^2 = f(x)$ for some $f \in \mathbf{F}_p[x]$. By mimicking the proof of Proposition 4.1.1 (see Problem 4.1.4), show that for any $r \geqslant 1$,

$$\#C(\mathbf{F}_{p^r}) \equiv 1 - \sum_{j=1}^{g} h^{(r)}_{j(p^r-1)} \pmod{p}, \tag{4.2.2}$$

where

$$h^{(r)} \coloneqq f^{(p^r-1)/2} \in \mathbf{F}_p[x].$$

Unfortunately, (4.2.2) is not much use on its own because the degree of the polynomial $h^{(r)} = f^{(p^r-1)/2}$ grows exponentially in r. Except for perhaps the smallest values of r, we cannot expect to be able to explicitly compute such a large power of f. Fortunately, there is a trick that enables us to extract all the information we need from the original polynomial $h = f^{(p-1)/2}$.

Definition 4.2.3. Let $C/\mathbf{F}_p$ be a hyperelliptic curve of genus $g \geqslant 1$ defined by $y^2 = f(x)$ for some $f \in \mathbf{F}_p[x]$. Define the matrix

$$A_f \in \mathrm{Mat}_g(\mathbf{F}_p)$$

by the formula

$$(A_f)_{v,u} \coloneqq h_{vp-u}, \qquad 1 \leqslant u, v \leqslant g, \tag{4.2.4}$$

where

$$h \coloneqq f^{(p-1)/2}.$$

The notation A_f emphasises that the matrix depends not only on the curve C, but also on the choice of equation $y^2 = f(x)$.

Figure 4.2.5. Illustration of the coefficients of $h = f^{(p-1)/2}$ for $p = 11$, $g = 3$, $\deg f = 2g+2 = 8$. The circles indicate the coefficients of $1, x, \dots, x^{40}$. Coefficients contributing to A_f are marked in black.

Remark 4.2.6. The matrix A_f is usually known as a *Cartier–Manin matrix* associated to C; this is the matrix of the Cartier operator on the space of regular differentials on C with respect to a particular basis. It is also closely related to the *Hasse–Witt matrix*, which is the matrix of the Frobenius operator on a certain cohomology group with respect to a particular basis. The precise definitions of these objects are beyond the scope of this course. You are warned that there are some inconsistencies in the literature regarding the definition of these matrices; for a detailed discussion of this issue see [2].

Theorem 4.2.7 (Modulo p trace formula)**.** *Let $C/\mathbf{F}_p$ be a hyperelliptic curve of genus $g \geqslant 1$ defined by $y^2 = f(x)$ for some $f \in \mathbf{F}_p[x]$, and let A_f be the associated matrix as in Definition 4.2.3. Then for any $r \geqslant 1$,*

$$\#C(\mathbf{F}_{p^r}) \equiv 1 - \operatorname{tr}(A_f^r) \pmod{p}.$$

Theorem 4.2.7 shows that to compute $\#C(\mathbf{F}_{p^r}) \pmod{p}$, it is enough to compute g^2 selected coefficients of $h = f^{(p-1)/2}$. (Proposition 4.1.1 only used a subset of g of these coefficients.) Notice that although the formula counts points in $\mathbf{F}_{p^r}$, all quantities involved in the formula, i.e., the entries of A_f, lie in $\mathbf{F}_p$.

Problem 4.2.8. Check that Theorem 4.2.7 specialises to Proposition 4.1.1 in the case $r = 1$.

Problem 4.2.9. Prove Theorem 4.2.7 as follows.

a) Show that for any $r \geqslant 1$,

$$f(x)^{(p^r-1)/2} = h(x)h(x^p)\cdots h(x^{p^{r-1}}),$$

where $h = f^{(p-1)/2}$ as in Definition 4.2.3.

b) Let $\widetilde{A}_f$ be the $(g+2) \times (g+2)$ matrix over $\mathbf{F}_p$ defined by

$$(\widetilde{A}_f)_{\nu,u} \coloneqq h_{\nu p - u}, \qquad 0 \leqslant u, \nu \leqslant g+1.$$

Show by induction on $r \geqslant 1$ that

$$(\widetilde{A}_f^r)_{\nu,u} = (h(x)h(x^p)\cdots h(x^{p^{r-1}}))_{\nu p^r - u}, \qquad 0 \leqslant u, \nu \leqslant g+1.$$

(Hint: find an expression for the coefficient of x^{vp^r-u} in $h(x)\cdots h(x^{p^{r-1}})$ in terms of the coefficients of $h(x)\cdots h(x^{p^{r-2}})$ and $h(x)$.)

c) Show that all entries in the first and last rows of $\widetilde{A}_f$ are zero, except possibly for $(\widetilde{A}_f)_{0,0}$ and $(\widetilde{A}_f)_{g+1,g+1}$. Deduce that for all $r \geqslant 1$,

$$(A_f^r)_{v,u} = (\widetilde{A}_f^r)_{v,u}, \qquad 1 \leqslant u, v \leqslant g.$$

d) By combining (a), (b), (c) and Problem 4.2.1, complete the proof of Theorem 4.2.7.

Problem 4.2.10. (💻) For the curve in Problem 3.2.3, use your FCAS to compute $\#C(\mathbf{F}_{p^r}) \pmod p$ for $r = 1, 2, 3$, by computing $h = f^{(p-1)/2}$ (the same degree 350 polynomial that appeared in Problem 4.1.5), constructing A_f, and applying Theorem 4.2.7. Check that your answer is consistent with the point counts given in Problem 3.2.3. What happens if you try to explicitly compute the polynomials $h^{(r)}$ from Problem 4.2.1 for $r = 1, 2, 3$?

4.3. Recovering the L-polynomial modulo p What does the trace formula (Theorem 4.2.7) tell us about the L-polynomial

$$L_C(T) = a_0 + a_1 T + \cdots + a_{2g} T^{2g}?$$

We saw in Problem 3.3.4(c) that for any $t \geqslant 1$ there exist formulas allowing us to move back and forth between $(a_1, \ldots, a_t)$ and $(N_1, \ldots, N_t)$, where $N_r := \#C(\mathbf{F}_{p^r})$. Moreover, it follows from the functional equation that $a_{g+1}, \ldots, a_{2g}$ are all zero modulo p (see Problem 3.3.5). Since the trace formula tells us how to compute $N_1, \ldots, N_g \pmod p$, it is natural to wonder whether the latter quantities determine $a_1, \ldots, a_g \pmod p$, and hence $L_C(T) \pmod p$. The answer in general is *no*, as illustrated by the following counterexample.

Problem 4.3.1. (💻) Let $p = 3$ and consider the genus 3 curves over $\mathbf{F}_p$ given by

$$C\colon\ y^2 = x^7 + 1, \qquad C'\colon\ y^2 = x^7 - x^5 + 1.$$

Use your FCAS to show that $\#C(\mathbf{F}_{p^r}) \equiv \#C'(\mathbf{F}_{p^r}) \pmod p$ for $r = 1, 2, 3$, but that $L_C(T) \not\equiv L_{C'}(T) \pmod p$.

The underlying issue is that the formulas relating $(a_1, \ldots, a_g)$ and $(N_1, \ldots, N_g)$ involve *denominators* that may be divisible by p. Some of these denominators are already clearly visible in the defining formula (3.3.1) for the zeta function.

Despite this, it turns out that there is a very elegant identity expressing $L_C(T) \pmod p$ directly in terms of A_f:

Theorem 4.3.2. *We have*

$$L_C(T) \equiv \det(I - TA_f) \pmod p.$$

Theorem 4.3.2 follows quite easily from the trace formula when $p > g$, essentially because the none of the relevant denominators are divisible by p. In this case knowledge of $a_1, \ldots, a_g \pmod p$ is equivalent to knowledge of $N_1, \ldots, N_g$

$\pmod p$, and Theorem 4.3.2 is really just a restatement of the trace formula. The proof of Theorem 4.3.2 for this case is outlined in Problem 4.3.3 below.

However, when $p \leqslant g$, Theorem 4.3.2 is genuinely stronger than the trace formula; the counterexample in Problem 4.3.1 shows that A_f and $L_C(T) \pmod p$ contain potentially more information than the point counts $N_1, \dots, N_g \pmod p$. To the best of my knowledge, the proof of Theorem 4.3.2 for $p \leqslant g$ requires cohomological methods (see [38]), and we will not discuss it in this course.

Problem 4.3.3. This problem gives a proof of Theorem 4.3.2 in the $p > g$ case.

a) Assume temporarily that we are working over a field $\mathbb{K}$ of characteristic zero. Let $A \in \mathrm{Mat}_g(\mathbb{K})$. Show that

$$\det(I - TA) = \exp(\mathrm{tr}(\log(I - TA))). \tag{4.3.4}$$

In this formula, $\log(I - TA)$ should be interpreted as a formal power series with matrix coefficients, i.e.,

$$\log(I - TA) = -AT - \frac{1}{2}A^2T^2 - \cdots,$$

and the trace is taken term by term.

b) Now suppose that $\operatorname{char} \mathbb{K} = p$. In this case the right hand side of (4.3.4) is not even defined, since we cannot divide by p. Nevertheless, show that (4.3.4) is still correct when regarded modulo T^p, i.e., show that the coefficients of $1, T, \dots, T^{p-1}$ on the right hand side are well-defined, and that they agree with those on the left hand side.

c) By combining (3.3.1), Theorem 4.2.7 and part (b), show that the coefficients of $1, T, \dots, T^{p-1}$ in $(1 - T)Z_C(T)$ and $\det(I - TA_f)$ agree modulo p.

d) Use (c) and Theorem 3.3.2 to prove Theorem 4.3.2 in the case that $p > g$.

Problem 4.3.5. (🗲) Is it possible to prove Theorem 4.3.2 by "elementary" (non-cohomological) means, using methods similar to the proof of Theorem 4.2.7?

Problem 4.3.6. (🥽) Use your FCAS to verify Theorem 4.3.2 numerically for the curve in Problem 3.2.3, i.e., compute $\det(I - TA_f)$ and check that the result is consistent with the L-polynomial computed in Problem 3.3.7(b).

4.4. Complexity bounds How quickly can we compute A_f from f? The most obvious approach is to expand out the power $h = f^{(p-1)/2}$ using fast polynomial arithmetic, and then extract the relevant coefficients. The following theorem indicates the complexity of this procedure.

Theorem 4.4.1 (Computing A_f via naive expansion)**.** *Let $C/\mathbf{F}_p$ be a hyperelliptic curve of genus $g \geqslant 1$ defined by $y^2 = f(x)$ for some $f \in \mathbf{F}_p[x]$. Then we may compute A_f in time*

$$O\big(gp \log^2(gp) + g^2 \log p\big).$$

Problem 4.4.2. Prove Theorem 4.4.1, by showing that the cost of computing $h = f^{(p-1)/2}$ is $O(gp \log^2(gp))$ (see Proposition 2.3.2).

(Where does the $O(g^2 \log p)$ term come from? Hint: when $g > p$, some coefficients of h appear multiple times in A_f.)

Having computed A_f, we may use Theorem 4.3.2 to deduce $L_C(T) \pmod p$. Thanks to Proposition 2.5.2, the complexity of this step is given as follows.

Corollary 4.4.3. *Given A_f, we may compute $L_C(T) \pmod p$ in time*

$$O(g^\omega \log^{1+\varepsilon} p).$$

Combining this corollary with Theorem 4.4.1, we obtain our first point counting algorithm that is genuinely more powerful than the enumeration method: namely, given $f \in \mathbf{F}_p[x]$ as input, we may compute $L_C(T) \pmod p$ in time

$$O\big(gp \log^2(gp) + g^\omega \log^{1+\varepsilon} p\big). \tag{4.4.4}$$

We stress that this complexity bound is *polynomial* in g. This should be contrasted with the *exponential* dependence on g in the case of the enumeration algorithm, which must inspect at least p^g values of x.

On the other hand, whereas the enumeration algorithm computes the whole polynomial $L_C(T)$, the new algorithm only recovers $L_C(T) \pmod p$. This shortcoming will be addressed later in §8. In the meantime, in §§5–7 we will focus on improving the complexity with respect to p, which is slightly worse than linear in (4.4.4).

5. Recurrences for polynomial powers

Fix some $f \in \mathbf{F}_p[x]$ defining a hyperelliptic curve $C/\mathbf{F}_p$ of genus $g \geqslant 1$. In this section we establish a *recurrence* for the coefficients of $h(x) = f(x)^{(p-1)/2}$. We then attempt to use this recurrence to compute the coefficients h_{vp-u} appearing in the A_f matrix (see Definition 4.2.3). Unfortunately, this fails due to occasional divisions by p. In §5.2 we show how to resolve this difficulty by lifting the whole computation to $\mathbf{Z}/p^\mu\mathbf{Z}$ for a suitable "precision" parameter $\mu \geqslant 1$.

Throughout this section we write

$$m \coloneqq \frac{p-1}{2}, \qquad d \coloneqq 2g+2.$$

To simplify the discussion, we assume throughout that

$$f_0 \neq 0.$$

See Remark 5.1.7 for some hints on what to do if $f_0 = 0$.

5.1. A recurrence over $\mathbf{F}_p$

Proposition 5.1.1. *Let $f \in \mathbf{F}_p[x]$ and $h = f^m$ be as above. Then for any integer $k \geqslant 1$ not divisible by p, we have*

$$h_k = \frac{1}{kf_0} \sum_{j=1}^{d} (\tfrac{1}{2}j - k) f_j h_{k-j}. \tag{5.1.2}$$

In other words, (5.1.2) expresses each h_k as a linear combination of the previous d coefficients $h_{k-1}, \dots, h_{k-d}$, provided that we can divide by kf_0 in $\mathbf{F}_p$. (Recall that by convention $h_i = 0$ for $i < 0$.)

Problem 5.1.3. Prove Proposition 5.1.1 as follows. Let ∂ denote the differential operator $s \mapsto x\frac{ds}{dx}$, i.e.,

$$\partial(s_n x^n + \cdots + s_2 x^2 + s_1 x + s_0) := n s_n x^n + \cdots + 2 s_2 x^2 + s_1 x.$$

Show that $h = f^m$ satisfies the differential equation

$$f \cdot \partial h = m \cdot \partial f \cdot h. \tag{5.1.4}$$

By equating coefficients of x^k in this identity, deduce (5.1.2).

What happens if we try to use (5.1.2) to compute the coefficients h_{vp-u} appearing in A_f? To get started, we need to know the constant term h_0. It is easy to compute $h_0 = (f_0)^m \in \mathbf{F}_p$ using a fast powering algorithm (see Proposition 2.3.1); this only requires time $O(\log m \cdot \log^{1+\varepsilon} p) = O(\log^{2+\varepsilon} p)$.

We now apply the recurrence to compute successively $h_1, h_2, \dots, h_{p-1}$. The last g such coefficients yield the first row of A_f. But after this point we hit a wall: we cannot compute h_p because that would involve dividing by zero in $\mathbf{F}_p$. Another way of saying this is that the differential equation (5.1.4) simply does not contain enough information to recover h_p from knowledge of $h_0, \dots, h_{p-1}$. Unfortunately, this is not good enough for our purposes; for the second row of A_f we need to get to h_{2p-1}, and for the remaining rows we need to get all the way up to h_{gp-1}.

Problem 5.1.5. (🥽) Verify (5.1.2) numerically in your FCAS, i.e., take a small g, moderate p, and random $f \in \mathbf{F}_p[x]$, and use the recurrence to compute successively $h_0, \dots, h_{p-1} \in \mathbf{F}_p$. Check that the coefficients produced in this way agree with a direct computation of $h = f^m$.

Problem 5.1.6. Derive a recurrence for $h = f^m$ going in the opposite direction, i.e., expressing h_k in terms of $h_{k+1}, \dots, h_{k+d}$. What conditions on k and the coefficients of $f(x)$ must be satisfied for your formula to be valid?

Remark 5.1.7. At the outset we imposed the hypothesis $f_0 \neq 0$. To handle an input polynomial with $f_0 = 0$, there are a few possible ways to proceed.

One reasonable solution is to replace $f(x)$ by $f(x)/x$. Notice that the constant term of $f(x)/x$ is nonzero because we assumed f to be squarefree. We can easily write down a similar recurrence for $(f(x)/x)^m$, and the problem of computing selected coefficients of $f(x)^m$ is really the same as computing selected coefficients of $(f(x)/x)^m$. If anything, the latter problem is slightly easier because the degree is smaller.

Another option is to choose some $c \in \mathbf{F}_p$ such that $f(c) \neq 0$ (such c exists provided that $p > 2g + 2$), and replace $f(x)$ by $f(x + c)$. Since the equations

$y^2 = f(x)$ and $y^2 = f(x+c)$ define isomorphic curves, the modified equation will lead to the same zeta function.

5.2. Lifting the coefficient ring Following a suggestion of Bostan, Gaudry and Schost [7], our strategy for circumventing the "division by p" problem will be to *lift* the entire computation from $\mathbf{F}_p = \mathbf{Z}/p\mathbf{Z}$ to the ring $\mathbf{Z}/p^\mu\mathbf{Z}$ for some parameter $\mu \geqslant 1$ (to be chosen later).

We begin by choosing a modulo p^μ lift of f, i.e., a polynomial

$$F = F_d x^d + \cdots + F_1 x + F_0 \in (\mathbf{Z}/p^\mu\mathbf{Z})[x]$$

such that $F \equiv f \pmod{p}$. For example, if the coefficients of f are represented by integers from the set $\{0, 1, \ldots, p-1\}$, we can take F to be the polynomial whose coefficients are literally the same integers, but now regarded as elements of $\mathbf{Z}/p^\mu\mathbf{Z}$.

Now define

$$H \coloneqq F^m \in (\mathbf{Z}/p^\mu\mathbf{Z})[x].$$

The reduction of H modulo p is simply h, so to compute the desired coefficients of h, it suffices to compute the corresponding coefficients of H modulo p.

The coefficients of H satisfy a recurrence analogous to the one satisfied by the coefficients of h:

Proposition 5.2.1. *Let $F \in (\mathbf{Z}/p^\mu\mathbf{Z})[x]$ and $H = F^m$ be as above. Then for any integer $k \geqslant 1$, we have*

$$kH_k = \frac{1}{F_0} \sum_{j=1}^{d} \big((m+1)j - k\big) F_j H_{k-j}. \tag{5.2.2}$$

Keep in mind that (5.2.2) is really a congruence modulo p^μ. Also, note that our running assumption $f_0 \neq 0$ implies that $F_0 \not\equiv 0 \pmod{p}$, so F_0 is a unit in $\mathbf{Z}/p^\mu\mathbf{Z}$.

Problem 5.2.3. Prove Proposition 5.2.1, by first showing that $H = F^m$ satisfies the differential equation

$$F \cdot \partial H = m \cdot \partial F \cdot H. \tag{5.2.4}$$

What happens if we try to use (5.2.2) to compute the coefficients of H? As before, we can compute the first term $H_0 = (F_0)^m \in \mathbf{Z}/p^\mu\mathbf{Z}$ efficiently using a fast powering algorithm. Then we try to solve (5.2.2) for $H_1, H_2, \ldots$ in turn. Whenever we encounter an index k divisible by p, the division by k introduces some *error* into our computed value for H_k. For example, our computed value for H_p will be correct modulo $p^{\mu-1}$, but not necessarily modulo p^μ. Moreover, these errors might propagate into subsequent values of H_k. It is even conceivable that the errors might become so large that (5.2.2) fails to be solvable for some large k.

The following problem asks you to carry out a "naive" analysis of the error propagation. We write $v_p(k)$ for the p-adic valuation of k, i.e., the largest integer $t \geqslant 0$ such that $p^t \mid k$.

Problem 5.2.5. Let $n \geqslant 1$. Show that after solving for $H_1, \dots, H_n$, the total number of powers of p that we have divided by is $v_p(n!)$. Deduce that if we choose

$$\mu \coloneqq v_p(n!) + 1,$$

then our computed values of $H_0, \dots, H_n \in \mathbf{Z}/p^\mu\mathbf{Z}$ will agree modulo p with the target values $h_0, \dots, h_n \in \mathbf{F}_p$.

Problem 5.2.5 suggests that the "precision loss" in our proposed algorithm is at least *linear* in n. For example, to compute A_f we need to get up to h_{gp-1}, so we need to take

$$\mu = v_p((gp-1)!) + 1 \geqslant g.$$

Rather surprisingly, it turns out that the errors that occur in reality are much smaller than those predicted by the above discussion. Indeed, we will next show that the precision loss is only *logarithmic* in n. (I am indebted to Jan Tuitman for explaining the following argument to me. The underlying principle is similar to [31, Lemma 2].)

First we prove that the congruence (5.2.2) is *always* solvable for H_k, no matter how much error has accumulated, as long as all the previously computed values satisfy the recurrence.

Proposition 5.2.6. *Let $n \geqslant 1$, and suppose that we are given $\widetilde{H}_0, \dots, \widetilde{H}_{n-1} \in \mathbf{Z}/p^\mu\mathbf{Z}$ such that $\widetilde{H}_0 = H_0$ and such that*

$$k\widetilde{H}_k = \frac{1}{F_0} \sum_{j=1}^{d} \big((m+1)j - k\big) F_j \widetilde{H}_{k-j} \tag{5.2.7}$$

for all $k = 1, \dots, n-1$. Then for $k = n$, the congruence (5.2.7) admits at least one solution $\widetilde{H}_n \in \mathbf{Z}/p^\mu\mathbf{Z}$.

Problem 5.2.8. Prove Proposition 5.2.6 as follows.

a) Let

$$\widetilde{H} \coloneqq \widetilde{H}_0 + \widetilde{H}_1 x + \cdots + \widetilde{H}_{n-1} x^{n-1} \in (\mathbf{Z}/p^\mu\mathbf{Z})[x].$$

Show that

$$F \cdot \partial\widetilde{H} \equiv m \cdot \partial F \cdot \widetilde{H} \pmod{x^n},$$

i.e., $\widetilde{H}$ satisfies the same differential equation as H, modulo x^n.

b) Let

$$E = 1 + E_1 x + \cdots + E_{n-1} x^{n-1} \in (\mathbf{Z}/p^\mu\mathbf{Z})[x]$$

be such that

$$E \equiv \widetilde{H}/H \pmod{x^n}.$$

(In other words, E consists of the first n coefficients of the power series expansion of $\widetilde{H}/H$. Division by H is permissible, since our running assumption $f_0 \neq 0$ implies that $H_0 = (F_0)^m$ is a unit in $\mathbf{Z}/p^\mu\mathbf{Z}$.)

Show that

$$\partial E = 0.$$

c) Let

$$\widetilde{H}' \coloneqq E \cdot H \pmod{x^{n+1}}.$$

Show that $\widetilde{H}'$ satisfies the differential equation modulo x^{n+1}. Deduce that $\widetilde{H}'_n$ is a solution to (5.2.7) for $k = n$.

Proposition 5.2.6 implies that our "algorithm" never terminates: we may compute a sequence of approximations $\widetilde{H}_0, \widetilde{H}_1, \ldots \in \mathbf{Z}/p^\mu\mathbf{Z}$ that all satisfy (5.2.7), continuing happily into the sunset. (This sequence is not uniquely determined, as the congruence (5.2.7) will have more than one solution for $\widetilde{H}_k$ whenever $p \mid k$.)

The next result establishes the promised logarithmic error bound for these approximations.

Proposition 5.2.9. *Let $n \geqslant 1$, and suppose that we are given $\widetilde{H}_0, \ldots, \widetilde{H}_n \in \mathbf{Z}/p^\mu\mathbf{Z}$ such that $\widetilde{H}_0 = H_0$ and such that* (5.2.7) *holds for $k = 1, \ldots, n$. Then*

$$\widetilde{H}_k \equiv H_k \pmod{p^{\mu - \lfloor \log_p k \rfloor}} \tag{5.2.10}$$

for all $k = 1, \ldots, n$ such that $\lfloor \log_p k \rfloor \leqslant \mu$.

Problem 5.2.11. Prove Proposition 5.2.9 as follows.

a) Let

$$\widetilde{H} \coloneqq \widetilde{H}_0 + \widetilde{H}_1 x + \cdots + \widetilde{H}_n x^n \in (\mathbf{Z}/p^\mu\mathbf{Z})[x],$$

and set $E \coloneqq \widetilde{H}/H \pmod{x^{n+1}}$. The same argument as in parts (a) and (b) of Problem 5.2.8 shows that $\partial E = 0$. Deduce that

$$E_k \equiv 0 \pmod{p^{\mu - \nu_p(k)}}$$

for $k = 1, \ldots, n$, whenever $\nu_p(k) \leqslant \mu$.

b) Conclude that (5.2.10) holds whenever $\lfloor \log_p k \rfloor \leqslant \mu$.

Problem 5.2.12. (🥽) Verify Proposition 5.2.6 and Proposition 5.2.9 numerically in your FCAS. That is, for some small parameters g, p and μ, take some random $F \in (\mathbf{Z}/p^\mu\mathbf{Z})[x]$ and compute a sequence $\widetilde{H}_0, \widetilde{H}_1, \ldots \in (\mathbf{Z}/p^\mu\mathbf{Z})$ satisfying (5.2.7). Check that these values agree with the coefficients of $H = F^m$, up to the accuracy indicated in Proposition 5.2.9.

5.3. Back to point counting Let us now see what Proposition 5.2.9 tells us about computing A_f. Recall that we need to compute h_{vp-u} for $1 \leqslant u, v \leqslant g$, which means that we must use the recurrence to compute the coefficients of $h(x)$ up to and including h_{gp-1}. This leads to the following result.

Corollary 5.3.1. *If we take*

$$\mu \coloneqq \lfloor \log_p(gp-1) \rfloor + 1,$$

then the approximations $\widetilde{H}_k$ *are correct modulo* p *for* $k = 0, \dots, gp-1$. *In particular,*

$$h_{vp-u} \equiv \widetilde{H}_{vp-u} \pmod{p}, \qquad 1 \leqslant u, v \leqslant g.$$

Problem 5.3.2. (🎮) Use your FCAS to verify Corollary 5.3.1 for the genus 3 curve from Problem 3.2.3. That is, taking $\mu := \lfloor \log_p(3p-1) \rfloor + 1 = 2$, choose a lift $F \in (\mathbf{Z}/p^2\mathbf{Z})[x]$ of f, and solve (5.2.7) for $\widetilde{H}_1, \dots, \widetilde{H}_{3p-1} \in \mathbf{Z}/p^2\mathbf{Z}$. Check that you do correctly recover those $h_{vp-u} \in \mathbf{F}_p$ appearing in (4.2.4).

(Note that the "naive" analysis would require you to take $\mu = 3$, since you have to divide by p twice.)

After computing all of the h_{vp-u}, we may construct the matrix A_f as in (4.2.4), and then compute $L_C(T) \pmod{p}$ via Corollary 4.4.3. This leads to the following result.

Theorem 5.3.3 (Computing A_f via recurrences)**.** *Let* $C/\mathbf{F}_p$ *be a hyperelliptic curve of genus* $g \geqslant 1$ *defined by* $y^2 = f(x)$ *for some* $f \in \mathbf{F}_p[x]$. *Assume that* $p \geqslant g$. *Then we may compute* A_f *in time*

$$O(g^2 p \log^{1+\varepsilon} p).$$

Problem 5.3.4. Prove Theorem 5.3.3 as follows.

a) Show that for each $k = 1, \dots, gp-1$, we may compute $\widetilde{H}_k \in \mathbf{Z}/p^\mu\mathbf{Z}$ from $\widetilde{H}_{k-1}, \dots, \widetilde{H}_{k-d}$, i.e., solve (5.2.7), in time
$$O(d\,(\mu \log p)^{1+\varepsilon}).$$
(You may need various results from §2.1.)

b) Show that the assumption $p \geqslant g$ implies that we may take $\mu = 2$.

c) Complete the proof of Theorem 5.3.3.

Remark 5.3.5. Recall that in Theorem 4.4.1 we established the complexity bound

$$O(gp \log^2 p)$$

for computing A_f (assuming that $p \gg g$), by simply expanding out the polynomial $h = f^{(p-1)/2}$. It is interesting to compare this to the bound

$$O(g^2 p \log^{1+\varepsilon} p)$$

obtained in this section via recurrences (Theorem 5.3.3). On one hand, we have lost a factor of g. On the other hand, we have gained a factor of almost $\log p$. In practice, we have probably gained much more than this: the real problem with the algorithm in Theorem 4.4.1 is that polynomial multiplication is extremely memory-intensive. By contrast, the recurrence algorithm uses almost no memory: one only has to keep track of the last $d = O(g)$ coefficients. Experience shows that this makes an enormous difference to the overall performance in practice.

Problem 5.3.6. Recall that the cost of multiplication in $\mathbf{F}_p$ is $O(\log p \log\log p)$, whereas the cost of division is $O(\log p\,(\log\log p)^2)$ (see §2.1). When solving (5.2.7), we need to divide by k. Show how to modify the algorithm to remove

almost all of these divisions. Conclude that the complexity bound in Theorem 5.3.3 may be improved to $O(g^2 p \log p \log\log p)$.

(This sort of optimisation is very important in practice. Divisions in $\mathbf{F}_p$ are typically *much* more expensive than multiplications, by a factor much worse than the theoretical $\log\log p$.)

Problem 5.3.7. (ϟ) Recall from §5.1 that we may compute the *first row* of A_f without any divisions by p. Sutherland and I showed that it is possible to recover the entire matrix A_f from knowledge of just the first rows of the matrices corresponding to g *translates* of the original curve [26, §5]. Specifically, one may take the curves $y^2 = f_i(x)$ with $f_i(x) := f(x + c_i)$ for $i = 1, \ldots, g$, where $c_1, \ldots, c_g$ are distinct elements of $\mathbf{F}_p$. Doing things this way, one needs to solve (5.2.7) the same number of times as before, but working modulo p everywhere instead of modulo p^2. The proof in [26] relies on the interpretation of A_f as the matrix of a Frobenius operator on differentials. Can you give an "elementary" argument, more in the style of the proof of Theorem 4.2.7 suggested in Problem 4.2.9?

Remark 5.3.8. I am unsure of the history of the main idea of this section, i.e., using a linear recurrence to compute selected coefficients of high index in a large power of a univariate polynomial of small degree. I learned this idea from [7], and the earliest reference of which I am aware is [18].

6. A square-root time algorithm

In the previous sections we have seen two algorithms for computing A_f whose complexity is roughly linear in p (Theorem 4.4.1 and Theorem 5.3.3). In this section we will see how to improve this to roughly linear in the *square root* of p.

6.1. A warm-up: Wilson primes To explain the basic idea, let us first consider a simpler problem. Recall that Wilson's theorem in elementary number theory states that for any prime p,

$$(p-1)! \equiv -1 \pmod{p}.$$

We say that p is a *Wilson prime* if the above congruence holds modulo p^2. For example, 5 is a Wilson prime because $4! = 24 \equiv -1 \pmod{5^2}$. Only three Wilson primes are known: 5, 13 and 563. The search for Wilson primes has an interesting history; see [11] for more information.

Suppose we want to test whether a given prime p is a Wilson prime. How quickly can we compute $(p-1)! \pmod{p^2}$?

We could simply compute $(p-1)! \in \mathbf{Z}$ and then find its remainder modulo p^2, but this is grossly inefficient because $(p-1)!$ grows very rapidly with p.

A better method is to start with 1, and then multiply successively by 2, 3, and so on up to $p-1$, taking the remainder modulo p^2 after each multiplication. Using this method we never need to consider integers larger than about p^3, and the running time is essentially linear in p.

The following theorem shows that we can do much better than this.

Theorem 6.1.1. *We may compute* $(p-1)! \pmod{p^2}$ *in time*

$$O(p^{1/2} \log^3 p).$$

To prove the theorem, let $s \coloneqq p-1$ and consider the product

$$1 \times 2 \times \cdots \times s \pmod{p^2}.$$

The idea is to split up this product into roughly $s^{1/2}$ groups of $s^{1/2}$ terms. Define

$$t \coloneqq \lfloor s^{1/2} \rfloor,$$

so that

$$s = t^2 + t', \qquad 0 \leqslant t' \leqslant 2s^{1/2}.$$

Now break up the product into t groups of t terms, plus t' leftover terms, i.e.,

$$\big(1 \times \cdots \times t\big)\big((t+1) \times \cdots \times 2t\big) \cdots \big(((t-1)t+1) \times \cdots \times t^2\big)\big((t^2+1) \times \cdots \times s\big).$$

Introducing the polynomial

$$Q(k) \coloneqq (k+1)(k+2)\cdots(k+t) \in (\mathbf{Z}/p^2\mathbf{Z})[k],$$

the product may be rewritten as

$$Q(0) \cdot Q(t) \cdot Q(2t) \cdots Q((t-1)t) \cdot \big((t^2+1) \times \cdots \times s\big).$$

Problem 6.1.2. Prove Theorem 6.1.1 as follows.

a) Show that we may use a product tree (see §2.4) to compute $Q \in (\mathbf{Z}/p^2\mathbf{Z})[k]$ in time

$$O(p^{1/2} \log^3 p).$$

(By "compute Q", we mean compute its coefficients.)

b) Show that we may evaluate $Q(k)$ at the t points $k = 0, t, \ldots, (t-1)t$ via fast multipoint evaluation (Proposition 2.4.4) in time

$$O(p^{1/2} \log^3 p).$$

c) Show that we may multiply together the values found in (b), and the t' leftover terms, in time

$$O(p^{1/2} \log p \log\log p).$$

Problem 6.1.3. (🕶) Implement the above method for computing $(p-1)! \pmod{p^2}$ in your FCAS. Can you observe the running time growing roughly like $p^{1/2}$? How big does p need to get before your implementation beats the straightforward linear-time method, i.e., multiplying successively by $2, \ldots, p-1 \pmod{p^2}$?

Problem 6.1.4. The above strategy for computing factorials was first proposed by Strassen [49]. As an application, he described the following deterministic algorithm for integer factorisation. Consider the simplest case where we want to factor a number of the form $N = pq$, where p and q are distinct (unknown) primes. Show that we may compute $\lfloor N^{1/2} \rfloor! \pmod{N}$, and hence recover the

factors p and q, in time $O(N^{1/4} \log^3 N)$. (A similar scheme was suggested by Pollard [44] at around the same time.)

Problem 6.1.5. *Wolstenholme's theorem* states that

$$\binom{2p-1}{p-1} \equiv 1 \pmod{p^3}$$

for all primes $p \geqslant 5$. A *Wolstenholme prime* is a prime that satisfies the congruence modulo p^4. Describe an algorithm that can test whether p is a Wolstenholme prime in time $O(p^{1/2} \log^3 p)$.

(Several other characterisations of Wolstenholme primes are known; see for example [39]. There are only two Wolstenholme primes less than 10^{11}, namely 16843 and 2124679 [6]. I am not aware of any explicit mention in the literature of an $O(p^{1/2+\varepsilon})$-time algorithm for this problem, but it is hinted at in [39].)

Remark 6.1.6. Bostan, Gaudry and Schost [7] gave an improvement to Strassen's method that reduces the complexity in Theorem 6.1.1 to

$$O(p^{1/2} \log^2 p).$$

They never actually compute the coefficients of $Q(k)$. Instead, they give a subroutine that takes as input the values of the polynomial $Q_n(k) \coloneqq (k+1)\cdots(k+n)$ at n points in an arithmetic progression, and outputs the values of $Q_{2n}(k) \coloneqq (k+1)\cdots(k+2n)$ at $2n$ points in a related arithmetic progression, at the cost of $O(1)$ polynomial multiplications of degree n. They apply this doubling process repeatedly to "grow" the polynomial and the set of evaluation points in parallel to reach the desired size. As far as I am aware, this is the best published complexity bound for computing $(p-1)! \pmod{p^2}$ for a single prime p.

Remark 6.1.7. A downside of all variants of the square-root time algorithm is that they use a lot of memory, and this is usually the bottleneck that limits the size of the problems that can be tackled in practice. The basic version worked out in Problem 6.1.2 uses $O(p^{1/2} \log^2 p)$ space; with some effort this can be reduced to $O(p^{1/2} \log p)$ [55, Lemma 2.1]. The Bostan–Gaudry–Schost variant also uses $O(p^{1/2} \log p)$ space. In all of these algorithms, it is possible to make a time-space tradeoff by adjusting the degree and the number of evaluation points. Roughly speaking, one can save a factor of $\alpha > 1$ in space by giving up a factor of α in time.

Problem 6.1.8. (🗲) Can the complexity of Theorem 6.1.1 be further improved via the "factorial sieving" scheme described in [14, §2]? (This is also closely related to the integer factorisation algorithm in [12].) For example, to compute $N!$, we may split into odd and even terms, and remove a factor of 2 from each even term. Iterating this process, we arrive at a factorisation of the form

$$N! = 2^n (1 \times 3 \times \cdots \times N_0)(1 \times 3 \times \cdots \times N_1) \cdots$$

where $N_i \approx N/2^i$ for each $i \geqslant 0$. Notice that the first product has length roughly $N/2$ and the remaining products are subproducts of the first one. I would therefore expect that the entire expression could be computed using a modified square-root scheme in roughly the same time as the first product. This should save a factor of $2^{1/2}$ compared to handling $N!$ directly. Now suppose that we use the same trick to pull out factors of a few other small primes, not just the prime 2. Does this lead to an *asymptotic* improvement in the complexity of Theorem 6.1.1? My guess would be perhaps

$$O\left(\frac{p^{1/2}\log^2 p}{(\log\log p)^{1/2}}\right)?$$

6.2. Solving recurrences in square-root time Let us now return to the situation from §5. Recall that we have a polynomial

$$F = F_0 + F_1 x + \cdots + F_d x^d \in (\mathbf{Z}/p^\mu\mathbf{Z})[x]$$

where $d := 2g+2$, and we assume that $F_0 \not\equiv 0 \pmod p$. We are interested in computing certain coefficients of $H := F^m$ where $m := (p-1)/2$.

It will be helpful to re-express the recurrence (5.2.2) in matrix-vector form. For $k \geqslant 0$, define a row vector U_k consisting of d consecutive coefficients of H, namely

$$U_k := (H_{k-d+1}, \ldots, H_{k-1}, H_k) \in (\mathbf{Z}/p^\mu\mathbf{Z})^d.$$

Then (5.2.2) may be rewritten as

$$kU_k = \frac{1}{F_0} U_{k-1} T_k \tag{6.2.1}$$

where $T_k \in \mathrm{Mat}_d(\mathbf{Z}/p^\mu\mathbf{Z})$ is the *transition matrix* given by

$$T_k := \begin{pmatrix} 0 & \cdots & 0 & 0 & (d(m+1)-k)F_d \\ kF_0 & & & & \vdots \\ & \ddots & & & \vdots \\ & & kF_0 & & (2(m+1)-k)F_2 \\ & & & kF_0 & (m+1-k)F_1 \end{pmatrix}. \tag{6.2.2}$$

The last column of T_k implements (5.2.2) to compute kF_0H_k, and the remaining off-diagonal entries simply slide the window over from index $k-1$ to k. The "initial condition" is given by

$$U_0 = (0, \ldots, 0, H_0).$$

To compute a single coefficient H_n for a large index n, we want to chain several steps together. Using (6.2.1) repeatedly, we obtain

$$(k+1)\cdots(k+s)\cdot U_{k+s} = \frac{1}{(F_0)^s} U_k T_{k+1} \cdots T_{k+s}$$

for any $k \geqslant 0$, $s \geqslant 1$. This shows that the s-fold matrix product

$$T_{k+1} \cdots T_{k+s} \in \mathrm{Mat}_d(\mathbf{Z}/p^\mu\mathbf{Z})$$

may be viewed as a transition matrix going from index k to index $k+s$.

The most obvious algorithm for computing such an s-fold product is to simply multiply the matrices together one at a time. This has complexity growing linearly in s, and is obviously no better than executing the recurrence separately for each k. The next result shows that the complexity can be improved to roughly linear in $s^{1/2}$, by using a matrix generalisation of the square-root trick from §6.1.

Proposition 6.2.3. *Given $k_0 \in \mathbf{Z}/p^\mu\mathbf{Z}$ and an integer $s \geqslant 2$ with $s \ll p^{O(1)}$, we may compute the matrix product*

$$T_{k_0+1} \cdots T_{k_0+s}$$

in time

$$O\big(d^\omega \, M_{\mathrm{int}}(s^{1/2}\mu \log p) \log s\big).$$

Proposition 6.2.3 may be be proved by the same strategy as Theorem 6.1.1, i.e., splitting up the product into roughly $s^{1/2}$ groups of $s^{1/2}$ terms. The following problem asks you to fill in the details.

Problem 6.2.4.

a) Let $t \coloneqq \lfloor s^{1/2} \rfloor$ and $t' \coloneqq s - t^2$ as in §6.1, and define
$$Q(k) \coloneqq T_{k+1} \cdots T_{k+t} \in \mathrm{Mat}_d((\mathbf{Z}/p^\mu\mathbf{Z})[k]).$$
Show that the entries of $Q(k)$ are polynomials in k of degree at most t.

b) Show that we may use a product tree to compute $Q(k)$, i.e., compute the coefficients of the polynomial in each entry of the matrix, in time
$$O\big(d^\omega \, M_{\mathrm{int}}(s^{1/2}\mu \log p) \log s\big).$$

c) Using fast multipoint evaluation, show that we may compute the values
$$Q(k_0), Q(k_0+t), \ldots, Q(k_0+(t-1)t) \in \mathrm{Mat}_d(\mathbf{Z}/p^\mu\mathbf{Z})$$
in time
$$O\big(d^2 \, M_{\mathrm{int}}(s^{1/2}\mu \log p) \log s\big).$$

d) Finish the proof of Proposition 6.2.3 by estimating the cost of multiplying together the values computed in (c) and the t' leftover matrices.

Problem 6.2.5. *Kurepa's conjecture* states that there exists no odd prime p such that p divides the "left factorial" of p, which is defined to be

$$!p \coloneqq 0! + 1! + \cdots + (p-1)!.$$

By setting up an appropriate recurrence involving 2×2 matrices, design an algorithm that can test Kurepa's conjecture for a single prime p in time $O(p^{1/2} \log^3 p)$. (An algorithm along these lines is mentioned briefly in [4].)

Remark 6.2.6. A matrix version of the square-root trick first appeared in [9, §6], and several improvements were given in [7]. The latter improves the complexity

of Proposition 6.2.3 to

$$O\big(d^{\omega} s^{1/2} M_{\mathrm{int}}(\mu \log p) + d^2 M_{\mathrm{int}}(s^{1/2} \mu \log p)\big).$$

6.3. Back to point counting We now discuss how to apply Proposition 6.2.3 to the problem of computing A_f, i.e., computing $H_{\nu p - u} \pmod p$ for $1 \leqslant u, \nu \leqslant g$.

For each $\nu = 1, \dots, g$, the coefficients $H_{\nu p - u}$ for $u = 1, \dots, g$ (corresponding to the entries of the ν-th row of A_f) are exactly the last g entries of the vector $U_{\nu p - 1}$. Therefore, we may rephrase our goal as wanting to compute the vectors

$$U_{p-1}, U_{2p-1}, \dots, U_{gp-1} \pmod p.$$

Of course, we must still deal with the complications arising from the divisions by p that were discussed in §5. Let $\widetilde{H}_0, \widetilde{H}_1, \dots \in \mathbf{Z}/p^{\mu}\mathbf{Z}$ be the sequence of approximations for H_k defined in §5, and let

$$\widetilde{U}_k \coloneqq (\widetilde{H}_{k-d+1}, \dots, \widetilde{H}_{k-1}, \widetilde{H}_k) \in (\mathbf{Z}/p^{\mu}\mathbf{Z})^d$$

be the corresponding vectors.

Notice that for any $k \geqslant 1$ we can compute $\widetilde{U}_k$ from $\widetilde{U}_{k-1}$ by exactly the same method explained in §5, i.e., after sliding all the entries over by one slot, we can compute the last entry of $\widetilde{U}_k$, namely $\widetilde{H}_k$, by solving (5.2.7) (see Problem 5.3.4(a)). Recall that when $k \equiv 0 \pmod p$, the resulting value of $\widetilde{H}_k$, and hence $\widetilde{U}_k$, is not uniquely determined.

On the other hand, when $k \not\equiv 0 \pmod p$, the vector $\widetilde{U}_k$ *is* uniquely determined by $\widetilde{U}_{k-1}$. In fact, from (6.2.1) we have

$$\widetilde{U}_k = \frac{1}{kF_0} \widetilde{U}_{k-1} T_k.$$

Chaining $p-1$ of these identities together, for any $i \geqslant 0$ we obtain

$$\widetilde{U}_{ip+p-1} = \frac{1}{(ip+1)\cdots(ip+p-1)(F_0)^{p-1}} \widetilde{U}_{ip} T_{ip+1} \cdots T_{ip+p-1}. \tag{6.3.1}$$

The above discussion suggests the following strategy. Starting with $\widetilde{U}_0$, we use the matrix $T_1 \cdots T_{p-1}$ to sprint to $\widetilde{U}_{p-1}$ in square-root time with the help of Proposition 6.2.3; then take a single step to $\widetilde{U}_p$; then sprint again to $\widetilde{U}_{2p-1}$ with the matrix $T_{p+1} \cdots T_{2p-1}$, and so on. This strategy may be formalised more precisely as follows.

Proposition 6.3.2. *Suppose that the quantities*

$$T_{ip+1} \cdots T_{ip+p-1} \in \mathrm{Mat}_d(\mathbf{Z}/p^{\mu}\mathbf{Z}) \tag{6.3.3}$$

and

$$(ip+1)\cdots(ip+p-1) \in \mathbf{Z}/p^{\mu}\mathbf{Z} \tag{6.3.4}$$

are known for $i = 0, \dots, g-1$. *Then we may compute* A_f *in time*

$$O\big((g^3 + \log p)(\mu \log p)^{1+\varepsilon}\big).$$

Problem 6.3.5. Prove Proposition 6.3.2 as follows.

a) Show that H_0, and hence $\widetilde{U}_0 = U_0$, may be computed using $O(\log p)$ arithmetic operations in $\mathbf{Z}/p^\mu\mathbf{Z}$.
b) Using (6.3.1), show that for each i, we may compute $\widetilde{U}_{ip+p-1}$ from $\widetilde{U}_{ip}$ using $O(d^2)$ operations in $\mathbf{Z}/p^\mu\mathbf{Z}$.
c) Show that for each i, we may compute $\widetilde{U}_{(i+1)p}$ from $\widetilde{U}_{ip+p-1}$ using $O(d)$ operations in $\mathbf{Z}/p^\mu\mathbf{Z}$.
d) Finish the proof of Proposition 6.3.2.

The last remaining piece of the puzzle is to actually compute the quantities (6.3.3) and (6.3.4) for $i = 0, \dots, g-1$. The obvious thing to do is to apply Proposition 6.2.3 to handle each value of i in square-root time. But we have one more trick up our sleeves, which enables us to save a factor of $O(g)$. Recall from Problem 5.3.4(b) that if we impose the mild assumption $p \geqslant g$, then we can get away with a working precision of $\mu = 2$. This leads to the following "p-adic interpolation" optimisation.

Proposition 6.3.6. *Assume that* $\mu = 2$*, and suppose that* (6.3.3) *and* (6.3.4) *are known for* $i = 0$ *and* $i = 1$*. Then we may compute* (6.3.3) *and* (6.3.4) *for the remaining values* $i = 2, \dots, g-1$ *in time*

$$O(g^3 \log^{1+\varepsilon} p).$$

Problem 6.3.7. Prove Proposition 6.3.6 as follows. We only deal with (6.3.3); handling (6.3.4) is even easier. Consider the matrix

$$R(k) \coloneqq T_{k+1} \cdots T_{k+p-1} \in \mathrm{Mat}_d((\mathbf{Z}/p^2\mathbf{Z})[k]).$$

Let us write this as

$$R(k) = R_0 + R_1 k + \cdots + R_{p-1} k^{p-1},$$

for matrices $R_0, \dots, R_{p-1} \in \mathrm{Mat}_d(\mathbf{Z}/p^2\mathbf{Z})$.

a) Show that given $R(0)$ and $R(p)$, we may compute $R_0 \pmod{p^2}$ and $R_1 \pmod{p}$ using $O(d^2)$ operations in $\mathbf{Z}/p^2\mathbf{Z}$.
b) Then show that for any i, we may deduce $R(ip)$ using $O(d^2)$ operations in $\mathbf{Z}/p^2\mathbf{Z}$.
c) Finish the proof of Proposition 6.3.6.

Putting everything together, we finally obtain a square-root bound for the cost of computing A_f:

Theorem 6.3.8 (Computing A_f in square-root time)**.** *Let* $C/\mathbf{F}_p$ *be a hyperelliptic curve of genus* $g \geqslant 1$ *defined by* $y^2 = f(x)$ *for some* $f \in \mathbf{F}_p[x]$*. Assume that* $p \geqslant g^2$*. Then we may compute* A_f *in time*

$$O(g^\omega p^{1/2} \log^3 p).$$

Problem 6.3.9. Prove Theorem 6.3.8, by first using Proposition 6.2.3 to evaluate (6.3.3) for $i = 0, 1$ (and a similar method to handle (6.3.4)), and then applying Proposition 6.3.6 and Proposition 6.3.2.

Remark 6.3.10. In the special case $g = 1$, we of course only need (6.3.3) and (6.3.4) for $i = 0$, not for $i = 1$. When $g = 2$, we need both values of i, but we need not bother with Proposition 6.3.6.

Problem 6.3.11. (🎮) Implement the algorithm from Theorem 6.3.8 in your FCAS. Can you observe the running time behaving like $p^{1/2}$, and growing polynomially in g? How big does p have to get to beat the linear-time algorithms from Theorem 4.4.1 and Theorem 5.3.3?

Problem 6.3.12. The products (6.3.3) for $i = 0$ and $i = 1$ are actually subproducts of the longer product $T_1 \cdots T_{2p-1}$. Show that by tweaking the proof of Proposition 6.2.3, it is possible to compute both products in about the same time as it takes to compute the longer product. Show that for large p, this trick speeds up the computation of A_f by a factor of about $2^{1/2}$.

(For a generalisation, see [7, §7].)

Remark 6.3.13. Using the Bostan–Gaudry–Schost improvements mentioned in Remark 6.2.6, the complexity in Theorem 6.3.8 may be improved to

$$O\big(g^{\omega} p^{1/2} \log p \log\log p + g^2 p^{1/2} \log^2 p\big).$$

When p is large compared to g, this is the best complexity bound that I know for the problem of computing A_f. Note that this bound improves on [7, Theorem 17 (part 2)] by a factor of about $g^{3/2}$, due to the use of better p-adic error bounds (Proposition 5.2.9) and subsequent use of the interpolation trick (Proposition 6.3.6). Note also that for p sufficiently large relative to g, the second term is dominant, and this is optimal with respect to g since the output (the matrix A_f) has size $O(g^2)$. I suspect, but have not checked in detail, that the same complexity bound (up to a constant factor) could be obtained by the methods of [21], which works in the framework of p-adic cohomology (i.e., Kedlaya's algorithm). If this is the case, it would be interesting to compare (theoretically) the constant factors between the two algorithms. It would also be interesting to write an efficient implementation and compare its performance against the code accompanying [21].

7. An average polynomial time algorithm

Suppose that someone hands us a hyperelliptic curve C defined not over a finite field, but over $\mathbf{Q}$, say $y^2 = \bar{f}(x)$ for $\bar{f} \in \mathbf{Z}[x]$. Then for each prime $p \geqslant 3$, we may reduce $\bar{f}$ modulo p to obtain a polynomial $f_p \in \mathbf{F}_p[x]$, and hence an equation $y^2 = f_p(x)$. For all but finitely many p, this equation defines a hyperelliptic curve C_p over $\mathbf{F}_p$ (see §7.2), and for these "good" primes it makes sense to consider the zeta function of C_p,

$$Z_{C_p}(T) = \frac{L_{C_p}(T)}{(1-T)(1-pT)}.$$

One might guess that the L-polynomials $L_p(T) \coloneqq L_{C_p}(T)$ for different primes should be completely unrelated: knowing the number of solutions to an equation

modulo one prime p_1 should tell us nothing at all about the number of solutions modulo a different prime p_2. Nevertheless, when one averages over many primes, intriguing statistical regularities begin to appear. Examples include the Birch and Swinnerton-Dyer conjecture and the Sato–Tate conjecture, both of which were originally formulated for elliptic curves, and later generalised to curves of higher genus, including hyperelliptic curves.

Further discussion of these phenomena is beyond the scope of this course, but they motivate the following computational question: given a large integer N, how quickly can we compute $L_p(T)$ for all (good) primes $p \leqslant N$?

In Theorem 6.3.8 we saw that for a single prime p we can compute A_{f_p}, and hence $L_p(T) \pmod p$, in time $O(g^\omega p^{1/2} \log^3 p)$. If we do this separately for each prime $p \leqslant N$, the overall complexity is

$$O(g^\omega N^{3/2+\varepsilon}).$$

In this section we will see that it is possible to do much better! In fact, we will show that we can compute A_{f_p} for all $p \leqslant N$ (except finitely many) in time

$$O(g^\omega N \log^3 N).$$

This is incredibly fast — for fixed g, it is almost *linear* in the size of the output! Indeed, each A_{f_p} needs $O(g^2 \log p)$ bits to write down, so according to Lemma 1.5.2 the total output size is

$$\sum_{p \leqslant N} O(g^2 \log p) = O(g^2 N).$$

Another point of view is that the *average* complexity per prime $p \leqslant N$ is

$$\frac{O(g^\omega N \log^3 N)}{N/\log N} = O(g^\omega \log^4 N).$$

Loosely speaking, the time spent on each prime p is only $O(g^\omega \log^4 p)$, which is polynomial in the bit size of f_p. For this reason, the algorithm is said to run in "average polynomial time". In §8 we will see how to extend this result to compute not just $L_p(T) \pmod p$, but the entire L-polynomial $L_p(T) \in \mathbf{Z}[T]$, for all $p \leqslant N$, in average polynomial time. (Compare with Remark 3.3.3.)

All of these average polynomial time point counting algorithms depend on a marvellous device called the *accumulating remainder tree*. The accumulating remainder tree was invented by Robert Gerbicz, an applied mathematics student, in the context of a large-scale search for Wilson primes. (See §6.1 for the definition of Wilson primes.) He outlined the algorithm in a 2011 post on mersenneforum.org [19], and it appeared in published form a few years later [11]. In retrospect, it is surprising that the algorithm was not discovered decades earlier. The basic ingredients were already known by the 1980s, but somehow all the experts missed it.

7.1. A warm-up: Wilson primes The Wilson prime example provides an excellent introduction to the main ideas behind the accumulating remainder tree. In

this section we will look at this example in detail, and prove the following theorem.

Theorem 7.1.1. *Given an integer* $N \geqslant 2$*, we may compute* $(p-1)! \pmod{p^2}$ *for all primes* $p \leqslant N$ *in time*

$$O(N \log^3 N).$$

To be very explicit, the quantities that we want to compute are

$$\begin{aligned} &1 \times 2 && \pmod{3^2}, \\ &1 \times 2 \times 3 \times 4 && \pmod{5^2}, \\ &1 \times 2 \times 3 \times 4 \times 5 \times 6 && \pmod{7^2}, \\ &1 \times 2 \times 3 \times 4 \times 5 \times 6 \times 7 \times 8 \times 9 \times 10 && \pmod{11^2}, \end{aligned} \tag{7.1.2}$$

and so on, for all $p \leqslant N$.

For simplicity, we assume throughout that N is a power of two, say $N = 2^b$.

We start by defining a certain binary tree called the *modulus tree*. For $j = 1, \dots, N$, define

$$m_j \coloneqq \begin{cases} j^2 & \text{if } j \text{ is prime,} \\ 1 & \text{otherwise.} \end{cases}$$

Then the modulus tree is the product tree associated to $m_1, \dots, m_N$ (see §2.4). An example for $N = 16$ is shown in Figure 7.1.3.

We index the levels of the tree by $\ell = 0, \dots, b$, where $\ell = 0$ corresponds to the root and $\ell = b$ to the leaves. The nodes at level ℓ are labelled by the pairs (ℓ, j) for $j = 1, \dots, 2^\ell$. We write $m_{\ell,j}$ for the value in the modulus tree at the node (ℓ, j); thus for example $m_{b,j} = m_j$ for $j = 1, \dots, N$, and $m_{0,1} = \prod_{p \leqslant N} p^2$.

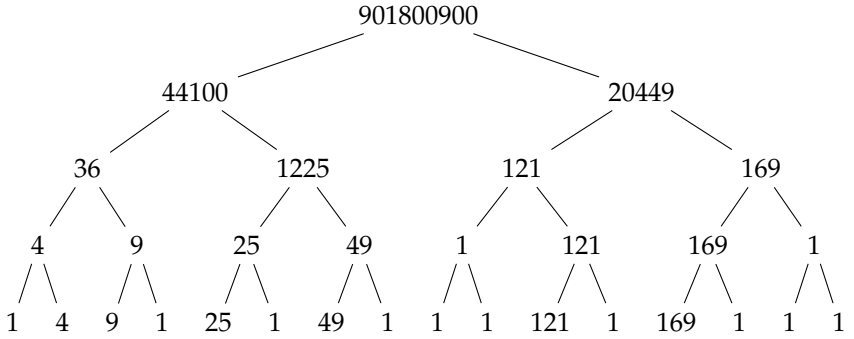

Figure 7.1.3. The modulus tree for $N = 16$.

Lemma 7.1.4. *We may compute the modulus tree, i.e., all values* $m_{\ell,j}$*, in time*

$$O(N \log^2 N).$$

Problem 7.1.5. Use Lemma 1.5.2 to show that the product $\prod_{p \leqslant N} p^2$ has $O(N)$ bits. Then prove Lemma 7.1.4 via Proposition 2.6.1 and Proposition 2.4.2.

We now define a second binary tree, unimaginatively named the *value tree*, to be the product tree on the values $V_j \coloneqq j$ for $j = 1, \ldots, N$. We write $V_{\ell,j}$ for the value at the node (ℓ, j). For instance, the value at the root node is $V_{0,1} = N!$. An example for $N = 16$ is shown in Figure 7.1.6.

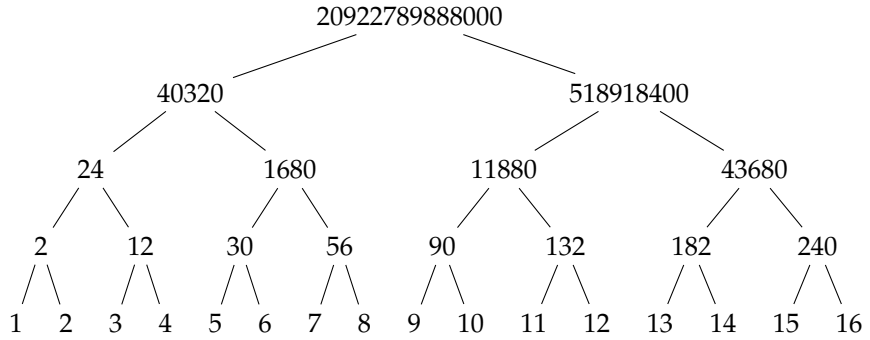

Figure 7.1.6. The value tree for $N = 16$.

Lemma 7.1.7. *We may compute the value tree, i.e., all values* $V_{\ell,i}$*, in time*

$$O(N \log^3 N).$$

Problem 7.1.8. Prove Lemma 7.1.7 along the same lines as the proof of Lemma 7.1.4, by first showing that the bit size of $N!$ is $O(N \log N)$.

Finally we arrive at our third and most interesting binary tree, the *accumulating remainder tree*. The value assigned to the node (ℓ, j) is defined to be

$$R_{\ell,j} \coloneqq V_{\ell,1} \cdots V_{\ell,j-1} \pmod{m_{\ell,j}},$$

thought of as an element of $\mathbf{Z}/m_{\ell,j}\mathbf{Z}$. In other words, $R_{\ell,j}$ is the product of all values strictly to the *left* of the node (ℓ, j) in the value tree, taken modulo the corresponding value $m_{\ell,j}$ in the modulus tree. Note that if $j = 1$, the product is empty so by definition $R_{\ell,j} = 1 \pmod{m_{\ell,j}}$. See Figure 7.1.9.

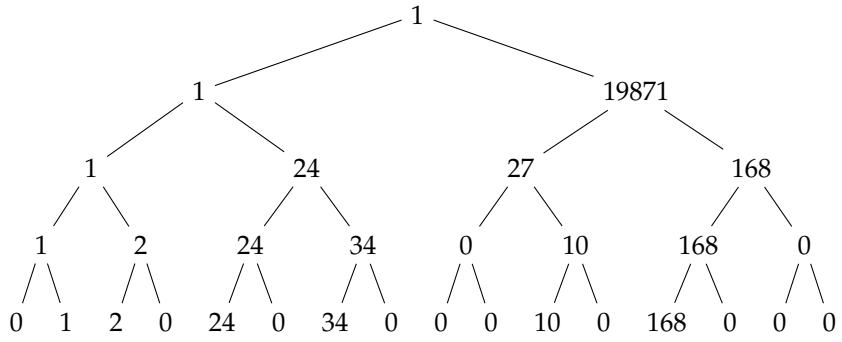

Figure 7.1.9. The accumulating remainder tree for $N = 16$.

Problem 7.1.10. Show that for any prime $p \leqslant N$,

$$R_{b,p} = (p-1)! \pmod{p^2}.$$

In other words, the values in the leaf nodes of the accumulating remainder tree are exactly the Wilson remainders. For example, in Figure 7.1.9 we can immediately see that $p = 5$ and $p = 13$ are the only Wilson primes less than 16.

The following result is the engine that drives the whole accumulating remainder tree algorithm.

Lemma 7.1.11. *Let (ℓ, j) be a non-leaf node, i.e., with $\ell < b$. Then*

$$R_{\ell+1,2j-1} \equiv R_{\ell,j} \pmod{m_{\ell+1,2j-1}},$$
$$R_{\ell+1,2j} \equiv R_{\ell,j} V_{\ell+1,2j-1} \pmod{m_{\ell+1,2j}}.$$

Lemma 7.1.11 says that if we start with any non-leaf node, then its *left child* is obtained by simply reducing its value modulo the corresponding modulus tree node, and that its *right child* is obtained by first multiplying by an appropriate node from the value tree, and then reducing modulo the corresponding modulus tree node. For example, in Figure 7.1.9, starting with the value 19871 at the node $(\ell, j) = (1, 2)$, the value of its left child is $R_{2,3} = 19871 \bmod 121 = 27$, and its right child is $R_{2,4} = 19871 \cdot 11880 \bmod 169 = 168$.

Corollary 7.1.12. *Assume that the modulus tree and value tree are known. We may compute the remainder tree, i.e., all values $R_{\ell,j}$, in time*

$$O(N \log^3 N).$$

Problem 7.1.13. Prove Lemma 7.1.11, and then deduce Corollary 7.1.12 by using Lemma 7.1.11 to show that if we know the values of the accumulating remainder tree at level ℓ for some $\ell < b$, then we may compute the values at level $\ell + 1$ in time

$$O(N \log^2 N).$$

The main theorem of this section, Theorem 7.1.1, now follows immediately by stringing together Lemma 7.1.4, Lemma 7.1.7, Corollary 7.1.12, and Problem 7.1.10.

Problem 7.1.14. (🥽) Implement the algorithm described in Theorem 7.1.1 to compute $(p-1)! \pmod{p^2}$ for all $p \leqslant N$ in your FCAS. For what N is it faster than running the $O(p^{1/2+\varepsilon})$ implementation from Problem 6.1.3 separately for each $p \leqslant N$?

An algorithm similar to the one described above was used in [11] to carry out a large-scale search for Wilson primes. The computation expended about 1.1 million hours of CPU time (in the early 2010s), and found that there are no Wilson primes less than 2×10^{13}, apart from the three already known. The main difficulty in this project was memory usage: for $N = 2 \times 10^{13}$, just writing down the value tree, as defined above, would require about 5000 terabytes of storage!

For this and other reasons, the algorithm actually used in [11] was considerably more complicated than the one presented in this section.

Problem 7.1.15. Describe an algorithm that can find all Wolstenholme primes $p \leqslant N$ in time $O(N \log^3 N)$.

(See Problem 6.1.5 for the definition of Wolstenholme primes. The observation that this problem yields to an average polynomial time algorithm is not new — I learned of this from Robert Gerbicz many years ago. But it does not yet seem to have appeared in print.)

Problem 7.1.16. Describe an algorithm that can find all counterexamples $p \leqslant N$ to Kurepa's conjecture in time $O(N \log^3 N)$. Note that some of the trees will now consist of *matrices* instead of scalars.

(See Problem 6.2.5 for the statement of Kurepa's conjecture. An average polynomial time algorithm for this problem was described in [4], and the authors used it to show that there are no counterexamples less than $2^{40} \approx 1.1 \times 10^{12}$. They also expressed the opinion, which I share, that Kurepa's conjecture is almost certainly false.)

Problem 7.1.17. (ϟ) Can the "factorial sieving" scheme in [14, §2] (see Problem 6.1.8) be used to improve the complexity of Theorem 7.1.1, perhaps to

$$O\left(\frac{N \log^3 N}{\log\log N}\right)?$$

Problem 7.1.18. (ϟϟ) Is it possible to improve the $O(N \log^3 N)$ complexity bound for the accumulating remainder tree algorithm? I have wondered about this for many years. The bottleneck seems to be the value tree, which occupies altogether $O(N \log^2 N)$ bits of space, and really does seem to require $O(N \log^3 N)$ time to build. On the other hand, in some sense only $O(N \log N)$ bits of information from the value tree are fed into the the accumulating remainder tree, namely the remainders $V_{\ell+1,2j-1} \pmod{m_{\ell+1,2j}}$ (see Lemma 7.1.11). This suggests that when we compute the value tree, we are somehow computing too much data by a factor of $\log N$. Can we somehow use this observation to improve the overall complexity to $O(N \log^2 N)$? Or even something less ambitious like $O(N \log^3 N / \log\log N)$?

Note that for the specific case of Wilson primes, there is a lot of additional structure that might lead to a provable asymptotic speedup; see for example Problem 7.1.17, or the factorial identities described in [11, §3]. That would be interesting in its own right, but in this problem what I am really asking for is a generic speedup that applies to all instances of the accumulating remainder tree, including the point counting algorithms presented in the next section.

7.2. Back to point counting Let us now return from our Wilson prime detour to the problem of point counting on hyperelliptic curves.

The setup is that $C/\mathbf{Q}$ is a hyperelliptic curve of genus $g \geqslant 1$ given by $y^2 = \bar{F}(x)$ where $\bar{F} \in \mathbf{Z}[x]$ is squarefree of degree $2g+1$ or $2g+2$. To avoid problems with

recurrences later on, let us also assume that

$$\bar{F}_0 \neq 0.$$

(The $\bar{F}_0 = 0$ case may be handled along similar lines to Remark 5.1.7.) For each prime $p \geqslant 3$, we write $f_p \in \mathbf{F}_p[x]$ for the reduction of $\bar{F}$ modulo p.

Definition 7.2.1. We say that a prime $p \geqslant 3$ is *admissible* for $\bar{F}$ if $p \geqslant g$, and if the polynomial $f_p \in \mathbf{F}_p[x]$ is squarefree, has the same degree as $\bar{F}$, and has nonzero constant term.

Equivalently, $p \geqslant g$ is admissible if it does not divide the leading coefficient, constant term, or discriminant of $\bar{F}$. This implies that there are only finitely many inadmissible primes for $\bar{F}$.

Remark 7.2.2. The above definition of admissibility is nonstandard, and is customised for the requirements of the algorithms later in this section. It is closely related to the concept of *good reduction* (for a definition see [30, §A.9.1]), but is not the same. If p is admissible for $\bar{F}$, then C has good reduction at p, but the converse is false in general.

If p is admissible for $\bar{F}$, then the curve C_p defined by $y^2 = f_p(x)$ is a hyperelliptic curve over $\mathbf{F}_p$ of the same genus g, so it makes sense to define $L_p(T) \coloneqq L_{C_p}(T)$.

Now suppose that we are given some bound $N \geqslant 3$, and we wish to compute $L_p(T) \pmod p$ for all admissible $p \leqslant N$. By Theorem 4.3.2, it suffices to compute the matrices

$$A_{f_p} \in \mathrm{Mat}_g(\mathbf{F}_p)$$

for all admissible $p \leqslant N$. In other words, we want to compute the quantities

$$(\bar{F}^{(p-1)/2})_{vp-u} \pmod p, \qquad 1 \leqslant u, v \leqslant g$$

for each p. At first glance this looks hopeless. Not only is the modulus different for each p, but we want to compute different coefficients for each p, and we are even dealing with a different polynomial $\bar{F}^{(p-1)/2}$ for each p! It is not at all obvious that there is any redundancy between the various primes that can be exploited.

To make progress, we will leverage the framework of §§5–6. Since we are only considering $p \geqslant g$, we may fix $\mu \coloneqq 2$ (see Problem 5.3.4(b)). For each admissible p, let $F^{(p)} \in (\mathbf{Z}/p^2\mathbf{Z})[x]$ be the polynomial defined by

$$F^{(p)} \coloneqq \bar{F} \pmod{p^2}.$$

By construction, $F^{(p)}$ is a lift of $f_p \in \mathbf{F}_p[x]$. Moreover, since p is admissible we have $(f_p)_0 \neq 0$, so everything in §§5–6 may be carried out for $F \coloneqq F^{(p)}$.

Remark 7.2.3. Notice what just happened: the single "global" polynomial $\bar{F} \in \mathbf{Z}[x]$ serves as a lift of f_p for *all* p at the same time. This is one of the key ideas of the average polynomial time approach.

Recall the transition matrix $T_k \in \mathrm{Mat}_d(\mathbf{Z}/p^2\mathbf{Z})$ defined in (6.2.2); to indicate the dependence on p, we will denote it here by $T_k^{(p)}$. According to Proposition 6.3.2 and Proposition 6.3.6, to compute A_{f_p} it suffices to compute the products

$$\begin{aligned} T_1^{(p)} \cdots T_{p-1}^{(p)} &\in \mathrm{Mat}_d(\mathbf{Z}/p^2\mathbf{Z}), & 1 \cdot 2 \cdots (p-1) &\in \mathbf{Z}/p^2\mathbf{Z}, \\ T_{p+1}^{(p)} \cdots T_{2p-1}^{(p)} &\in \mathrm{Mat}_d(\mathbf{Z}/p^2\mathbf{Z}), & (p+1) \cdots (2p-1) &\in \mathbf{Z}/p^2\mathbf{Z}. \end{aligned} \tag{7.2.4}$$

In the present context, we need to compute these products for all admissible $p \leqslant N$ simultaneously. For example, we want to compute

$$\begin{aligned} &T_1^{(3)}T_2^{(3)} && \pmod{3^2}, && T_4^{(3)}T_5^{(3)} && \pmod{3^2}, \\ &T_1^{(5)}T_2^{(5)}T_3^{(5)}T_4^{(5)} && \pmod{5^2}, && T_6^{(5)}T_7^{(5)}T_8^{(5)}T_9^{(5)} && \pmod{5^2}, \\ &T_1^{(7)}T_2^{(7)}T_3^{(7)}T_4^{(7)}T_5^{(7)}T_6^{(7)} && \pmod{7^2}, && T_8^{(7)}T_9^{(7)}T_{10}^{(7)}T_{11}^{(7)}T_{12}^{(7)}T_{13}^{(7)} && \pmod{7^2}, \end{aligned}$$

and so on. Formally, these products look extremely similar to the products appearing in (7.1.2). However, when we attempt to throw the accumulating remainder tree at this problem, we encounter two new difficulties that did not arise in the Wilson prime context.

First, notice that the indices do not line up properly in the $i = 1$ case. For example, the product $T_6T_7T_8T_9$ for $p = 5$ is not a subproduct of the corresponding product $T_8 \cdots T_{13}$ for $p = 7$.

The second and more serious problem is that *the entries of the transition matrices depend on* p! Specifically, the last column of T_k involves the quantity $m = (p-1)/2$, which is different for each p. So we cannot directly build a "value tree" in the same way that we did for the Wilson prime problem.

7.3. Introducing a generic prime We may solve both of the problems mentioned above by employing a technique suggested in [23, §4.4].

Consider the ring

$$\mathbf{Z}[P]/P^2\mathbf{Z}[P],$$

which for brevity we will write as $\mathbf{Z}[P]/P^2$. This may be regarded as a ring of truncated power series in P; a typical element of $\mathbf{Z}[P]/P^2$ has the form $\gamma(P) = \gamma_0 + \gamma_1 P$ for integers γ_0 and γ_1. Multiplication in this ring is given by

$$(\gamma_0 + \gamma_1 P)(\delta_0 + \delta_1 P) = \gamma_0\delta_0 + (\gamma_0\delta_1 + \gamma_1\delta_0)P.$$

We will think of P as a "generic prime" that has not yet been specialised to an actual prime number.

For any $\gamma = \gamma_0 + \gamma_1 P \in \mathbf{Z}[P]/P^2$, it makes sense to substitute a prime number p for P to obtain an element

$$\gamma(p) \coloneqq \gamma_0 + \gamma_1 p \in \mathbf{Z}/p^2\mathbf{Z}.$$

For each p, this operation is clearly a ring homomorphism

$$\mathbf{Z}[P]/P^2 \longrightarrow \mathbf{Z}/p^2\mathbf{Z},$$

i.e., for any $\gamma, \delta \in \mathbf{Z}[P]/P^2$ we have $(\gamma+\delta)(p) = \gamma(p)+\delta(p)$ and $(\gamma\cdot\delta)(p) = \gamma(p)\cdot\delta(p)$ in $\mathbf{Z}/p^2\mathbf{Z}$.

These observations extend to matrices as well. If $\gamma \in \mathrm{Mat}_d(\mathbf{Z}[P]/P^2)$, say $\gamma = \gamma_0 + \gamma_1 P$ for $\gamma_0, \gamma_1 \in \mathrm{Mat}_d(\mathbf{Z})$, then we may substitute an actual prime p for P to obtain $\gamma(p) \coloneqq \gamma_0 + \gamma_1 p \in \mathrm{Mat}_d(\mathbf{Z}/p^2\mathbf{Z})$. Again, for each p this operation is a ring homomorphism $\mathrm{Mat}_d(\mathbf{Z}[P]/P^2) \to \mathrm{Mat}_d(\mathbf{Z}/p^2\mathbf{Z})$.

The idea is now to define a matrix over $\mathbf{Z}[P]/P^2$, that is completely independent of p, but that nevertheless specialises for each p to the desired transition matrix over $\mathbf{Z}/p^2\mathbf{Z}$. For each $i = 0, 1$ and $j \geqslant 1$, define

$$\bar{T}_j^i \in \mathrm{Mat}_d(\mathbf{Z}[P]/P^2)$$

to be the matrix obtained from $2 \cdot T_k$ (see (6.2.2)) by making the substitutions

$$m \mapsto \frac{P-1}{2}, \qquad k \mapsto iP + j,$$

and replacing the coefficients of F by the corresponding coefficients of $\bar{F}$. Explicitly,

$$\bar{T}_j^i(P) = \begin{pmatrix} 0 & \cdots & 0 & 0 & (d(P+1) - 2(iP+j))\bar{F}_d \\ 2(iP+j)\bar{F}_0 & & & & \vdots \\ & \ddots & & & \vdots \\ & & 2(iP+j)\bar{F}_0 & & (2(P+1) - 2(iP+j))\bar{F}_2 \\ & & & 2(iP+j)\bar{F}_0 & (P+1-2(iP+j))\bar{F}_1 \end{pmatrix}.$$

(The purpose of the extra factor of 2 is to make the entries of $\bar{T}_j^i(P)$ integral.)

Problem 7.3.1. Let $i = 0, 1$. Show that for any prime $p \geqslant 3$,

$$\bar{T}_j^i(p) \equiv 2 \cdot T_{ip+j}^{(p)} \pmod{p^2}.$$

Deduce that

$$(\bar{T}_1^i \cdots \bar{T}_{p-1}^i)(p) \equiv 2^{p-1} \cdot T_{ip+1}^{(p)} \cdots T_{ip+p-1}^{(p)} \pmod{p^2}. \tag{7.3.2}$$

We are now in business! Briefly, the algorithm runs as follows. First we build a modulus tree on the squares of the primes, exactly as in §7.1. Next, for each $i = 0, 1$ we build a value tree from the matrices

$$\bar{T}_1^i, \ldots, \bar{T}_N^i \in \mathrm{Mat}_d(\mathbf{Z}[P]/P^2).$$

We then compute an accumulating remainder tree whose leaf nodes contain the matrices

$$\begin{aligned} &\bar{T}_1^i \bar{T}_2^i && \pmod{3^2}, \\ &\bar{T}_1^i \bar{T}_2^i \bar{T}_3^i \bar{T}_4^i && \pmod{5^2}, \\ &\bar{T}_1^i \bar{T}_2^i \bar{T}_3^i \bar{T}_4^i \bar{T}_5^i \bar{T}_6^i && \pmod{7^2}, \end{aligned}$$

and so on, for all $p \leqslant N$. Note that the leaf node corresponding to a given p lies in the ring

$$\mathrm{Mat}_d((\mathbf{Z}/p^2\mathbf{Z})[P]/P^2).$$

At this stage we are finally allowed to "remember" which prime p the generic P is supposed to stand for. Substituting p for P in each leaf node, we obtain the matrices

$$(\bar{T}_1^i \cdots \bar{T}_{p-1}^i)(p) \pmod{p^2}$$

as elements of $\mathrm{Mat}_d(\mathbf{Z}/p^2\mathbf{Z})$. Thanks to (7.3.2), after removing the factors of 2^{p-1} we have found all of the desired matrices in (7.2.4). The scalar quantities in (7.2.4) may be computed by an analogous (much simpler) procedure. Finally, we recover the desired A_{f_p} by applying Proposition 6.3.2 and Proposition 6.3.6 separately for each p.

7.4. Complexity analysis In this section we investigate the complexity of the algorithm sketched above. We will prove the following theorem.

Theorem 7.4.1 (Computing A_{f_p} in average polynomial time)**.** *Let $C/\mathbf{Q}$ be a hyperelliptic curve of genus $g \geqslant 1$ defined by $y^2 = \bar{F}(x)$ for some $\bar{F} \in \mathbf{Z}[x]$. Let $N \geqslant 2$ and assume that*

$$\log N \gg \log \max_i |\bar{F}_i| \quad \text{and} \quad \log N \gg g. \tag{7.4.2}$$

Then we may compute A_{f_p} for all admissible $p \leqslant N$ in time

$$O(g^\omega N \log^3 N).$$

It is possible to drop the hypothesis on $\log \bar{F}_i$ and give a more detailed complexity bound that takes into account the size of the coefficients of $\bar{F}$, but this just complicates the analysis.

As in §7.1, for simplicity we assume that $N = 2^b$.

Compared to the Wilson prime case, the main difficulty in the complexity analysis is keeping track of the sizes of the objects (polynomials, matrices) being manipulated. The following definitions assist in managing this issue.

Definition 7.4.3.

- For $\gamma = \gamma_0 + \gamma_1 P \in \mathbf{Z}[P]/P^2$, define
$$\|\gamma\| \coloneqq |\gamma_0| + |\gamma_1|.$$
- For $\gamma \in \mathrm{Mat}_d(\mathbf{Z}[P]/P^2)$, define
$$\|\gamma\| \coloneqq \max_{1 \leqslant i \leqslant d} \sum_{j=1}^{d} \|\gamma_{i,j}\|.$$
In other words, the norm of a matrix is defined to be the maximum of the L^1 norms of its rows.

The next problem asks you to show that these norms are both *submultiplicative*.

Problem 7.4.4.

a) Let $\gamma, \delta \in \mathbf{Z}[P]/P^2$. Show that $\|\gamma\delta\| \leqslant \|\gamma\|\|\delta\|$.
b) Let $\gamma, \delta \in \mathrm{Mat}_d(\mathbf{Z}[P]/P^2)$. Show that $\|\gamma\delta\| \leqslant \|\gamma\|\|\delta\|$.

Problem 7.4.5. Show that the amount of space (i.e., number of bits) required to represent a matrix $\gamma \in \mathrm{Mat}_d(\mathbf{Z}[P]/P^2)$ is

$$O(d^2 \log \|\gamma\|),$$

and that matrices $\gamma, \delta \in \mathrm{Mat}_d(\mathbf{Z}[P]/P^2)$ may be multiplied in time

$$O\big(d^\omega M_{\mathrm{int}}(\log \|\gamma\| + \log \|\delta\|)\big).$$

We now define two value trees, one for each $i = 0, 1$, to be the product trees on the values

$$\bar{T}_1^i, \ldots, \bar{T}_N^i \in \mathrm{Mat}_d(\mathbf{Z}[P]/P^2).$$

We write $V_{\ell,j}^i \in \mathrm{Mat}_d(\mathbf{Z}[P]/P^2)$ for the matrix in the node (ℓ, j) of the i-th value tree, for $0 \leqslant \ell \leqslant b$ and $1 \leqslant j \leqslant 2^\ell$.

Lemma 7.4.6. *We may compute the value trees for $i = 0, 1$ in time*

$$O(d^\omega N \log^3 N).$$

Problem 7.4.7. Prove Lemma 7.4.6 as follows.

a) Using the hypotheses (7.4.2), show that

$$\|\bar{T}_j^i\| = O(\log N), \qquad 1 \leqslant j \leqslant N.$$

b) Show that the total bit size of the nodes at each level of each value tree is

$$O(d^2 N \log N).$$

c) Finish the proof of Lemma 7.4.6, along the same lines as the proof of Lemma 7.1.7.

Next, we define an accumulating remainder tree for each $i = 0, 1$. In the i-th tree, the value assigned to the node (ℓ, j) is

$$R_{\ell,j}^i \coloneqq V_{\ell,1}^i \cdots V_{\ell,j-1}^i \pmod{m_{\ell,j}} \qquad \in \mathrm{Mat}_d((\mathbf{Z}/m_{\ell,j}\mathbf{Z})[P]/P^2).$$

Lemma 7.4.8. *Assume that the modulus tree and value trees are known. We may compute the accumulating remainder trees for $i = 0, 1$ in time*

$$O(d^\omega N \log^3 N).$$

Problem 7.4.9. Prove Lemma 7.4.8 along the same lines as the proof of Corollary 7.1.12.

Problem 7.4.10. Prove Theorem 7.4.1 as follows.

a) State and prove analogues of Lemma 7.4.6 and Lemma 7.4.8 to handle the scalar quantities in (7.2.4).
b) Assume that all four accumulating remainder trees are known, i.e., the matrix and scalar trees, for both $i = 0$ and $i = 1$. Use (7.3.2) to prove that

for each admissible prime p we may deduce all quantities in (7.2.4) from the leaf nodes of these trees in time

$$O\big((g^2 + \log p)\log^{1+\varepsilon} p\big).$$

Show that we may then use Proposition 6.3.2 and Proposition 6.3.6 to recover A_{f_p} in time

$$O\big((g^3 + \log p)\log^{1+\varepsilon} p\big).$$

c) Show that total cost of (b) over all $p \leqslant N$ is

$$O\big((g^3 + \log N)N\log^{\varepsilon} N\big),$$

and use the hypotheses (7.4.2) to show that this is dominated by

$$O(g^{\omega} N \log^3 N).$$

d) Finish the proof of Theorem 7.4.1.

Problem 7.4.11. (🥽) Implement the algorithm described above to compute A_{f_p} for all admissible $p \leqslant N$ in your FCAS. For what N is it faster than running the $O(p^{1/2+\varepsilon})$ implementation from Problem 6.3.11 separately for each $p \leqslant N$?

Problem 7.4.12. Here is a sketch of another approach that yields an average polynomial time algorithm for computing $L_C(T) \pmod p$ without using the "generic prime" trick. Fix some $\nu = 1, \dots, g$, and consider the vectors

$$W_k^{\nu} \coloneqq (\bar{F}^k_{\nu(2k+1)-1}, \dots, \bar{F}^k_{\nu(2k+1)-d}) \in \mathbf{Z}^d, \qquad k \geqslant 1.$$

For $k = (p-1)/2$, this vector (reduced modulo p) contains the ν-th row of A_{f_p}. Show how to construct a "transition matrix" over $\mathbf{Z}$ that takes W^{ν}_{k-1} to W^{ν}_k, whose entries do not depend on p.

(This "diagonal recurrence" is much closer in spirit to the first published average polynomial time point counting algorithm [22], and the same idea was used in the first reported *implementation* of such an algorithm [25]. The recurrences obtained in this way are horribly complicated, and were abandoned in subsequent implementations such as [13, 24, 26, 52].)

Remark 7.4.13. As mentioned previously in the context of Wilson primes, these algorithms use a huge amount of memory. One systematic method for reducing the memory usage is to replace the accumulating remainder tree by a remainder *forest*, as explained in [25, §4.1].

Remark 7.4.14. The obvious algorithm for multiplying two matrices in $\mathrm{Mat}_d(\mathbf{Z})$ with n-bit entries runs in time $O(d^{\omega} M_{\mathrm{int}}(n))$. This may be improved to

$$O(d^2 M_{\mathrm{int}}(n))$$

if n is large enough compared to d, by reusing the Fourier transforms of the matrix entries [27]. This idea can be used to improve the main complexity bound in Theorem 7.4.1 to

$$O(g^2 N \log^3 N). \tag{7.4.15}$$

This is the best complexity bound I know of for the problem of computing A_{f_p} for all $p \leqslant N$. The dependence on g is optimal, as the output (the list of $A_{\mathbf{F}_p}$ matrices) has total bit size $O(g^2 N)$. The bound (7.4.15) improves on the main theorem of [26] by a factor of $O(g)$; the source of the improvement is that we have in effect replaced the "translated curves" trick (see Problem 5.3.7) by p-adic interpolation (Proposition 6.3.6). I am not aware of any serious attempts at implementing this algorithm, and I do not know how it would perform in practice compared to [26].

8. A modulo p^λ trace formula

So far, all of the point counting algorithms developed in this course have relied on Theorem 4.2.7, which yields information about point counts modulo p. The main result of this section is Theorem 8.4.1, which gives a congruence for these point counts modulo a *power* of p. Since we know *a priori* upper bounds for the point counts, this will enable us to obtain the point counts exactly in $\mathbf{Z}$.

Remark 8.0.1. Theorem 8.4.1 may be viewed as a specialisation of the trace formula [23, Theorem 3.1] to the case of hyperelliptic curves over a prime field. The proofs in both cases are entirely elementary, in the sense that no cohomology is involved.

Remark 8.0.2. The approach taken in this section is not the only way to gain information above and beyond Theorem 4.2.7. One could also use ℓ-adic methods and/or generic group methods to supplement the modulo p data; see for example the last paragraph of [7, §8]. We will not discuss this further in this course.

8.1. A motivating example Recall from (1.5.1) that if $\beta \in \mathbf{F}_p$, then $\beta^{(p-1)/2} \equiv \chi_p(\beta) \pmod{p}$. What happens if we step back and look at things modulo p^2?

Let $\beta \in \mathbf{Z}/p^2\mathbf{Z}$, and let $\overline{\beta} \in \mathbf{F}_p$ be its reduction modulo p. Then

$$\beta^{(p-1)/2} \equiv \chi_p(\overline{\beta}) + cp \pmod{p^2} \tag{8.1.1}$$

for some integer c. For the purposes of a modulo p trace formula such as Theorem 4.2.7, the cp term is irrelevant. But if we want to construct a "modulo p^2 trace formula", the cp term is major nuisance. What we want is an analogue of the expression $\beta^{(p-1)/2}$ that yields $\chi_p(\overline{\beta}) \pmod{p^2}$, *without* the garbage cp term.

We can achieve this by playing some algebraic games with (8.1.1). Let $\beta \in \mathbf{Z}/p^2\mathbf{Z}$ and consider the expression $\beta^{3(p-1)/2}$. From (8.1.1) we have

$$\beta^{3(p-1)/2} \equiv (\chi_p(\overline{\beta}) + cp)^3 \equiv \chi_p(\overline{\beta})^3 + 3\chi_p(\overline{\beta})^2 cp \pmod{p^2}.$$

We now distinguish two cases. First, if $\chi_p(\overline{\beta}) = \pm 1$, then $\chi_p(\overline{\beta})^2 = 1$ and we get

$$\beta^{3(p-1)/2} \equiv \chi_p(\overline{\beta}) + 3cp \pmod{p^2}. \tag{8.1.2}$$

We can now *eliminate* the unwanted cp term by taking a suitable linear combination of (8.1.1) and (8.1.2):

$$\frac{1}{2}\left(3\beta^{(p-1)/2} - \beta^{3(p-1)/2}\right) \equiv \chi_p(\overline{\beta}) \pmod{p^2}. \tag{8.1.3}$$

What about the case $\chi_p(\overline{\beta}) = 0$? In this case $\overline{\beta} = 0$, so $p \mid \beta$. This implies that $\beta^{(p-1)/2} \equiv 0 \pmod{p^2}$, provided that $p \geqslant 5$. Therefore, the formula (8.1.3) holds for all β, except possibly when $p = 3$.

Equation (8.1.3) suggests that (for $p \geqslant 5$) the expression

$$\varphi(\beta^{(p-1)/2}), \qquad \text{where } \varphi(t) \coloneqq \frac{1}{2}(3t - t^3), \tag{8.1.4}$$

plays the role of a "quadratic residue indicator function modulo p^2". In the next subsection we will explore how to generalise this construction to obtain an analogous indicator function modulo p^λ, and subsequently a "modulo p^λ trace formula", for any desired $\lambda \geqslant 1$.

Problem 8.1.5. As mentioned above, (8.1.3) only works for $p \geqslant 5$. Find a polynomial $\varphi \in \mathbf{Q}[t]$ such that $\varphi(\beta^{(p-1)/2}) \equiv \chi_p(\overline{\beta}) \pmod{p^2}$ for all $\beta \in \mathbf{Z}/p^2\mathbf{Z}$ and all primes $p \geqslant 3$.

8.2. The indicator function

Lemma 8.2.1. *Let $\lambda \geqslant 1$ be an integer. There exists a unique polynomial $\varphi_\lambda \in \mathbf{Q}[t]$ of degree at most $2\lambda - 1$ such that*

$$\varphi_\lambda(t) \equiv \begin{cases} 1 & \pmod{(1-t)^\lambda}, \\ -1 & \pmod{(1+t)^\lambda}. \end{cases} \tag{8.2.2}$$

The polynomial φ_λ is odd and its coefficients lie in $\mathbf{Z}[\frac{1}{2}]$.

For example, when $\lambda = 2$ we have $\varphi_\lambda(t) = \frac{1}{2}(3t - t^3)$, which is exactly the polynomial that showed up in (8.1.4).

Note that the notation $\mathbf{Z}[\frac{1}{2}]$ means the ring generated by $\mathbf{Z}$ and $\frac{1}{2}$, i.e., the set of all rational numbers whose denominators are powers of two. "Odd" means that the polynomial has only terms of odd degree.

The existence (and uniqueness) of $\varphi_\lambda \in \mathbf{Q}[t]$ of degree at most $2\lambda - 1$ satisfying (8.2.2) follows immediately from the Chinese Remainder Theorem for $\mathbf{Q}[t]$. The oddness (oddity?) of φ_λ is also easy to deduce from (8.2.2). Proving that the coefficients of φ_λ lie in $\mathbf{Z}[\frac{1}{2}]$ requires slightly more work, and is discussed in the following problem.

Problem 8.2.3. Prove Lemma 8.2.1 as follows.

a) Show that $1 - 2^{-\lambda}(1-t)^\lambda$ is divisible by $1 + t$ in $\mathbf{Z}[\frac{1}{2}][t]$. Let $\eta \in \mathbf{Z}[\frac{1}{2}][t]$ be the quotient.
b) Let $\psi \in \mathbf{Q}[t]$ be the unique polynomial of degree at most $\lambda - 1$ such that $\psi \equiv \eta^\lambda \pmod{(1-t)^\lambda}$. Show that $\psi \in \mathbf{Z}[\frac{1}{2}][t]$ and that $(1+t)^\lambda \psi \equiv 1 \pmod{(1-t)^\lambda}$.
c) Show that $\varphi_\lambda(t) = \psi(t)(1+t)^\lambda - \psi(-t)(1-t)^\lambda$, and hence prove Lemma 8.2.1.

Problem 8.2.4. Establish the following explicit formula for $\varphi_\lambda(t)$:

$$\varphi_\lambda(t) = \sum_{j=0}^{\lambda-1} \frac{1}{2^{\lambda+j}} \binom{\lambda+j-1}{j} \left((1-t)^j (1+t)^\lambda - (1+t)^j (1-t)^\lambda \right).$$

Now we can use $\varphi_\lambda(t)$ to construct the desired quadratic residue indicator function modulo p^λ:

Corollary 8.2.5. *Let $p \geqslant 2\lambda+1$ be a prime. Let $\beta \in \mathbf{Z}/p^\lambda\mathbf{Z}$ and let $\overline{\beta}$ be its reduction modulo p. Then*

$$\varphi_\lambda(\beta^{(p-1)/2}) \equiv \chi_p(\overline{\beta}) \pmod{p^\lambda}.$$

For example, for $\lambda = 2$, Corollary 8.2.5 says exactly that the expression given in (8.1.4) is an indicator function modulo p^2, provided that $p \geqslant 5$.

Problem 8.2.6. Prove Corollary 8.2.5 as follows.

a) Suppose that $\chi_p(\overline{\beta}) = 1$. From (8.2.2) we may write

$$\varphi_\lambda(t) = 1 + \psi(t)(1-t)^\lambda.$$

for some $\psi \in \mathbf{Z}[\frac{1}{2}][t]$. Use this to show that

$$\varphi_\lambda(\beta^{(p-1)/2}) \equiv 1 \pmod{p^\lambda}.$$

b) Deal with the case $\chi_p(\overline{\beta}) = -1$ along similar lines to (a).

c) Finish the proof by handling the remaining case $\chi_p(\overline{\beta}) = 0$. Where do you use the hypothesis that $p \geqslant 2\lambda+1$?

Problem 8.2.7. Write out explicitly the modulo p^3 analogue of (8.1.3) (which holds for $p \geqslant 7$).

Problem 8.2.8. (🎮) Use your FCAS to compute the polynomial $\varphi_\lambda(t)$ for a few small values of λ, and check numerically that Corollary 8.2.5 holds for a few randomly chosen primes p and $\beta \in \mathbf{Z}/p^\lambda\mathbf{Z}$.

Problem 8.2.9. Let $\lambda \geqslant 1$. Find a polynomial $\varphi \in \mathbf{Q}[t]$ for which an analogue of Corollary 8.2.5 holds for all $p \geqslant 3$. You will need to allow the degree of φ to be bigger than $2\lambda - 1$. What is the smallest degree you can get away with?

(The case $\lambda = 2$ was discussed in Problem 8.1.5. This problem is related to the "fudge factor" denoted by τ in [23, Theorem 3.1].)

8.3. Counting points with coordinates in $\mathbf{F}_p$ We now explore how to use the indicator function to write down a modulo p^λ trace formula. As a warm-up, we give a formula for counting points defined over $\mathbf{F}_p$. Recall that $\widetilde{C}$ is the variety obtained from C by removing the points with $x = 0, \infty$ (see Definition 4.1.3).

Proposition 8.3.1. *Let $\lambda \geqslant 1$ and let $p \geqslant 2\lambda+1$ be a prime. Let $C/\mathbf{F}_p$ be a hyperelliptic curve of genus $g \geqslant 1$ defined by $y^2 = f(x)$ for some $f \in \mathbf{F}_p[x]$. Let*

$$F = F_0 + F_1 x + \cdots + F_{2g+2} x^{2g+2} \in (\mathbf{Z}/p^\lambda\mathbf{Z})[x]$$

be any lift of f*, i.e., so that* $F \equiv f \pmod{p^\lambda}$*. Let*

$$H \coloneqq F^{(p-1)/2} \in (\mathbf{Z}/p^\lambda\mathbf{Z})[x].$$

Then

$$\#\widetilde{C}(\mathbf{F}_p) \equiv (p-1)\left(1 + \sum_{\ell=0}^{\lambda-1} \varphi_{\lambda,2\ell+1} \sum_{j=0}^{(2\ell+1)(g+1)} H_{j(p-1)}^{2\ell+1}\right) \pmod{p^\lambda}, \tag{8.3.2}$$

where $\varphi_{\lambda,i}$ *denotes the coefficient of* t^i *in* $\varphi_\lambda(t)$*.*

There are three key differences between (8.3.2) and the modulo p version (4.1.2) that deserve to be highlighted.

First, whereas (4.1.2) is expressed entirely in terms of the defining polynomial $f \in \mathbf{F}_p[x]$, the modulo p^λ version (8.3.2) involves an arbitrary *lift* of that polynomial to $F \in (\mathbf{Z}/p^\lambda\mathbf{Z})[x]$.

Second, whereas (4.1.2) involves only coefficients from a single polynomial $h = f^{(p-1)/2}$, the formula (8.3.2) involves coefficients from several powers of $H = F^{(p-1)/2}$, namely H, $H^3, \ldots, H^{2\lambda-1}$. For example, in the case $g = 3$, $\deg F = 8$, $\lambda = 3$, the right hand side of (8.3.2) involves:

- the coefficients of $1, x^{p-1}, \ldots, x^{4(p-1)}$ in $H = F^{(p-1)/2}$,
- the coefficients of $1, x^{p-1}, \ldots, x^{12(p-1)}$ in $H^3 = F^{3(p-1)/2}$,
- the coefficients of $1, x^{p-1}, \ldots, x^{20(p-1)}$ in $H^5 = F^{5(p-1)/2}$.

Third, whereas (4.1.2) gives a congruence for $\#C(\mathbf{F}_p)$, the new formula (8.3.2) gives a congruence for $\#\widetilde{C}(\mathbf{F}_p)$. This is not unrelated to the presence of the "extra terms" $j = 0$ and $j = (2\ell+1)(g+1)$ that have no analogue in (4.1.2). Actually, it *is* possible to write down an explicit formula for $\#C(\mathbf{F}_p) \pmod{p^\lambda}$, but unfortunately it is not as neat as (4.1.2). For further discussion, see Problem 8.4.9.

Let us turn now to the proof of Proposition 8.3.1. We will require a few facts about the $(p-1)$-th roots of unity in $\mathbf{Z}/p^\lambda\mathbf{Z}$. Define

$$\Sigma_\lambda(\mathbf{F}_p^*) \coloneqq \left\{\alpha \in \mathbf{Z}/p^\lambda\mathbf{Z} : \alpha^{p-1} = 1\right\}.$$

Proposition 8.3.3. *The reduction map* $\mathbf{Z}/p^\lambda\mathbf{Z} \to \mathbf{F}_p$ *induces a bijection between* $\Sigma_\lambda(\mathbf{F}_p^*)$ *and* $\mathbf{F}_p^*$*.*

The element $\alpha \in \Sigma_\lambda(\mathbf{F}_p^*)$ corresponding to $\beta \in \mathbf{F}_p^*$ is called the *Teichmüller representative* (modulo p^λ) for β. For example, when $p = 5$, the Teichmüller representatives modulo p^3 for the elements $1, 2, 3, 4 \in \mathbf{F}_5^*$ are respectively $1, 57, 68, 124 \in \mathbf{Z}/5^3\mathbf{Z}$. These are exactly the fourth roots of unity in $\mathbf{Z}/5^3\mathbf{Z}$.

Problem 8.3.4.

a) Prove Proposition 8.3.3. (Hint: if $\beta \in \mathbf{F}_p^*$, lift β to some $\gamma \in \mathbf{Z}/p^\lambda\mathbf{Z}$ and consider $\alpha = \gamma^{p^{\lambda-1}}$.)

b) For any integer $i \geqslant 0$, show that

$$\sum_{\alpha \in \Sigma_\lambda(\mathbf{F}_p^*)} \alpha^i = \begin{cases} p-1 & \text{if } i \text{ is divisible by } p-1, \\ 0 & \text{otherwise.} \end{cases}$$

c) Prove Proposition 8.3.1, by using part (b) and Corollary 8.2.5 to evaluate the sum

$$\sum_{\alpha \in \Sigma_\lambda(\mathbf{F}_p^*)} \Big(\varphi_\lambda\big(F(\alpha)^{(p-1)/2}\big) + 1 \Big) \pmod{p^\lambda}$$

in two different ways.

Problem 8.3.5. (🎮) Check Proposition 8.3.1 numerically in your FCAS, i.e., for some small p, g, λ and randomly chosen $f \in \mathbf{F}_p[x]$, choose a suitable lift $F \in (\mathbf{Z}/p^\lambda\mathbf{Z})[x]$ and check that (8.3.2) correctly counts the number of points in $\widetilde{C}(\mathbf{F}_p)$ modulo p^λ.

8.4. Counting points with coordinates in $\mathbf{F}_{p^r}$ Proposition 8.3.1 is not very interesting by itself, because we already know how to compute $\#C(\mathbf{F}_p)$ by computing it modulo p, at least if p is not too small (see Problem 4.1.6). What we really want is a modulo p^λ trace formula for counting points with coordinates in $\mathbf{F}_{p^r}$, i.e., a simultaneous generalisation of Theorem 4.2.7 and Proposition 8.3.1. The following result does the job.

Theorem 8.4.1 (Modulo p^λ trace formula)**.** *Let $\lambda \geqslant 1$ and let $p \geqslant 2\lambda + 1$ be a prime. Let $C/\mathbf{F}_p$ be a hyperelliptic curve of genus $g \geqslant 1$ defined by $y^2 = f(x)$ for some $f \in \mathbf{F}_p[x]$. Let*

$$F = F_0 + F_1 x + \cdots + F_{2g+2} x^{2g+2} \in (\mathbf{Z}/p^\lambda\mathbf{Z})[x]$$

be any lift of f, i.e., so that $F \equiv f \pmod{p^\lambda}$, and let

$$H \coloneqq F^{(p-1)/2} \in (\mathbf{Z}/p^\lambda\mathbf{Z})[x].$$

For $\ell = 0, \dots, \lambda - 1$, let

$$\widetilde{A}_{F,\ell} \in \mathrm{Mat}_{(2\ell+1)(g+1)+1}(\mathbf{Z}/p^\lambda\mathbf{Z})$$

be the matrix defined by

$$(\widetilde{A}_{F,\ell})_{\nu,u} \coloneqq (H^{2\ell+1})_{\nu p - u}, \qquad 0 \leqslant u, \nu \leqslant (2\ell+1)(g+1).$$

Then for any $r \geqslant 1$,

$$\#\widetilde{C}(\mathbf{F}_{p^r}) \equiv (p^r - 1)\left(1 + \sum_{\ell=0}^{\lambda-1} \varphi_{\lambda,2\ell+1} \operatorname{tr}(\widetilde{A}_{F,\ell}^r)\right) \pmod{p^\lambda}. \tag{8.4.2}$$

The main new feature in Theorem 8.4.1 is that instead of one matrix $\widetilde{A}_f$, we have to deal with λ matrices $\widetilde{A}_{F,\ell}$ for $\ell = 0, \dots, \lambda - 1$, whose size increases with ℓ.

Problem 8.4.3. Check that Theorem 8.4.1 specialises to Proposition 8.3.1 in the case $r = 1$.

To prove Proposition 8.4.1, we need to be able to sum over the "$(p^r - 1)$-th roots of unity modulo p^λ". To make sense of this, recall that $\mathbf{F}_{p^r}$ may be represented as

$$\mathbf{F}_{p^r} = \mathbf{F}_p[\theta]/z(\theta)$$

for some irreducible monic polynomial $z \in \mathbf{F}_p[\theta]$ of degree r. Let $Z \in (\mathbf{Z}/p^\lambda\mathbf{Z})[\theta]$ be any monic degree-r lift of z, i.e., so that $Z \equiv z \pmod{p}$, and define

$$W_\lambda(\mathbf{F}_{p^r}) \coloneqq (\mathbf{Z}/p^\lambda\mathbf{Z})[\theta]/Z(\theta).$$

Elements of $W_\lambda(\mathbf{F}_{p^r})$ may be represented by polynomials of degree at most $r-1$ in $(\mathbf{Z}/p^\lambda\mathbf{Z})[\theta]$. The ring $W_\lambda(\mathbf{F}_{p^r})$ plays the analogous role for $\mathbf{F}_{p^r}$ as $\mathbf{Z}/p^\lambda\mathbf{Z}$ does for $\mathbf{F}_p$. (Readers familiar with Witt vectors will recognise this construction.) Now define

$$\Sigma_\lambda(\mathbf{F}_{p^r}^*) \coloneqq \left\{\alpha \in W_\lambda(\mathbf{F}_{p^r}) : \alpha^{p^r-1} = 1\right\}.$$

Proposition 8.4.4. *For each $r \geqslant 1$, the reduction map $W_\lambda(\mathbf{F}_{p^r}) \to \mathbf{F}_{p^r}$ induces a bijection between $\Sigma_\lambda(\mathbf{F}_{p^r}^*)$ and $\mathbf{F}_{p^r}^*$.*

The element of $\Sigma_\lambda(\mathbf{F}_{p^r}^*)$ corresponding to a given $\beta \in \mathbf{F}_{p^r}^*$ is called the Teichmüller representative (modulo p^λ) for β.

Problem 8.4.5. Suppose that we represent $\mathbf{F}_{3^2}$ as $\mathbf{F}_3[\theta]/(\theta^2+1)$, and $W_2(\mathbf{F}_{3^2})$ as $(\mathbf{Z}/3^2\mathbf{Z})[\theta]/(\theta^2+1)$. Find the Teichmüller representatives for all elements of $\mathbf{F}_{3^2}^*$.

Problem 8.4.6.

a) Prove Proposition 8.4.4.

b) For any integer $i \geqslant 0$, show that

$$\sum_{\alpha \in \Sigma_\lambda(\mathbf{F}_{p^r}^*)} \alpha^i = \begin{cases} p^r - 1 & \text{if } i \text{ is divisible by } p^r - 1, \\ 0 & \text{otherwise.} \end{cases}$$

Problem 8.4.7. Prove Theorem 8.4.1 as follows.

a) Let

$$H^{(r)}(x) \coloneqq H(x)H(x^p)\cdots H(x^{p^{r-1}}) \in (\mathbf{Z}/p^\lambda\mathbf{Z})[x].$$

By evaluating the sum

$$\sum_{\alpha \in \Sigma_\lambda(\mathbf{F}_{p^r}^*)} \left(\varphi_\lambda\left(H^{(r)}(\alpha)\right) + 1\right) \pmod{p^\lambda}$$

in two different ways, show that

$$\#\widetilde{C}(\mathbf{F}_{p^r}) \equiv (p^r-1)\left(1 + \sum_{\ell=0}^{\lambda-1} \varphi_{\lambda,2\ell+1} \sum_{j=0}^{(2\ell+1)(g+1)} (H^{(r)})^{2\ell+1}_{j(p^r-1)}\right) \pmod{p^\lambda}.$$

b) Using a similar argument to Problem 4.2.9(b), show that for each $\ell = 0, \ldots, \lambda-1$ we have

$$\sum_{j=0}^{(2\ell+1)(g+1)} (H^{(r)})^{2\ell+1}_{j(p^r-1)} = \operatorname{tr}(\widetilde{A}^r_{F,\ell}).$$

Problem 8.4.8. (🕶) Check Theorem 8.4.1 numerically in your FCAS, i.e., for some small p, g, λ, r, and randomly chosen $f \in \mathbf{F}_p[x]$, take an arbitrary lift

$F \in (\mathbf{Z}/p^\lambda\mathbf{Z})[x]$, evaluate (8.4.2), and check that the result agrees modulo p^λ with the actual number of points (obtained for example by the enumeration method).

Problem 8.4.9. Notation and hypotheses as in Theorem 8.4.1. In this problem we investigate the possibility of an elegant modulo p^λ trace formula for $C(\mathbf{F}_{p^r})$ rather than $\widetilde{C}(\mathbf{F}_{p^r})$.

a) For $\ell = 0, \dots, \lambda - 1$ define matrices

$$A_{F,\ell} \in \mathrm{Mat}_{(2\ell+1)(g+1)-1}(\mathbf{Z}/p^\lambda\mathbf{Z})$$

by the formula

$$(A_{F,\ell})_{\nu,u} \coloneqq (H^{2\ell+1})_{\nu p - u}, \qquad 1 \leqslant u, \nu \leqslant (2\ell+1)(g+1) - 1.$$

State and prove a formula of the type

$$\#C(\mathbf{F}_{p^r}) \equiv (p^r - 1)\left(1 + \sum_{\ell=0}^{\lambda-1} \varphi_{\lambda,2\ell+1} \operatorname{tr}(A_{F,\ell}^r)\right) + (\text{correction terms}) \pmod{p^\lambda},$$

where the "correction terms" involve suitable linear combinations of powers of F_0 and F_{2g+2}.

b) Show that that the correction terms vanish when $r \geqslant \lambda$. In other words, when $r \geqslant \lambda$, we get a stylish trace formula for $C(\mathbf{F}_{p^r})$, just like (4.2.7).

Problem 8.4.10. (🗲) In point counting algorithms based on p-adic cohomology, the strategy is usually to construct a certain matrix over $\mathbf{Q}_p$ (the matrix of a Frobenius operator on some p-adic vector space) whose characteristic polynomial yields the L-polynomial $L_p(T)$ modulo arbitrarily large powers of p. Is there a similar formula that yields $L_p(T) \pmod{p^\lambda}$ in terms of the matrices $\widetilde{A}_{F,\ell}$ appearing in Theorem 8.4.1? (Or in terms of the matrices $A_{F,\ell}$ in Problem 8.4.9?) More generally, is there a "p-adic cohomological" interpretation of the modulo p^λ trace formulas given in this course?

8.5. Algorithms and complexity bounds In this final section of the course, we discuss how to design algorithms that use Theorem 8.4.1 to compute not just $L_C(T) \pmod p$, but the entire L-polynomial $L_C(T) \in \mathbf{Z}[T]$. We will not give many details, but rather leave these to you as challenging exercises.

Our first task is to determine a suitable value of λ.

Lemma 8.5.1. *Let λ be the smallest integer such that*

$$\lambda > \frac{g}{2} + \log_p(4g).$$

Then $L_C(T) \in \mathbf{Z}[T]$ may be recovered from the values

$$\#C(\mathbf{F}_p), \dots, \#C(\mathbf{F}_{p^g}) \pmod{p^\lambda}. \tag{8.5.2}$$

Problem 8.5.3. Prove Lemma 8.5.1, via (3.2.2) and Problem 3.3.5.

In principle, it is now straightforward to compute $L_C(T) \in \mathbf{Z}[T]$. Given as input $f \in \mathbf{F}_p[x]$, we lift to some $F \in (\mathbf{Z}/p^\lambda\mathbf{Z})[x]$ where λ is chosen as in Lemma 8.5.1.

Notice in particular that $\lambda = O(g)$. According to Theorem 8.4.1, we can recover the quantities in (8.5.2) from the matrices

$$\widetilde{A}_{F,0}, \ldots, \widetilde{A}_{F,\lambda-1}. \tag{8.5.4}$$

(We tacitly assume here that $p \geqslant 2\lambda + 1$. To handle smaller p, see Problem 8.2.9.)

Problem 8.5.5. Estimate the complexity of computing $L_C(T) \in \mathbf{Z}[T]$ from knowledge of the matrices in (8.5.4). In other words, state and prove a suitable analogue of Corollary 4.4.3. You should get a complexity bound of the form

$$O(g^c \log^{1+\varepsilon} p)$$

for some constant $c > 0$.

Computing the matrices in (8.5.4) amounts to computing certain selected coefficients of $H, H^3, \ldots, H^{2\lambda-1}$ modulo p^λ, where $H = F^{(p-1)/2}$, or in other words, the polynomials

$$F^{(p-1)/2}, F^{3(p-1)/2}, \ldots, F^{(2\lambda-1)(p-1)/2}. \tag{8.5.6}$$

This task may be carried out by adapting any of the various strategies given earlier in the course for computing the coefficients of $h = f^{(p-1)/2}$. Let us examine each of these strategies in turn.

Problem 8.5.7 (Computing $A_{F,\ell}$ via naive expansion)**.** State and prove an analogue of Theorem 4.4.1 for computing the matrices in (8.5.4), i.e., by simply expanding out the polynomials in (8.5.6). You should get a complexity bound of the form

$$O(g^c p \log^2(gp))$$

for some constant $c > 0$.

Next we consider the recurrence strategy from §5. For each power $H^{2\ell+1}$, we may set up a suitable recurrence, and define a sequence of approximations for the coefficients of $H^{2\ell+1}$, analogous to the sequence $\widetilde{H}_k$ from §5.2. In order to deal with the "division-by-p" problem, we need to lift F further to $(\mathbf{Z}/p^\mu\mathbf{Z})[x]$ for some $\mu > \lambda$.

Problem 8.5.8.

a) State and prove an analogue of Corollary 5.3.1, i.e., determine how big μ must be to ensure that our approximations for the coefficients of $H^{2\ell+1}$ are correct modulo p^λ.

b) State and prove an analogue of Problem 5.3.4(b), i.e., show that for p large enough relative to g, to correctly compute the matrices in (8.5.4) it suffices to take $\mu := \lambda + 1$. (In particular, we may take $\mu = O(g)$.)

Problem 8.5.9 (Computing $A_{F,\ell}$ via recurrences)**.** State and prove an analogue of Theorem 5.3.3 for computing the matrices in (8.5.4). You should get a complexity bound of the form

$$O(g^c p \log^{1+\varepsilon} p)$$

for some constant $c > 0$.

Next in line is the square-root algorithm.

Problem 8.5.10. Generalise Proposition 6.3.6 (the interpolation trick) from the $\mu = 2$ case to arbitrary values of μ. More precisely, show that once we have computed (6.3.3) and (6.3.4) for $i = 0, \dots, \mu - 1$, then we may quickly deduce (6.3.3) and (6.3.4) for any desired value of i. (Careful: you may need to impose a lower bound on p to avoid running into difficulties during the interpolation.)

Problem 8.5.11 (Computing $A_{F,\ell}$ in square-root time)**.** State and prove an analogue of Theorem 6.3.8 for computing the matrices in (8.5.4). Taking into account Remark 6.2.6, you should get a complexity bound of the form

$$O(g^c p^{1/2} \log^2 p)$$

for some constant $c > 0$. (It should be possible to generalise Problem 6.3.12 from two subproducts to μ subproducts, in order to squeeze out another factor of $g^{1/2}$.)

Remark 8.5.12. The complexity bound attained in the previous problem is of a similar nature to the main result of [21], which is based on p-adic cohomology. As mentioned in Remark 6.3.13, it would be interesting to compare the two approaches, both from a theoretical and a practical point of view.

Finally, we consider the average polynomial time algorithm. After replacing the ring $\mathbf{Z}[P]/P^2$ with $\mathbf{Z}[P]/P^\mu$, everything should run much the same way as before.

Problem 8.5.13 (Computing $A_{F,\ell}$ in average polynomial time)**.** State and prove an analogue of Theorem 7.4.1 for computing the matrices in (8.5.4), for all admissible $p \leqslant N$, for some suitably adjusted notion of "admissible". You should get a complexity bound of the form

$$O(g^c N \log^3 N)$$

for some constant $c > 0$.

Problem 8.5.14. (🗲) In all of the algorithms suggested above, the powers H, $H^3, \dots, H^{2\lambda-1}$ are handled separately. Can the exponents of g in the complexity bounds be be further improved by taking advantage of some redundancy between these powers?

To the best of my knowledge, no-one has yet made a serious effort to implement *any* average polynomial point counting algorithm that obtains point counts modulo p^λ for any $\lambda > 1$.

Problem 8.5.15. (🎮) Implement all the algorithms discussed in this section!

References

[1] T. G. Abbott, K. S. Kedlaya, and D. Roe, *Bounding Picard numbers of surfaces using* p*-adic cohomology*, Arithmetics, geometry, and coding theory (AGCT 2005), 2010, pp. 125–159, Sémin. Congr., vol. 21. MR2856564 ←6

[2] J. D. Achter and E. W. Howe, *Hasse-Witt and Cartier-Manin matrices: a warning and a request*, Arithmetic geometry: computation and applications, [2019] ©2019, pp. 1–18, Contemp. Math., vol. 722. MR3896846 ←19

[3] J. Alman and V. Vassilevska Williams, *A refined laser method and faster matrix multiplication*, Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA), 2021, pp. 522–539. MR4262465 ←11

[4] V. Andrejić, A. Bostan, and M. Tatarevic, *Improved algorithms for left factorial residues*, Inform. Process. Lett. **167** (2021), Paper No. 106078, 3. MR4195830 ←32, 40

[5] D. J. Bernstein, *Fast multiplication and its applications*, Algorithmic number theory: lattices, number fields, curves and cryptography, 2008, pp. 325–384, Math. Sci. Res. Inst. Publ., vol. 44. MR2467550 (2010a:68186) ←6

[6] A. R. Booker, S. Hathi, M. J. Mossinghoff, and T. S. Trudgian, *Wolstenholme and Vandiver primes*, Ramanujan J. **58** (2022), no. 3, 913–941. MR4437433 ←30

[7] 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. MR2299425 (2008a:11156) ←24, 28, 30, 32, 35, 47

[8] R. P. Brent and P. Zimmermann, *Modern Computer Arithmetic*, Cambridge Monographs on Applied and Computational Mathematics, vol. 18, Cambridge University Press, Cambridge, 2011. MR2760886 ←6

[9] D. V. Chudnovsky and G. V. Chudnovsky, *Approximations and complex multiplication according to Ramanujan*, Ramanujan revisited (Urbana-Champaign, Ill., 1987), 1988, pp. 375–472. MR938975 (89f:11099) ←32

[10] H. Cohen, G. Frey, R. Avanzi, C. Doche, T. Lange, K. Nguyen, and F. Vercauteren (eds.), *Handbook of Elliptic and Hyperelliptic Curve Cryptography*, Discrete Mathematics and its Applications (Boca Raton), Chapman & Hall/CRC, Boca Raton, FL, 2006. MR2162716 (2007f:14020) ←2

[11] E. Costa, R. Gerbicz, and D. Harvey, *A search for Wilson primes*, Math. Comp. **83** (2014), no. 290, 3071–3091. MR3246824 ←28, 36, 39, 40

[12] E. Costa and D. Harvey, *Faster deterministic integer factorization*, Math. Comp. **83** (2014), no. 285, 339–345. MR3120593 ←30

[13] E. Costa, D. Harvey, and A. V. Sutherland, *Counting points on smooth plane quartics*, 2022. To appear in proceedings of ANTS XV. ←46

[14] R. Crandall, K. Dilcher, and C. Pomerance, *A search for Wieferich and Wilson primes*, Math. Comp. **66** (1997), no. 217, 433–449. MR1372002 (97c:11004) ←30, 40

[15] N. D. Elkies, *Elliptic and modular curves over finite fields and related computational issues*, Computational perspectives on number theory (Chicago, IL, 1995), 1998, pp. 21–76, AMS/IP Stud. Adv. Math., vol. 7. MR1486831 (99a:11078) ←2

[16] M. Farach-Colton and M. Tsai, *On the complexity of computing prime tables*, Algorithms and computation, 2015, pp. 677–688, Lecture Notes in Comput. Sci., vol. 9472. MR3489524 ←11

[17] 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. MR2982436 ←2

[18] P. Flajolet and B. Salvy, *The SIGSAM challenges: Symbolic asymptotics in practice*, SIGSAM Bull. **31** (1997dec), no. 4, 36–47. ←28

[19] R. Gerbicz, 2011. https://www.mersenneforum.org/showthread.php?p=270602. ←36

[20] T. Granlund, *The GNU Multiple Precision Arithmetic Library*. http://gmplib.org/. ←7

[21] D. Harvey, *Kedlaya's algorithm in larger characteristic*, Int. Math. Res. Not. IMRN **22** (2007), Art. ID rnm095, 29. MR2376210 (2009d:11096) ←35, 55

[22] D. Harvey, *Counting points on hyperelliptic curves in average polynomial time*, Ann. of Math. (2) **179** (2014), no. 2, 783–803. MR3152945 ←46

[23] D. Harvey, *Computing zeta functions of arithmetic schemes*, Proc. Lond. Math. Soc. (3) **111** (2015), no. 6, 1379–1401. MR3447797 ←1, 5, 6, 42, 47, 49

[24] D. Harvey, M. Massierer, and A. V. Sutherland, *Computing* L*-series of geometrically hyperelliptic curves of genus three*, LMS J. Comput. Math. **19** (2016), no. suppl. A, 220–234. MR3540957 ←46

[25] D. Harvey and A. V. Sutherland, *Computing Hasse-Witt matrices of hyperelliptic curves in average polynomial time*, LMS J. Comput. Math. **17** (2014), no. suppl. A, 257–273. MR3240808 ←46

[26] D. Harvey and A. V. Sutherland, *Computing Hasse-Witt matrices of hyperelliptic curves in average polynomial time, II*, Frobenius distributions: Lang-Trotter and Sato-Tate conjectures, 2016, pp. 127–147, Contemp. Math., vol. 663. MR3502941 ←28, 46, 47

[27] D. Harvey and J. van der Hoeven, *On the complexity of integer matrix multiplication*, J. Symbolic Comput. **89** (2018), 1–8. MR3804803 ←46
[28] D. Harvey and J. van der Hoeven, *Integer multiplication in time* $O(n \log n)$, Ann. of Math. (2) **193** (2021), no. 2, 563–617. MR4224716 ←7
[29] D. Harvey and J. van der Hoeven, *Polynomial multiplication over finite fields in time* $O(n \log n)$, J. ACM **69** (2022mar), no. 2. ←9
[30] M. Hindry and J. H. Silverman, *Diophantine Geometry*, Graduate Texts in Mathematics, vol. 201, Springer-Verlag, New York, 2000. An introduction. MR1745599 ←41
[31] K. S. Kedlaya, *Counting points on hyperelliptic curves using Monsky-Washnitzer cohomology*, J. Ramanujan Math. Soc. **16** (2001), no. 4, 323–338. MR1877805 (2002m:14019) ←2, 5, 25
[32] K. S. Kedlaya, *Quantum computation of zeta functions of curves*, Comput. Complexity **15** (2006), no. 1, 1–19. MR2226067 ←2
[33] K. S. Kedlaya and A. V. Sutherland, *Computing* L*-series of hyperelliptic curves*, Algorithmic number theory, 2008, pp. 312–326, Lecture Notes in Comput. Sci., vol. 5011. MR2467855 (2010d:11070) ←14
[34] D. E. Knuth, *The Art of Computer Programming. Vol. 2*, Addison-Wesley, Reading, MA, 1998. Seminumerical algorithms, Third edition [of MR0286318]. MR3077153 ←6
[35] M. Kyng, *Computing zeta functions of algebraic curves using Harvey's trace formula*, 2022. Preprint at https://arxiv.org/abs/2203.02070. To appear in proceedings of ANTS XV. ←5
[36] A. G. B. Lauder, *Counting solutions to equations in many variables over finite fields*, Found. Comput. Math. **4** (2004), no. 3, 221–267. MR2078663 (2005f:14048) ←6
[37] D. Lorenzini, *An Invitation to Arithmetic Geometry*, Graduate Studies in Mathematics, vol. 9, American Mathematical Society, Providence, RI, 1996. MR1376367 (97e:14035) ←15
[38] J. I. Manin, *The Hasse–Witt matrix of an algebraic curve*, Fifteen papers on algebra, 1965, pp. 245–264, American Mathematical Society Translations: Series 2, vol. 45. Translated by J.W.S. Cassels. ←21
[39] R. J. McIntosh and E. L. Roettger, *A search for Fibonacci-Wieferich and Wolstenholme primes*, Math. Comp. **76** (2007), no. 260, 2087–2094. MR2336284 (2008k:11008) ←30
[40] H. L. Montgomery and R. C. Vaughan, *Multiplicative Number Theory. I. Classical Theory*, Cambridge Studies in Advanced Mathematics, vol. 97, Cambridge University Press, Cambridge, 2007. MR2378655 ←4
[41] V. Neiger and C. Pernet, *Deterministic computation of the characteristic polynomial in the time of matrix multiplication*, J. Complexity **67** (2021), Paper No. 101572, 35. MR4311525 ←11
[42] C. H. Papadimitriou, *Computational Complexity*, Addison-Wesley Publishing Company, Reading, MA, 1994. MR1251285 (95f:68082) ←4
[43] J. Pila, *Frobenius maps of abelian varieties and finding roots of unity in finite fields*, Math. Comp. **55** (1990), no. 192, 745–763. MR1035941 (91a:11071) ←2
[44] J. M. Pollard, *Theorems on factorization and primality testing*, Proc. Cambridge Philos. Soc. **76** (1974), 521–528. MR0354514 (50 #6992) ←30
[45] A. Schönhage, A. F. W. Grotefeld, and E. Vetter, *Fast algorithms*, Bibliographisches Institut, Mannheim, 1994. A multitape Turing machine implementation. MR1290996 ←11
[46] R. Schoof, *Elliptic curves over finite fields and the computation of square roots mod* p, Math. Comp. **44** (1985), no. 170, 483–494. MR777280 (86e:11122) ←2
[47] H. Stichtenoth, *Algebraic Function Fields and Codes*, Second, Graduate Texts in Mathematics, vol. 254, Springer-Verlag, Berlin, 2009. MR2464941 (2010d:14034) ←2, 15
[48] V. Strassen, *Gaussian elimination is not optimal*, Numer. Math. **13** (1969), 354–356. MR0248973 (40 #2223) ←11
[49] V. Strassen, *Einige Resultate über Berechnungskomplexität*, Jber. Deutsch. Math.-Verein. **78** (1976/77), no. 1, 1–8. MR0438807 (55 #11713) ←29
[50] A. V. Sutherland, *Smalljac software library version 4.1.3*. https://math.mit.edu/~drew/smalljac_v4.1.3.tar. ←14
[51] A. V. Sutherland, *Order computations in generic groups*, Ph.D. Thesis, 2007. ←2
[52] A. V. Sutherland, *Counting points on superelliptic curves in average polynomial time*, ANTS XIV—Proceedings of the Fourteenth Algorithmic Number Theory Symposium, 2020, pp. 403–422, Open Book Ser., vol. 4. MR4235126 ←46
[53] J. Tuitman, *Counting points on curves using a map to* $\mathbf{P}^1$*, II*, Finite Fields Appl. **45** (2017), 301–322. MR3631366 ←2, 5

[54] J. von zur Gathen and J. Gerhard, *Modern Computer Algebra*, 3rd ed., Cambridge University Press, Cambridge, 2013. MR3087522 ←6

[55] J. von zur Gathen and V. Shoup, *Computing Frobenius maps and factoring polynomials*, Comput. Complexity **2** (1992), no. 3, 187–224. MR1220071 (94d:12011) ←30

School of Mathematics and Statistics, UNSW Sydney, NSW 2052, Australia
Email address: d.harvey@unsw.edu.au