Understanding binary-Goppa decoding

Daniel J. Bernstein

University of Illinois at Chicago, USA
Ruhr University Bochum, Germany
Academia Sinica, Taiwan

Abstract. This paper reviews, from bottom to top, a polynomial-time algorithm to correct t errors in classical binary Goppa codes defined by squarefree degree-t polynomials. The proof is factored through a proof of a simple Reed–Solomon decoder, and the algorithm is simpler than Patterson's algorithm. All algorithm layers are expressed as Sage scripts backed by test scripts. All theorems are formally verified. The paper also covers the use of decoding inside the Classic McEliece cryptosystem, including reliable recognition of valid inputs.

1 Introduction

This paper is aimed at a reader who

- is interested in how ciphertexts are decrypted in the McEliece cryptosystem,

- has arrived at a mysterious-sounding "Goppa decoding" subroutine, and

- wants to understand how this works without taking a coding-theory course.

A busy reader can jump straight to Algorithm 5.1.1 and Theorem 5.1.2 for a concise answer, highlighting the main mathematical objects inside the decoding process.

In more detail: The cryptosystem uses a large family of subspaces of the vector space $\mathbb{F}_2^n$, namely "classical binary Goppa codes" defined by squarefree degree-t polynomials. This paper reviews a simple polynomial-time "t-error-correction" algorithm for these codes: an algorithm that recovers a vector c in a specified subspace given a vector that agrees with c on at least $n - t$ positions. Components of the algorithm are introduced in a bottom-up order: Sections 2, 3, 4, and 5 present, respectively, "interpolation", finding "approximants", interpolation with errors ("Reed–Solomon decoding"), and Goppa decoding.

1.1 Hasn't this been done already?

Goppa codes are more than 50 years old. There are many descriptions of Goppa decoders in the literature. Self-contained descriptions appear in, e.g., van Tilborg's coding-theory textbook [Til93, Section 4.5, "A decoding algorithm"], a Preneel–Bosselaers–Govaerts–Vandewalle paper on a software implementation of the McEliece cryptosystem [PreBGV92,

---

Section 5.3], a Ghosh–Verbauwhede paper on a constant-time hardware implementation of the cryptosystem [GhoV14, Algorithm 3], and the Overbeck–Sendrier survey of code-based cryptography [OveS09, pages 139–140].

All of these sources—and many more—are describing an algorithm introduced by Patterson [Patt75, Section V] to correct t errors for binary Goppa codes defined by squarefree degree-t polynomials. McEliece's paper introducing the McEliece cryptosystem [Mce78] had also pointed to Patterson's algorithm.

However, Patterson's algorithm isn't the simplest fast binary-Goppa decoder. A side issue here is that there are tradeoffs between simplicity and the number of errors corrected (which in turn influences the required McEliece key size), as the following variations illustrate: Patterson's paper contained a simpler algorithm to correct $\lfloor t/2 \rfloor$ errors; more complicated "list decoding" algorithms, starting with Sudan [Sud97] and then Guruswami–Sudan [GurS98], correct slightly more than t errors. But let's focus on fast algorithms to correct exactly the t errors traditionally used in the McEliece cryptosystem. The main issue is that, within these algorithms, Patterson's algorithm isn't the simplest.

Goppa had already pointed out in the first paper on Goppa codes [Gop70, Section 4] that a binary Goppa code defined by a squarefree degree-t polynomial g is also defined by g^2. The problem of correcting t errors in the code defined by g^2 immediately reduces to the problem of polynomial interpolation with t errors, i.e., Reed–Solomon decoding. The resulting binary-Goppa decoder is simpler than Patterson's.

The benefits of simplicity go beyond general accessibility of the topic: software for simpler algorithms tends to be easier to optimize, easier to protect against timing attacks, and easier to test. It isn't a coincidence that the same simple structure is used in the state-of-the-art McEliece software from Bernstein–Chou–Schwabe [BernCS13], Chou [Cho18], and Chen–Chou [ChenC21]. This software eliminates data-dependent timing and at the same time includes many speedups in subroutines. Avoiding Patterson's algorithm also seems likely to help for formal verification of software correctness, a top challenge for post-quantum cryptography today.

Maybe someday software for Patterson's algorithm will catch up in these other features, and maybe it will bring further speedups—or maybe not. Patterson's algorithm uses degree t instead of degree $2t$ for some computations, but it also includes extra computations, such as inversion modulo g; the literature does not make clear whether the speedups outweigh the slowdowns. Also, even if Patterson's algorithm ends up faster, surely there will be applications where simplicity is more important. Having *only* Patterson's algorithm brings to mind Knuth's quote [Knu74, page 268] that "premature optimization is the root of all evil".

For an audience familiar with coding theory, it suffices to say "the Goppa code for g is the same as the Goppa code for g^2; now use your favorite Reed–Solomon decoder as an alternant decoder" (essentially as in [BernLP11, Section 5], which also generalizes from $\mathbb{F}_2$ to $\mathbb{F}_q$). For a broader audience, one can reduce to the previous sentence by saying "Take the following course on coding theory". But it's more efficient for the audience to take a minicourse focusing on this type of decoder—and there doesn't seem to be any such minicourse in the literature.

To summarize, this paper is a general-audience introduction to a simple t-error decoder for binary Goppa codes defined by squarefree degree-t polynomials, with the proof factored through a proof of a t-error Reed–Solomon decoder.

1.2   Bonus features

This paper systematically presents each algorithm layer in two forms: a theorem with a full proof (Theorems 2.1.2, 3.1.2, 4.1.2, and 5.1.2), and an algorithm statement (Algorithms 2.1.1, 3.1.1, 4.1.1, and 5.1.1). Figure 1.2.1 summarizes the inputs and outputs.
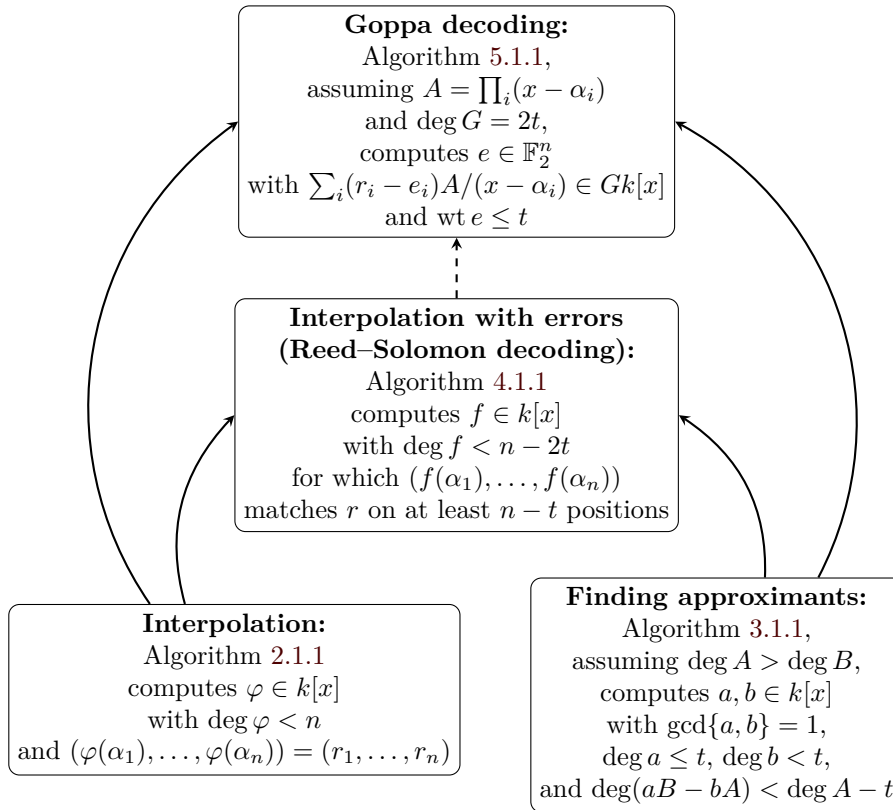
Figure 1.2.1: Summary of algorithm layers inside this Goppa decoder. See algorithm statements later in paper for full input-output restrictions. Solid arrow from L to M means that L is a subroutine in the M algorithm. Dashed arrow from L to M means that L is not a subroutine in the M algorithm, but the proof of L is used in the proof of M.

Readers not familiar with the polynomial ring $k[x]$ should start with Appendix A.

Each algorithm layer is presented as a script in the Sage [The20] mathematics system rather than as pseudocode. The scripts use Sage's built-in support for fields, matrices, and polynomials. The scripts do not use Sage's functions for interpolation, the Berlekamp–Massey algorithm, etc. Appendix B presents tests of the algorithms on random inputs.

As context, Section 8 explains how the Classic McEliece cryptosystem uses a Goppa decoder. In this context, it is important to reliably recognize invalid ciphertexts. Most descriptions of decoders in the literature simply *assume* that the input vector has at most t errors, but for cryptography one has to *verify* the input vector. Mathematically, the traditional question about the set of decoded vectors is whether it contains every vector with at most t errors; the more subtle question is whether the set is *exactly* the set of vectors with at most t errors, rather than a strict superset. This paper includes various efficient characterizations of vectors having at most t errors (Theorems 4.1.3, 5.1.3, and 7.1), and an analysis of safe options for recognizing valid ciphertexts (Sections 8.3 and 8.4).

For each theorem, this paper includes a HOL-Light-verified formalization of the theorem and a Lean-verified formalization of the theorem. See Appendix C. This verification is a step towards, but should not be confused with, verification of theorems stating the correctness of software for Goppa decoding (or for the McEliece cryptosystem) in a defined model of computation.

Finally, this paper includes extensive pointers to the literature, primarily to give appropriate credit but also to point the reader to further material explaining how to turn

```
def interpolator(n,k,a,r):
  a,r = list(a),list(r)
  assert k.is_field()
  assert len(a) == n and len(set(a)) == n and len(r) == n
  kpoly.<x> = k[]
  A = kpoly(prod(x-a[j] for j in range(n)))
  Aprime = A.derivative()
  return kpoly(sum(((r[i]/Aprime(a[i]))*(A//(x-a[i]))) for i in range(n)))
```

Algorithm 2.1.1: Direct interpolation algorithm to compute $\varphi \in k[x]$ with $\deg \varphi < n$ and $(\varphi(\alpha_1), \ldots, \varphi(\alpha_n)) = (r_1, \ldots, r_n)$. Inputs: integer $n \geq 0$; field k; $(\alpha_1, \ldots, \alpha_n) \in k^n$ with distinct entries; $(r_1, \ldots, r_n) \in k^n$.

this decoder into today's state-of-the-art software.

1.3 Acknowledgments

2 Interpolation

This section explains how to recover a polynomial $f \in k[x]$ with $\deg f < n$, given $(f(\alpha_1), \ldots, f(\alpha_n))$. Here $\alpha_1, \ldots, \alpha_n$ are distinct elements of a field k. See Section 4 for a generalization that handles as many as t errors in the input vector, at the expense of requiring $\deg f < n - 2t$.

2.1 An interpolation algorithm

Algorithm 2.1.1 interpolates a polynomial from its values, using the φ formula in Theorem 2.1.2. That formula is usually called the "Lagrange interpolation formula", but Waring [War1779] published the same formula earlier.

The algorithm starts by computing the polynomial $A = \prod_j (x - \alpha_j)$. This takes $\Theta(n^2)$ operations in k using schoolbook arithmetic in $k[x]$. Then, for each i, the algorithm uses $\Theta(n)$ operations to compute $A/(x - \alpha_i) = \prod_{j \neq i}(x - \alpha_j)$, and $\Theta(n)$ operations to compute $A'(\alpha_i) = \prod_{j \neq i}(\alpha_i - \alpha_j)$, where A' is the derivative of A. In total this takes $\Theta(n^2)$ operations.

Theorem 2.1.2 (direct interpolation). *Let n be a nonnegative integer. Let k be a field. Let $\alpha_1, \ldots, \alpha_n$ be distinct elements of k. Let $r_1, \ldots, r_n$ be elements of k. Define*

$$\varphi = \sum_i r_i \prod_{j \neq i} \frac{x - \alpha_j}{\alpha_i - \alpha_j}.$$

Then $\{f \in k[x] : \deg f < n, (f(\alpha_1), \ldots, f(\alpha_n)) = (r_1, \ldots, r_n)\} = \{\varphi\}$.

Proof. By construction φ is a sum of n terms, each term having degree at most $n - 1$ (more precisely, degree $n - 1$ if $r_i \neq 0$, otherwise degree $-\infty$), and hence has degree at most $n - 1$.

Observe that $\varphi(\alpha_h) = \sum_i r_i \prod_{j \neq i}(\alpha_h - \alpha_j)/(\alpha_i - \alpha_j)$. If $i \neq h$ then $\alpha_h - \alpha_j = 0$ for $j = h$ so $\prod_{j \neq i}(\alpha_h - \alpha_j)/(\alpha_i - \alpha_j) = 0$. If $i = h$ then $\prod_{j \neq i}(\alpha_h - \alpha_j)/(\alpha_i - \alpha_j) = \prod_{j \neq h}(\alpha_h - \alpha_j)/(\alpha_h - \alpha_j) = 1$. Hence $\varphi(\alpha_h) = r_h$ as claimed.

Any $f \in k[x]$ with $\deg f < n$ and $(f(\alpha_1), \ldots, f(\alpha_n)) = (r_1, \ldots, r_n)$ must have $f = \varphi$. Otherwise $f - \varphi$ is a nonzero polynomial, so it has at most $\deg(f - \varphi) < n$ roots, but it visibly has the distinct roots $\alpha_1, \ldots, \alpha_n$, contradiction. $\square$

2.2   More interpolation algorithms

An older interpolation method recursively interpolates a polynomial $g \in k[x]$ satisfying $(g(\alpha_2), \ldots, g(\alpha_n)) = ((r_2 - r_1)/(\alpha_2 - \alpha_1), \ldots, (r_n - r_1)/(\alpha_n - \alpha_1))$, and then takes $f = r_1 + (x - \alpha_1)g$. This interpolation method, Newton's "divided differences" method, is more complicated than direct interpolation to express as a concise formula but also costs $\Theta(n^2)$.

There is an extensive literature on algorithms using $n^{1+o(1)}$ operations in k, not just for interpolation but also for multiplication (multiplying $\sum_{0 \le i < n} f_i x^i$ by $\sum_{0 \le i < n} g_i x^i$), division, and other basic operations. See generally [Bern08a].

One particularly fast case is interpolating f from its values at every point in a finite field k, using various types of "fast Fourier transforms". A difficulty here is that each of these transforms uses a standard order of points, while $\alpha_1, \ldots, \alpha_n$ are in a secret order inside the McEliece cryptosystem. There are algorithms to apply a secret permutation without using secret array indices; see generally [Bern20].

3   Approximants

Let A, B be elements of $k[x]$ with $\deg A > \deg B$, and consider the rank-2 lattice $k[x] \cdot (A, 0) + k[x] \cdot (B, 1)$ inside $k[x]^2$. Readers familiar with integer-coefficient lattices should note that this is something different, a $k[x]$-lattice. The elements of this lattice have the form $a(B, 1) - b(A, 0) = (aB - bA, a)$ for polynomials $a, b \in k[x]$. The vector $(aB - bA, a)$ is a short vector when both $aB - bA$ and a have low degree.

It's useful to vary the weights put on the two vector components: let t be a nonnegative integer, and consider the lattice $k[x] \cdot (A, 0) + k[x] \cdot (B, x^{\deg A - 2t - 1})$. The point of this section is to find, inside this lattice, a minimum-length nonzero vector $(aB - bA, ax^{\deg A - 2t - 1})$.

If $2t \geq \deg A$ then there's a denominator here. One can manually track weights of polynomials to avoid ever having to consider denominators; this is how the theorems below are phrased. One can instead allow denominators, dropping the requirement of staying inside $k[x]^2$. Alternatively, one can clear denominators by considering the lattice $k[x] \cdot (x^{2t+1-\deg A} A, 0) + k[x] \cdot (x^{2t+1-\deg A} B, 1)$ in the case $2t \geq \deg A$. Or one can simply prohibit this case; such large values of t aren't of interest for the application to decoding.

3.1   An approximant algorithm

Theorem 3.1.2 says that one can arrange for both $aB - bA$ and $ax^{\deg A - 2t - 1}$ to have degree at most $\deg A - t - 1$. This also forces b to have degree below t. (Otherwise $\deg bA \geq \deg A + t$, while $\deg aB = \deg B + \deg a < \deg A + t$, so $\deg(aB - bA) \geq \deg A + t$.) One can also take a, b to be coprime; then, by Theorem 3.1.3, any lattice vector of degree at most $\deg A - t - 1$ must be a multiple of this particular vector $(aB - bA, ax^{\deg A - 2t - 1})$.

Algorithm 3.1.1 computes a, b from t, k, A, B. This algorithm works in the same way as the proof of Theorem 3.1.2, constructing coefficients of a, b as solutions to an explicit system of $2t$ equations in $2t + 1$ variables. Straightforward matrix algorithms use $O(t^3)$ operations in k, typically $\Theta(t^3)$ operations.

Theorem 3.1.2 (approximants). *Let t be a nonnegative integer. Let k be a field. Let A, B be elements of $k[x]$ with $\deg A > \deg B$. Then there exist $a, b \in k[x]$ such that $\gcd\{a, b\} = 1$, $\deg a \leq t$, $\deg b < t$, and $\deg(aB - bA) < \deg A - t$.*

```
def approximant(t,k,A,B):
  assert t >= 0 and A.base_ring() == k and B.base_ring() == k
  kpoly,n = A.parent(),A.degree()
  assert n > B.degree()
  M = [   [ B[t+n-1-i-j] for i in range(t+1)]
        + [-A[t+n-1-i-j] for i in range(t)  ] for j in range(2*t)]
  M = matrix(k,2*t,2*t+1,M)
  ab = list(M.right_kernel().gens()[0])
  a,b = kpoly(ab[:t+1]),kpoly(ab[t+1:])
  d = gcd(a,b)
  return a//d,b//d
```

Algorithm 3.1.1: Linear-algebra algorithm to compute $a, b \in k[x]$ with $\gcd\{a,b\} = 1$, $\deg a \le t$, $\deg b < t$, and $\deg(aB - bA) < \deg A - t$. Inputs: integer $t \ge 0$; field k; $A \in k[x]$; $B \in k[x]$ with $\deg A > \deg B$. Note that $[\ldots]+[\ldots]$ in Sage is concatenation of lists.

Proof. Define $n = \deg A$. Consider the following k-linear map from k^{2t+1} to k^{2t}: the input is a vector $(a_0, a_1, \ldots, a_{t-1}, a_t, b_0, b_1, \ldots, b_{t-1})$; the output entries are the coefficients of $x^{n+t-1}, x^{n+t-2}, \ldots, x^{n-t}$ in $aB - bA$, where $a = a_t x^t + a_{t-1} x^{t-1} + \cdots + a_1 x + a_0$ and $b = b_{t-1} x^{t-1} + \cdots + b_1 x + b_0$. Explicitly, if $A = A_n x^n + A_{n-1} x^{n-1} + \cdots$ and $B = B_{n-1} x^{n-1} + \cdots$ then the output entries are $a_t B_{n-1} - b_{t-1} A_n$, $a_t B_{n-2} + a_{t-1} B_{n-1} - b_{t-1} A_{n-1} - b_{t-2} A_n$, etc.

The input dimension $2t + 1$ exceeds the output dimension $2t$, so there is a nonzero input that maps to zero. The corresponding polynomials a, b have $(a, b) \ne (0, 0)$, $\deg a \le t$, $\deg b < t$, and $\deg(aB - bA) < n - t$. Finally, to ensure that $\gcd\{a, b\} = 1$, replace (a, b) with $(a/\gcd\{a, b\}, b/\gcd\{a, b\})$; this subtracts $\deg \gcd\{a, b\} \ge 0$ from $\deg a$, $\deg b$, and $\deg(aB - bA)$. $\square$

Theorem 3.1.3 (the best-approximation property of approximants). *Let t be a nonnegative integer. Let k be a field. Let A, B, a, b, c, d be elements of $k[x]$ such that $\gcd\{a, b\} = 1$, $\deg a \le t$, $\deg(aB - bA) < \deg A - t$, $\deg c \le t$, and $\deg(cB - dA) < \deg A - t$. Then $(c, d) = (\lambda a, \lambda b)$ for some $\lambda \in k[x]$.*

One way to describe the proof is as follows: if the lattice mentioned above has two independent vectors $(aB - bA, ax^{\deg A - 2t - 1})$, $(cB - dA, cx^{\deg A - 2t - 1})$ of degree at most $\deg A - t - 1$, then the lattice determinant has degree at most $2 \deg A - 2t - 2$; but, by inspection, the lattice determinant is $Ax^{\deg A - 2t - 1}$, of degree $2 \deg A - 2t - 1$. Combining linear dependence with $\gcd\{a, b\} = 1$ forces $(c, d) = (\lambda a, \lambda b)$.

Proof. $c(aB - bA) - a(cB - dA) = (ad - cb)A$. The left side has degree smaller than $\deg A$, so $ad - cb = 0$. In particular, $cb \in ak[x]$; but $\gcd\{a, b\} = 1$, so $c \in ak[x]$, and similarly $d \in bk[x]$. Write λ for c/a if $a \ne 0$, or for d/b if $b \ne 0$; in both cases $(c, d) = (a\lambda, b\lambda)$ as claimed. $\square$

3.2 More approximant algorithms

One can construct a, b via an extended-gcd computation. Straightforward extended-gcd algorithms use $O(t^2)$ operations, typically $\Theta(t^2)$ operations.

More sophisticated extended-gcd algorithms use $t^{1+o(1)}$ operations. See [Bern08a, Section 21]. Applying a sequence of $2t$ "divsteps", taking $n = 2t$ in [BernY19, Theorems A.1 and A.2], uses $t^{1+o(1)}$ operations with the "jump" algorithms in [BernY19] while avoiding the timing variability of polynomial division.

3.3 Approximants as ratios

Why does this section take a minus sign on b? Why multiply a by B and b by A, rather than a by A and b by B?

Answer: small $aB - bA$ means that the rational function b/a is close to B/A. This rational function b/a has small height, meaning that its numerator and denominator are small. The perspective of small-height rational approximations has played an important role in the development of theory and fast algorithms in this area.

From this perspective, Theorem 3.1.3 (in the case $c \neq 0$) is equivalent to the following: if $\deg a \leq t$, $\deg c \leq t$, $\deg(B/A - b/a) < -t - \deg a$, and $\deg(B/A - d/c) < -t - \deg c$, then d/c must equal b/a. To approximate B/A more closely than the fraction b/a constructed in Theorem 3.1.2, one must take larger-degree denominators.

With the following definition, the conclusion of Theorem 3.1.2 is that there is an approximant to B/A at degree t. This definition would also slightly compress the statement of Theorem 3.1.3 and the statements of some theorems later in this paper. For the benefit of a reader looking at just one theorem, this paper avoids using this definition in theorem statements, but readers exploring the literature may find this definition useful. Analogous comments apply to, e.g., Definition 4.3.1 below.

Definition 3.3.1. *Let k be a field. Let A, B be elements of $k((x^{-1}))$ with $A \neq 0$. Let t be a nonnegative integer. If $(a, b) \in k[x] \times k[x]$ satisfy $\gcd\{a, b\} = 1$, $\deg a \leq t$, and $\deg(aB - bA) < \deg A - t$ then b/a is an* **approximant to B/A at degree t.**

For simplicity the theorems in this section were stated specifically for $A, B \in k[x]$, but the concepts and proofs do not require this. This paper does not define $k((x^{-1}))$, but instead notes that $k((x^{-1}))$ contains the field $k(x)$ of rational functions in x, and that $k(x)$ in turn contains the polynomial ring $k[x]$, so readers not familiar with $k((x^{-1}))$ can substitute $k[x]$ for $k((x^{-1}))$ in the definition.

The condition $\deg(aB - bA) < \deg A - t$ is equivalent to $\deg(B/A - b/a) < -t - \deg a$; this is why it is safe to describe the input as B/A rather than (A, B). As for the output, knowing the ratio b/a and knowing $\gcd\{a, b\} = 1$ does not exactly determine the pair (a, b), but the only ambiguity is that one can replace (a, b) by $(\lambda a, \lambda b)$ for $\lambda \in k^*$; this replacement does not affect the conditions on $\deg a$, $\deg b$, and $\deg(aB - bA)$.

3.4 History

Euclid's subtractive algorithm [Euc300BC, Book VII, Propositions 1–2; translation: "the less of the numbers AB, CD being continually subtracted from the greater"] recognizes coprime integers, and, more generally, computes the gcd of two integers.

What is typically called Euclid's algorithm—see [Knu97, Section 4.5.2, text before Algorithm E] for an argument that this must be what Euclid had in mind—is a variant that iterates $(A, B) \mapsto (B, A \bmod B)$. This is much faster than the original algorithm when $\lfloor A/B \rfloor$ is large. This version also has a polynomial analogue: Stevin [Ste1585, page 241 of original, page 123 of cited PDF] computed polynomial gcd by iterating $(A, B) \mapsto (B, A \bmod B)$.

According to [Bre81, page 3], an extended-gcd algorithm computing solutions to $aB - bA = 1$, for coprime integers A, B, is due to Aryabhata around the 6th century, and the forward recurrence relation for coefficients in the extended algorithm—in other words, numerators and denominators of convergents to a continued fraction—is due to Bhascara in the 12th century.

Lagrange [Lag1776] used convergents to continued fractions of rational functions as small-height approximations to power series. Kronecker [Kro1881, pages 118–119 of cited PDF] gave both the continued-fraction construction and ("in directer Weise") the linear-algebra construction. Consequently, it seems reasonable to credit Theorem 3.1.2 to

Lagrange, but the short proof to Kronecker. Small-height approximations to power series are often miscredited to [Pad1892] under the name "Padé approximants".

An earlier paper of Lagrange [Lag1773, pages 723–728 of cited URL] had described, in the integer case, an algorithm for basis reduction for rank-2 lattices—in the context of simplifying quadratic forms, rather than as a perspective on extended-gcd computations. Lagrange reduction is often miscredited to [Gau1801] under the name "Gauss reduction".

In coding theory, finding an approximant is called "solving the key equation". The "key equation" is, by definition, the congruence $d - aB \in Ak[x]$ where $\deg a \leq t$ and $\deg d < \deg A - t$; this is equivalent to the equation $d = aB - bA$ where $\deg a \leq t$ and $\deg d < \deg A - t$. Decoding algorithms are typically factored through this concept, and often the proofs are factored through continued-fraction facts; when the continued-fraction machinery is stripped away, those facts boil down to Theorem 3.1.3. For the more complicated setting of list-decoding algorithms, short vectors in arbitrary-rank lattices often appear as an abstraction layer; see, e.g., [Bon02], [Bern08b], [CohH15], [Bern11a], and [Bern11b].

4   Interpolation with errors

This section explains how to recover a polynomial $f \in k[x]$ with $\deg f < n - 2t$, given a vector that matches $(f(\alpha_1), \ldots, f(\alpha_n))$ on at least $n - t$ positions. Here $\alpha_1, \ldots, \alpha_n$ are distinct elements of k. The special case $t = 0$ of this problem was handled in Section 2, and is used as a subroutine for handling the general case.

If $e \in k^n$, where n is a nonnegative integer, then the **Hamming weight** of e, written "wt e", means $\#\{i \in \{1, 2, \ldots, n\} : e_i \neq 0\}$, the number of nonzero positions in e. A vector $r \in k^n$ matches $c = (f(\alpha_1), \ldots, f(\alpha_n))$ on at least $n - t$ positions if and only if $\mathrm{wt}(r - c) \leq t$, i.e., $\mathrm{wt}(r_1 - f(\alpha_1), \ldots, r_n - f(\alpha_n)) \leq t$.

4.1   An interpolation-with-errors algorithm

Algorithm 4.1.1 recovers f, given (n, t, k, α, r) with $\mathrm{wt}(r_1 - f(\alpha_1), \ldots, r_n - f(\alpha_n)) \leq t$. The algorithm has three steps:

- Interpolate the input vector r into a polynomial $B \in k[x]$ with $\deg B < n$, as in Theorem 2.1.2.

- Compute an approximant b/a to B/A at degree t as in Theorem 3.1.2, where $A = \prod_i (x - \alpha_i)$.

- Compute $f = B - bA/a$. Theorem 4.1.2 says that this works.

The algorithm returns `None` for invalid input vectors, recognized as follows: f exists if and only if $A \in ak[x]$ (which is equivalent to $\#\{j : a(\alpha_j) = 0\} = \deg a$) and $\deg(aB - bA) < n - 2t + \deg a$. See Theorem 4.1.3.

Beware that Sage's `degree` function is not the same as the conventional degree function for polynomials: on input 0, it returns -1 rather than $-\infty$. This is why Algorithm 4.1.1 includes a separate test for $aB - bA = 0$.

Theorem 4.1.2 (interpolation with errors). *Let n, t be nonnegative integers. Let k be a field. Let $\alpha_1, \ldots, \alpha_n$ be distinct elements of k. Define $A = \prod_i (x - \alpha_i)$. Let B, a, b, f be elements of $k[x]$ with $\gcd\{a, b\} = 1$, $\deg a \leq t$, $\deg(aB - bA) < n - t$, and $\deg f < n - 2t$. Define $e = (B(\alpha_1) - f(\alpha_1), \ldots, B(\alpha_n) - f(\alpha_n))$. Assume $\mathrm{wt}\, e \leq t$. Then $A \in ak[x]$; $f = B - bA/a$; $\deg(aB - bA) < n - 2t + \deg a$; and $\{i : e_i \neq 0\} = \{i : a(\alpha_i) = 0\}$.*

```
from interpolator import interpolator
from approximant import approximant

def interpolator_with_errors(n,t,k,alpha,r):
  alpha,r = list(alpha),list(r)
  assert k.is_field()
  assert len(alpha) == n and len(set(alpha)) == n and len(r) == n
  B = interpolator(n,k,alpha,r)
  kpoly = B.parent()
  A = kpoly(prod(kpoly([-alpha[j],1]) for j in range(n)))
  a,b = approximant(t,k,A,B)
  if a.divides(A):
    if a*B-b*A == 0 or (a*B-b*A).degree() < n-2*t+a.degree():
      return B-b*A//a
```

Algorithm 4.1.1: Algorithm to compute the unique $f \in k[x]$ with $\deg f < n - 2t$ for which $(f(\alpha_1), \ldots, f(\alpha_n))$ matches r on at least $n - t$ positions, or `None` if no such f exists. Inputs: integer $n \geq 0$; integer $t \geq 0$; field k; $(\alpha_1, \ldots, \alpha_n) \in k^n$ with distinct entries; $r \in k^n$.

Proof. Define $E = \prod_{i:e_i=0}(x - \alpha_i)$ and $c = \prod_{i:e_i \neq 0}(x - \alpha_i)$. Then $Ec = A$.

If $e_i = 0$ then $B(\alpha_i) = f(\alpha_i)$ so $B - f \in (x - \alpha_i)k[x]$. This implies $B - f \in Ek[x]$, since $\alpha_1, \ldots, \alpha_n$ are distinct.

Define $d = (B - f)/E \in k[x]$. Then $dA = (B - f)c$ so $cB - dA = cf$. Note that $\deg c = \text{wt}\, e \leq t$; also $\deg f < n - 2t$ so $\deg(cB - dA) < n - t$.

The conditions of Theorem 3.1.3 are satisfied: A, B, a, b, c, d are elements of $k[x]$ with $\gcd\{a, b\} = 1$, $\deg a \leq t$, $\deg(aB - bA) < \deg A - t$, $\deg c \leq t$, and $\deg(cB - dA) < \deg A - t$.

Hence $(c, d) = (\lambda a, \lambda b)$ for some $\lambda \in k[x]$ by Theorem 3.1.3. By construction $c \neq 0$, so $a \neq 0$. Also $A \in ck[x] \subseteq ak[x]$. Consequently $B - f = dA/c = bA/a$, so $f = B - bA/a$ and $\deg(aB - bA) = \deg af < n - 2t + \deg a$.

To see that $e_i \neq 0$ exactly when $a(\alpha_i) = 0$: $A(\alpha_i) = 0$ so if $a(\alpha_i) = 0$ then, by Bernoulli's rule, $(A/a)(\alpha_i) = A'(\alpha_i)/a'(\alpha_i) \neq 0$ where a', A' are the derivatives of a, A respectively; also $b(\alpha_i) \neq 0$ since $\gcd\{a, b\} = 1$, so $e_i = (B - f)(\alpha_i) = (bA/a)(\alpha_i) \neq 0$. If $a(\alpha_i) \neq 0$ then $e_i = (bA/a)(\alpha_i) = 0$ since $A(\alpha_i) = 0$. □

Theorem 4.1.3 (checking interpolation with errors). *Let n, t be nonnegative integers. Let k be a field. Let $\alpha_1, \ldots, \alpha_n$ be distinct elements of k. Define $A = \prod_i(x - \alpha_i)$. Let B, a, b, f be elements of $k[x]$ such that $\deg a \leq t$, $A \in ak[x]$, $\deg(aB - bA) < n - 2t + \deg a$, and $af = aB - bA$. Then $a \neq 0$; $\deg f < n - 2t$; and $\text{wt}(B(\alpha_1) - f(\alpha_1), \ldots, B(\alpha_n) - f(\alpha_n)) \leq t$.*

The condition $\deg(aB - bA) < n - 2t + \deg a$ here cannot be weakened to the condition $\deg(aB - bA) < n - t$. Consider, e.g., $n = 3$; $t = 1$; any field k with $\#k \geq 3$; any distinct $\alpha_1, \alpha_2, \alpha_3 \in k$; $A = (x - \alpha_1)(x - \alpha_2)(x - \alpha_3)$; $B = x$; $a = 1$; and $b = 0$. Then $\deg(aB - bA) = \deg x = 1 = n - 2t + \deg a$. The values of B on $\alpha_1, \alpha_2, \alpha_3$ are $\alpha_1, \alpha_2, \alpha_3$ respectively, and there is no polynomial f with $\deg f < 1$ that matches more than one of those values.

Proof. By assumption $A \in ak[x]$. This forces $a \neq 0$ since $A \neq 0$; also note that $\deg f = \deg(aB - bA) - \deg a < n - 2t$.

If $B(\alpha_i) - f(\alpha_i) \neq 0$ then $(bA/a)(\alpha_i) \neq 0$, so $(A/a)(\alpha_i) \neq 0$, but $A(\alpha_i) = 0$, so $a(\alpha_i) = 0$. The number of roots of a is at most $\deg a \leq t$, and $\alpha_1, \ldots, \alpha_n$ are distinct, so $\text{wt}(B(\alpha_1) - f(\alpha_1), \ldots, B(\alpha_n) - f(\alpha_n)) \leq t$. □

4.2   More algorithms: varying the pair (A, B)

If $A = \prod_i (x - \alpha_i)$ and $B = \sum_i (r_i/A'(\alpha_i))A/(x - \alpha_i)$, where A' is the derivative of A, then

$$\frac{B}{A} = \sum_i \frac{r_i}{A'(\alpha_i)(x - \alpha_i)} = \sum_i \frac{r_i}{A'(\alpha_i)} \left(\frac{1}{x} + \frac{\alpha_i}{x^2} + \cdots \right) = \sum_{s \geq 0} \frac{1}{x^{s+1}} \sum_i \frac{r_i \alpha_i^s}{A'(\alpha_i)}.$$

One can vary the choice of (A, B) while preserving the ratio B/A: e.g., one can take $A = 1$ and $B = \sum_{s \geq 0} x^{-s-1} \sum_i r_i \alpha_i^s / \prod_{j \neq i}(\alpha_j - \alpha_i)$. Formally, this requires defining $k((x^{-1}))$; but the terms in B/A after x^{-2t} do not matter for decoding, so one can take $A = x^{2t}$ and $B = \sum_{0 \leq s < 2t} \sum_i r_i \alpha_i^s x^{2t-s-1} / \prod_{j \neq i}(\alpha_j - \alpha_i)$.

These variations preserve the set of $(a, b) \in k[x] \times k[x]$ such that $\gcd\{a, b\} = 1$, $\deg a \leq t$, and $\deg(aB - bA) < \deg A - t$. If $\deg f < n - 2t$ and $\operatorname{wt} e \leq t$ with $e = (r_1 - f(\alpha_1), \ldots, r_n - f(\alpha_n))$ then $\{i : e_i \neq 0\} = \{i : a(\alpha_i) = 0\}$ for any such (a, b). If one assumes that $\mathbb{F}_2 \subseteq k$ and that $e \in \mathbb{F}_2^n$ then knowing $\{i : e_i \neq 0\}$ is enough information to reconstruct e and thereby f. To instead handle arbitrary $e \in k^n$, one can use any of these variants of (A, B) to compute (a, b), and then return to the original (A, B) to apply the formula $f = B - bA/a$ in Theorem 4.1.2.

4.3   Reed–Solomon codes

The set of vectors $(f(\alpha_1), \ldots, f(\alpha_n))$ is called a Reed–Solomon code; see Definition 4.3.1. This is a subspace of the k-vector space k^n. Each vector in the code is called a codeword. With this terminology, Theorem 4.1.2 recovers a Reed–Solomon codeword from a vector that matches the codeword on at least $n - t$ positions.

Definition 4.3.1. *Let n, t be nonnegative integers. Let k be a field. Let $\alpha_1, \ldots, \alpha_n$ be distinct elements of k. Then $\{(f(\alpha_1), \ldots, f(\alpha_n)) : f \in k[x], \deg f < n - 2t\}$ is the* **Reed– Solomon code over k of dimension $n - 2t$ with support $(\alpha_1, \ldots, \alpha_n)$**.

4.4   History

Reed–Solomon [ReeS60] suggested encoding a polynomial $f \in k[x]$ with $\deg f < n - 2t$ as $(f(\alpha_1), \ldots, f(\alpha_n))$ for distinct $\alpha_1, \ldots, \alpha_n$, so as to be able to recover f even if t vector entries are corrupted. The point is that the code

$$C = \{(f(\alpha_1), \ldots, f(\alpha_n)) : f \in k[x], \deg f < n - 2t\}$$

has "minimum distance" at least $2t + 1$ (every nonzero $c \in C$ has $\operatorname{wt} c \geq 2t + 1$), so the map $(e, f) \mapsto e + (f(\alpha_1), \ldots, f(\alpha_n))$ from $\{(e, f) \in k^n \times k[x] : \operatorname{wt} e \leq t, \deg f < n - 2t\}$ to k^n is injective. This raises the question of how efficiently one can decode $\leq t$ errors in C, i.e., recover (e, f) from $e + (f(\alpha_1), \ldots, f(\alpha_n))$.

Assume $n > 2t$. Prange's "information-set decoding" [Pra62] interpolates f from $n - 2t$ values at selected positions in the input vector, checks the remaining values of f to deduce e, and, if e has the wrong weight, tries another selection of $n - 2t$ positions. This takes polynomial time if t is close enough to 0 or $n/2$, but is much slower in general. Reed and Solomon did not have the idea of checking the weight of e: they had instead suggested trying many selections of $n - 2t$ positions to find the most popular choice of f, and relying on an upper bound for how often any particular incorrect choice could appear.

Forney [For65a, Chapter 4] (see also [For65b]) introduced a polynomial-time decoding algorithm for Reed–Solomon codes. Forney's algorithm simplified and extended an algorithm by Gorenstein and Zierler [GorZ61], which handled the special case $\{\alpha_1, \ldots, \alpha_n\} = k^*$. The latter algorithm extended an algorithm by Peterson [Pet60], which handled the following special case: $\mathbb{F}_2 \subseteq k$, each $f(\alpha_j)$ is in $\mathbb{F}_2$, and $e \in \mathbb{F}_2^n$.

The Peterson–Gorenstein–Zierler–Forney algorithm is bottlenecked by matrix operations that, when carried out in a simple way, use $n^{3+o(1)}$ operations in k, assuming $n \in t^{1+o(1)}$. The exponent for generic matrix operations was later reduced below 3 (starting with exponent $\log_2 7$ for matrix multiplication by Strassen [Stra69], along with the same exponent for solving linear equations under various nonsingularity constraints), but it turns out that one can obtain much better decoding speeds using the structure of these particular matrices.

Berlekamp [Berl68] introduced a decoding algorithm using just $n^{2+o(1)}$ operations instead of $n^{3+o(1)}$ operations; the main work inside the algorithm is polynomial arithmetic rather than matrix arithmetic. Massey [Mas69] streamlined Berlekamp's algorithm and factored the algorithm into two layers, where the top layer is a decoder and the bottom layer is a subroutine for "shift register synthesis". The subroutine is called the Berlekamp–Massey algorithm.

Sugiyama–Kasahara–Hirasawa–Namekawa [SugKHN75] built an $n^{2+o(1)}$ algorithm for Reed–Solomon decoding on top of an extended-gcd computation. Algorithms using just $n^{1+o(1)}$ operations were already known for gcd (see [Bern08a, Section 21.6] for history) and for all other necessary subroutines; these algorithms were applied to Reed–Solomon decoding by Justesen [Jus76] and independently Sarwate [Sar77], reducing the costs of decoding to $n^{1+o(1)}$.

It turned out that Berlekamp decoders and Sugiyama–Kasahara–Hirasawa–Namekawa decoders are equivalent: Mills [Mil75] pointed out that "shift register synthesis" is the same as the problem of finding approximants, the problem of finding (a, b) in Theorem 3.1.2. See also [WelS79] for how the result after each polynomial division inside an extended-gcd computation appears inside in the Berlekamp–Massey algorithm; [Dor87] for an extended-gcd explanation of all further quantities inside the Berlekamp–Massey algorithm; and [BernY19, Appendix C] for a reformulation in terms of "divsteps". In a nutshell, the polynomials in the Berlekamp–Massey algorithm are polynomials in an extended-gcd computation but with coefficients in reverse order.

This does not mean that all Reed–Solomon decoders are the same. See, for example, Section 4.2 regarding different choices of (A, B); the choice of (A, B) in Theorem 4.1.2 was published by Shiozaki [Shi89, Section III] and later Gao [Gao03]. For the problem of computing (a, b) in Theorem 3.1.2, algorithms in the literature have costs ranging from $n^{3+o(1)}$ down through $n^{1+o(1)}$. A "systematic" Reed–Solomon code represents a polynomial f of degree below $n - 2t$ as the values $(f(\alpha_1), \ldots, f(\alpha_{n-2t})) \in k^{n-2t}$ rather than as the coefficients of f; one needs to look closely at algorithms to see which representation allows faster decoding, although obviously the gap cannot be larger than the cost of converting between representations, i.e., the cost of evaluation and (error-free) interpolation. Finally, there are list-decoding algorithms that can handle more than t errors.

5   Binary-Goppa decoding

The title problem of this paper and of this section, binary-Goppa decoding, is to recover $e, c \in \mathbb{F}_2^n$ from $e + c$, assuming wt $e \leq t$ and $\sum_i c_i A/(x - \alpha_i) \in Gk[x]$. Here $\alpha_1, \ldots, \alpha_n$ are distinct elements of a field k containing $\mathbb{F}_2$; A means $\prod_i (x - \alpha_i)$; and G is a degree-$2t$ element of $k[x]$ with $\gcd\{G, A\} = 1$ (i.e., with $G(\alpha_1), \ldots, G(\alpha_n)$ all nonzero). This section presents an algorithm to solve this problem.

This paper says "k containing $\mathbb{F}_2$" and "$\mathbb{F}_2 \subseteq k$" to mean not just that $\mathbb{F}_2$ is a subset of k, but that $\mathbb{F}_2$ is a subfield of k, i.e., the identity map from $\mathbb{F}_2$ to k is a ring morphism. This guarantees, for example, that the notation $e + c$ has the same meaning whether e, c are viewed as elements of $\mathbb{F}_2^n$ or as elements of k^n.

```
from interpolator import interpolator
from approximant import approximant

def goppa_errors(n,t,k,alpha,G,r):
  alpha,r = list(alpha),list(r)
  assert k.is_field() and k.characteristic() == 2
  assert G.base_ring() == k and G.degree() == 2*t
  assert len(alpha) == n and len(set(alpha)) == n and len(r) == n
  kpoly = G.parent()
  A = kpoly(prod(kpoly([-alpha[j],1]) for j in range(n)))
  Aprime = A.derivative()
  rtwist = [r[i]*Aprime(alpha[i])/G(alpha[i]) for i in range(n)]
  B = interpolator(n,k,alpha,rtwist)
  a,b = approximant(t,k,A,B)
  aprime = a.derivative()
  if a.divides(A):
    if a.divides(G*b-aprime):
      if a*B-b*A == 0 or (a*B-b*A).degree() < n-2*t+a.degree():
        return [k(a(alpha[j]) == 0) for j in range(n)]
```

Algorithm 5.1.1: Algorithm to compute the unique $e \in \mathbb{F}_2^n$ with $\sum_i (r_i - e_i)A/(x - \alpha_i) \in Gk[x]$ and wt $e \leq t$, or None if no such e exists. Here $A = \prod_i (x - \alpha_i) \in k[x]$. Inputs: integer $n \geq 0$; integer $t \geq 0$; field k containing $\mathbb{F}_2$; $(\alpha_1, \ldots, \alpha_n) \in k^n$ with distinct entries; $G \in k[x]$ with $\deg G = 2t$ and each $G(\alpha_j)$ nonzero; $r \in k^n$.

5.1 An algorithm to decode binary Goppa codes

Algorithm 5.1.1 allows any $r \in k^n$ as an input vector, and returns the unique $e \in \mathbb{F}_2^n$ with wt $e \leq t$ such that $\sum_i (r_i - e_i)A/(x - \alpha_i) \in Gk[x]$, or None if no such e exists. The algorithm has three steps:

- Interpolate a polynomial B satisfying $B(\alpha_i) = r_i A'(\alpha_i)/G(\alpha_i)$. Here A' is the derivative of A, so $A'(\alpha_j) = \prod_{i \neq j} (\alpha_j - \alpha_i)$.

- Compute an approximant b/a to B/A at degree t as in Theorem 3.1.2.

- If $A \in ak[x]$ and $Gb - a' \in ak[x]$ and $\deg(aB - bA) < n - 2t + \deg a$, then output e determined by $\{i : e_i = 1\} = \{i : a(\alpha_i) = 0\}$; otherwise output None. Here a' is the derivative of a.

Theorem 5.1.2 says that if e exists then this outputs e, and Theorem 5.1.3 says that if this produces output then e exists. The test $Gb - a' \in ak[x]$ can be skipped for inputs $r \in \mathbb{F}_2^n$ when G is a square; see Section 7.

Theorem 5.1.2 (Goppa decoding). *Let n, t be nonnegative integers. Let k be a field with $\mathbb{F}_2 \subseteq k$. Let $\alpha_1, \ldots, \alpha_n$ be distinct elements of k. Define $A = \prod_i (x - \alpha_i)$. Let G be an element of $k[x]$ such that $\deg G = 2t$ and $\gcd\{G, A\} = 1$. Let B, a, b be elements of $k[x]$ with $\gcd\{a, b\} = 1$, $\deg a \leq t$, and $\deg(aB - bA) < n - t$. Let A', a' be the derivatives of A, a respectively. Let e be an element of $\mathbb{F}_2^n$ such that wt $e \leq t$ and*

$$\sum_i \left(\frac{G(\alpha_i)B(\alpha_i)}{A'(\alpha_i)} - e_i \right) \frac{A}{x - \alpha_i} \in Gk[x].$$

Then $e_i = [a(\alpha_i) = 0]$ for all i; wt $e = \deg a$; $A \in ak[x]$; $Gb - a' \in ak[x]$; and $\deg(aB - bA) < n - 2t + \deg a$.

The notation $[a(\alpha_i) = 0]$ means 1 if $a(\alpha_i) = 0$, else 0.

Proof. Write $c_i = (GB)(\alpha_i)/A'(\alpha_i) - e_i$. By assumption $\sum_i c_i A/(x - \alpha_i) \in Gk[x]$. Write $f = (\sum_i c_i A/(x - \alpha_i))/G$. Then $f \in k[x]$ and $\deg f < n - 2t$, since $\deg A = n$ and $\deg G = 2t$.

Notice that $(Gf)(\alpha_i)/A'(\alpha_i) = c_i$. Indeed,

$$\sum_i c_i A'(\alpha_i) \prod_{j \neq i} \frac{x - \alpha_j}{\alpha_i - \alpha_j} = \sum_i c_i \prod_{j \neq i}(x - \alpha_j) = \sum_i c_i \frac{A}{x - \alpha_i} = Gf,$$

so $c_i A'(\alpha_i) = (Gf)(\alpha_i)$ by Theorem 2.1.2.

Now $(GB - Gf)(\alpha_i)/A'(\alpha_i) = e_i$, so $(B - f)(\alpha_i) = e_i A'(\alpha_i)/G(\alpha_i)$, which is nonzero exactly when $e_i \neq 0$, so $\mathrm{wt}((B - f)(\alpha_1), \dots, (B - f)(\alpha_n)) = \mathrm{wt}\, e \leq t$.

By Theorem 4.1.2, $A \in ak[x]$; $f = B - bA/a$; $\deg(aB - bA) < n - 2t + \deg a$; and $\{i : (B - f)(\alpha_i) \neq 0\} = \{i : a(\alpha_i) = 0\}$. Hence $\{i : e_i \neq 0\} = \{i : a(\alpha_i) = 0\}$. By assumption $e_i \in \mathbb{F}_2$, so $e_i \neq 0$ exactly when $e_i = 1$, so $e_i = [a(\alpha_i) = 0]$. Also, $\mathrm{wt}\, e$ equals the number of roots of a among $\alpha_1, \dots, \alpha_n$, namely $\deg a$ since $A \in ak[x]$.

Finally, say $a(\alpha_i) = 0$. Then $a'(\alpha_i) \neq 0$, and $(A/a)(\alpha_i) = A'(\alpha_i)/a'(\alpha_i)$ by Bernoulli's rule, so

$$1 = e_i = \frac{G(\alpha_i)(B - f)(\alpha_i)}{A'(\alpha_i)} = \frac{G(\alpha_i)(bA/a)(\alpha_i)}{A'(\alpha_i)} = \frac{G(\alpha_i)b(\alpha_i)}{a'(\alpha_i)},$$

so $(Gb - a')(\alpha_i) = 0$. Hence $Gb - a' \in ak[x]$. $\qquad\qquad\qquad\qquad\qquad\square$

Theorem 5.1.3 (checking Goppa decoding). *Let n, t be nonnegative integers. Let k be a field with $\mathbb{F}_2 \subseteq k$. Let $\alpha_1, \dots, \alpha_n$ be distinct elements of k. Define $A = \prod_i(x - \alpha_i)$. Let G be an element of $k[x]$ with $\deg G = 2t$. Let B, a, b be elements of $k[x]$ with $A \in ak[x]$, $\deg(aB - bA) < n - 2t + \deg a$, and $Gb - a' \in ak[x]$, where a' is the derivative of a. Define $e \in \mathbb{F}_2^n$ by $e_i = [a(\alpha_i) = 0]$. Then $\mathrm{wt}\, e = \deg a$ and*

$$\sum_i \left(\frac{G(\alpha_i)B(\alpha_i)}{A'(\alpha_i)} - e_i \right) \frac{A}{x - \alpha_i} \in Gk[x]$$

where A' is the derivative of A.

Proof. First $a \neq 0$ since $0 \neq A \in ak[x]$. Define $f = B - bA/a$. Then $f \in k[x]$ and $\deg f = \deg(aB - bA) - \deg a < n - 2t = n - \deg G$, so $\deg Gf < n$.

Observe that $e_i = (GbA/a)(\alpha_i)/A'(\alpha_i) = (GB - Gf)(\alpha_i)/A'(\alpha_i)$:

- If $a(\alpha_i) = 0$ then $a'(\alpha_i) \neq 0$ and $(A/a)(\alpha_i)/A'(\alpha_i) = 1/a'(\alpha_i)$. Also $Gb - a' \in ak[x]$ so $(Gb)(\alpha_i) = a'(\alpha_i)$). Multiply: $(GbA/a)(\alpha_i)/A'(\alpha_i) = 1 = e_i$.

- If $a(\alpha_i) \neq 0$ then $(GbA/a)(\alpha_i)/A'(\alpha_i) = 0 = e_i$ since $A(\alpha_i) = 0$.

Hence

$$\sum_i \left(\frac{(GB)(\alpha_i)}{A'(\alpha_i)} - e_i \right) \frac{A}{x - \alpha_i} = \sum_i \frac{(Gf)(\alpha_i)}{A'(\alpha_i)} \frac{A}{x - \alpha_i}$$

$$= \sum_i (Gf)(\alpha_i) \prod_{j \neq i} \frac{x - \alpha_i}{\alpha_j - \alpha_i} = Gf \in Gk[x]$$

by Theorem 2.1.2.

To see $\mathrm{wt}\, e = \deg a$: Since A splits into linear factors of the form $x - \alpha_i$, the same is true for a, so $\#\{i : e_i = 1\} = \#\{i : a(\alpha_i) = 0\} = \deg a$. $\qquad\qquad\qquad\square$

5.2 Goppa decoders via Reed–Solomon decoders

Fix $\beta_1, \ldots, \beta_n \in k^*$, and consider the problem of recovering $f \in k[x]$ with $\deg f < n - 2t$ given a vector that agrees with $(\beta_1 f(\alpha_1), \ldots, \beta_n f(\alpha_n))$ on at least $n - t$ positions. Dividing β_j out of the jth position immediately reduces this to the problem considered in Section 4.

   The main point of the proof of Theorem 5.1.2 is that the vectors $c \in k^n$ satisfying $\sum_i c_i A/(x - \alpha_i) \in Gk[x]$ are exactly the vectors $(\beta_1 f(\alpha_1), \ldots, \beta_n f(\alpha_n))$ where $\beta_j = G(\alpha_j)/A'(\alpha_j)$. Any Reed–Solomon decoder can thus be used as a Goppa decoder.

   Algorithm 5.1.1 starts from this approach but streamlines the computation of e, taking advantage of the assumption $e \in \mathbb{F}_2^n$. The critical information coming from the Reed–Solomon decoder is the "error-locator polynomial" a, which is a nonzero constant multiple of $\prod_{i:e_i \neq 0}(x - \alpha_i)$. Knowing the positions of nonzero entries in e immediately reveals e, since each entry of e is either 0 or 1.

   Without the assumption $e \in \mathbb{F}_2^n$, one can compute each e_i in the Reed–Solomon context as $(bA/a)(\alpha_i)$, which is $b(\alpha_i)A'(\alpha_i)/a'(\alpha_i)$ when $a(\alpha_i) = 0$. In the binary-Goppa context one multiplies by $\beta_i = G(\alpha_i)/A'(\alpha_i)$ to obtain $e_i = G(\alpha_i)b(\alpha_i)/a'(\alpha_i)$ when $a(\alpha_i) = 0$.

   Streamlining this to $e_i = 1$ might not seem helpful in Algorithm 5.1.1, since the algorithm checks $Gb - a' \in ak[x]$ anyway, and the obvious way to do this is to check $G(\alpha_i)b(\alpha_i) = a'(\alpha_i)$. However, Section 7 shows that this check can simply be skipped in the following important case: G is a square and the input vector is in $\mathbb{F}_2^n$.

5.3 Binary Goppa codes

The set of $c \in \mathbb{F}_2^n$ satisfying $\sum_i c_i A/(x - \alpha_i) \in Gk[x]$ is called a binary Goppa code; see Definition 5.3.1. As in Section 4.3, elements of the code are called codewords. With this terminology, the problem of binary-Goppa decoding solved above is the problem of recovering a Goppa codeword from a vector that matches the codeword on at least $n - t$ positions, assuming $\deg G = 2t$.

Definition 5.3.1. *Let n be a nonnegative integer. Let k be a field with $\mathbb{F}_2 \subseteq k$. Let $\alpha_1, \ldots, \alpha_n$ be distinct elements of k. Define $A = \prod_i (x - \alpha_i)$. Let G be a nonzero element of $k[x]$ with $\gcd\{G, A\} = 1$. Then $\{c \in \mathbb{F}_2^n : \sum_i c_i A/(x - \alpha_i) \in Gk[x]\}$ is the* **binary Goppa code with k-support $(\alpha_1, \ldots, \alpha_n)$ and Goppa polynomial G.**

   The binary Goppa code with k-support $(\alpha_1, \ldots, \alpha_n)$ and Goppa polynomial G is also the binary Goppa code with K-support $(\alpha_1, \ldots, \alpha_n)$ and Goppa polynomial G (by Theorem 5.4.1 below) if K is an extension field of k, but can be different if K is a superset of k with an incompatible field structure, so, at least formally, it is important for k to be named in the boldfaced phrase in Definition 5.3.1. In any case, this paper's theorems do not rely on Definition 5.3.1; the definition is provided only as context.

5.4 Lower bounds on dimensions of binary Goppa codes

Theorem 5.4.1 rewrites the condition $\sum_i c_i A/(x - \alpha_i) \in Gk[x]$ as the following system of k-linear equations: $\sum_i c_i/G(\alpha_i) = 0$, $\sum_i c_i \alpha_i/G(\alpha_i) = 0$, and so on through $\sum_i c_i \alpha_i^{\deg G - 1}/G(\alpha_i) = 0$. This theorem is from Goppa [Gop70, Section 3], and is used inside the standard method of computing McEliece keys.

   This system of $\deg G$ linear equations over k is equivalent to a system of $m \deg G$ linear equations over $\mathbb{F}_2$ if $\#k = 2^m$. The $\mathbb{F}_2$-vector space of solutions $c \in \mathbb{F}_2^n$ therefore has dimension at least $n - m \deg G$, i.e., at least $2^{n - m \deg G}$ elements. See Section 6.2 for better results in the case $G = g^2$ with squarefree g.

Theorem 5.4.1 (Goppa parity checks). *Let n be a nonnegative integer. Let k be a field. Let $\alpha_1, \ldots, \alpha_n$ be distinct elements of k. Define $A = \prod_i (x - \alpha_i)$. Let G be a nonzero element*

of $k[x]$ with $\gcd\{G, A\} = 1$. Let c be an element of k^n. Then $\sum_i c_i A/(x - \alpha_i) \in Gk[x]$ if and only if $\sum_i c_i \alpha_i^j / G(\alpha_i) = 0$ for all nonnegative integers $j < \deg G$.

Proof. Define $B = \sum_i (c_i/G(\alpha_i)) A/(x - \alpha_i)$ and $C = \sum_i c_i A/(x - \alpha_i)$. Then $B(\alpha_i) = c_i A'(\alpha_i)/G(\alpha_i)$ and $C(\alpha_i) = c_i A'(\alpha_i)$ by Theorem 2.1.2, so $C(\alpha_i) - G(\alpha_i)B(\alpha_i) = 0$, so $C - GB \in Ak[x]$ since $\alpha_1, \ldots, \alpha_n$ are distinct. Note that $\deg B < \deg A$ and $\deg C < \deg A$.

Define $d = \deg G$. If $\deg B < n - d$ then $\deg GB < n = \deg A$ so $C = GB \in Gk[x]$. Conversely, if $C \in Gk[x]$ then $(C/G - B)G = C - GB \in Ak[x]$ so $C/G - B \in Ak[x]$ since $\gcd\{G, A\} = 1$; but $\deg(C/G - B) < \deg A$, so $C/G - B = 0$, so $\deg B = \deg(C/G) < n - \deg G = n - d$.

Define $Q = \sum_i (c_i/G(\alpha_i)) \alpha_i^d A/(x - \alpha_i)$. Then

$$x^d B - Q = \sum_i \frac{c_i}{G(\alpha_i)} (x^d - \alpha_i^d) \frac{A}{x - \alpha_i} = A \sum_i \frac{c_i}{G(\alpha_i)} \sum_{0 \le j < d} x^{d-1-j} \alpha_i^j.$$

One has $\deg Q < n$, so $\deg B < n - d$ if and only if $\deg(x^d B - Q) < n$, i.e., if and only if $\sum_i (c_i/G(\alpha_i)) \sum_{0 \le j < d} x^{d-1-j} \alpha_i^j = 0$, i.e., if and only if $\sum_i (c_i/G(\alpha_i)) \alpha_i^j = 0$ for all j with $0 \le j < d$. Hence $C \in Gk[x]$ if and only if $\sum_i c_i \alpha_i^j / G(\alpha_i) = 0$ for all j with $0 \le j < d$. $\square$

If the formal structure of this paper allowed $k((x^{-1}))$ then one could replace the last paragraph of the proof with the following: $\deg B < n - d$ if and only if $\deg(B/A) < -d$, i.e., if and only if $\sum_i c_i \alpha_i^j / G(\alpha_i) = 0$ for all nonnegative integers $j < d$, since $B/A = \sum_j x^{-j-1} \sum_i c_i \alpha_i^j / G(\alpha_i)$ as in Section 4.2. The proof given above replaces B/A with the approximation $(x^d B - Q)/x^d A$ so as to work entirely with polynomials.

6 Squarefree binary-Goppa decoding

This section explains how to recover $e, c \in \mathbb{F}_2^n$ from $e + c$, assuming $\sum_i c_i A/(x - \alpha_i) \in gk[x]$ and $\mathrm{wt}\, e \le t$. Here $\alpha_1, \ldots, \alpha_n$ are distinct elements of a finite field k containing $\mathbb{F}_2$; A means $\prod_i (x - \alpha_i)$; and g is a squarefree degree-t element of $k[x]$ with $\gcd\{g, A\} = 1$.

The problem of recovering e, c in this section is the same problem as binary-Goppa decoding from Section 5, except that (1) this section requires k to be finite and (2) this section uses a squarefree polynomial g of degree t where Section 5 uses a polynomial G of degree $2t$.

6.1 Replacing g with g^2

The point of this section is Goppa's observation that, for $c \in \mathbb{F}_2^n$ and squarefree $g \in k[x]$, one has $\sum_i c_i A/(x - \alpha_i) \in gk[x]$ if and only if $\sum_i c_i A/(x - \alpha_i) \in g^2 k[x]$. See Theorem 6.1.1.

Recovering $e, c \in \mathbb{F}_2^n$ from $e + c$, assuming $\mathrm{wt}\, e \le t$ and $\sum_i c_i A/(x - \alpha_i) \in gk[x]$, is thus equivalent to recovering $e, c \in \mathbb{F}_2^n$ from $e + c$, assuming $\mathrm{wt}\, e \le t$ and $\sum_i c_i A/(x - \alpha_i) \in g^2 k[x]$, which is simply a matter of setting $G = g^2$ in Section 5.

Theorem 6.1.1 (Goppa squaring). *Let n be a nonnegative integer. Let k be a finite field with $\mathbb{F}_2 \subseteq k$. Let $\alpha_1, \ldots, \alpha_n$ be distinct elements of k. Define $A = \prod_i (x - \alpha_i)$. Let g be a squarefree element of $k[x]$ such that $\gcd\{g, A\} = 1$. Let c be an element of $\mathbb{F}_2^n$. Then $\sum_i c_i A/(x - \alpha_i) \in gk[x]$ if and only if $\sum_i c_i A/(x - \alpha_i) \in g^2 k[x]$.*

Goppa proved Theorem 6.1.1 in [Gop70, Section 4]. The same proof works for all perfect fields of characteristic 2, not just finite fields.

Proof. Write $Z = \prod_{i:c_i=0} (x - \alpha_i)$ and $C = \prod_{i:c_i=1} (x - \alpha_i)$. Then $A = ZC$. By hypothesis $\gcd\{g, A\} = 1$, so $\gcd\{g, Z\} = 1$.

The derivative C' of C is $\sum_{i:c_i=1} C/(x - \alpha_i)$. Hence $\sum_i c_i A/(x - \alpha_i) \in gk[x]$ if and only if $ZC' \in gk[x]$; i.e., if and only if $C' \in gk[x]$. Similarly, $\sum_i c_i A/(x - \alpha_i) \in g^2 k[x]$ if and only if $C' \in g^2 k[x]$. It thus suffices to show that $C' \in gk[x]$ if and only if $C' \in g^2 k[x]$.

Assume that $C' \in gk[x]$. Write C as $\sum_j C_j x^j$. By assumption k is a finite field containing $\mathbb{F}_2$, so $C' = \sum_j C_{2j+1} x^{2j}$; also, C_{2j+1} has a square root $C_{2j+1}^{1/2}$ in k, so $C' = S^2$ where $S = \sum_j C_{2j+1}^{1/2} x^j$. Thus $S^2 \in gk[x]$, implying $S \in gk[x]$ since g is squarefree, implying $C' \in g^2 k[x]$.

Conversely, if $C' \in g^2 k[x]$ then certainly $C' \in gk[x]$. $\square$

6.2   Lower bounds on dimensions of squarefree binary Goppa codes

Recall from Section 5.4 that the vectors $c \in \mathbb{F}_2^n$ satisfying $\sum_i c_i A/(x - \alpha_i) \in Gk[x]$ form an $\mathbb{F}_2$-vector space of dimension at least $n - m \deg G$ when $\#k = 2^m$.

In the case $G = g^2$ with squarefree g, the equivalence in Theorem 6.1.1 between the conditions $\sum_i c_i A/(x - \alpha_i) \in g^2 k[x]$ and $\sum_i c_i A/(x - \alpha_i) \in gk[x]$ immediately produces the conclusion that the dimension is at least $n - m \deg g$. This is a larger bound than $n - m \deg G = n - 2m \deg g$ whenever $\deg g > 0$.

In other words, subject to a requirement of the dimension being at least $n - 2mx$ for a specified positive integer x, one can guarantee decoding x errors by taking any G of degree $2x$, but for $x > 0$ one can make a stronger guarantee of decoding $2x$ errors by taking $G = g^2$ for a squarefree g of degree $2x$. This is the core reason for interest in the latter case.

7   A closer look at binary Goppa codes

The main point of this section is that if the input vector is assumed to be in $\mathbb{F}_2^n$, not merely in k^n, then the test $Gb - a' \in ak[x]$ can be removed from Algorithm 5.1.1 in the case $G = g^2$. See Theorem 7.1.

Theorem 7.1 (checking Goppa decoding for received words in $\mathbb{F}_2^n$). *Let n, t be nonnegative integers. Let k be a field with $\mathbb{F}_2 \subseteq k$. Let $\alpha_1, \ldots, \alpha_n$ be distinct elements of k. Define $A = \prod_i (x - \alpha_i)$. Let g be an element of $k[x]$ such that $\deg g = t$ and $\gcd\{g, A\} = 1$. Let B, a, b be elements of $k[x]$ with $\gcd\{a, b\} = 1$, $\deg a \le t$, $A \in ak[x]$, and $\deg(aB - bA) < n - 2t + \deg a$. Assume that $g(\alpha_i)^2 B(\alpha_i)/A'(\alpha_i) \in \mathbb{F}_2$ for all i, where A' is the derivative of A. Define $e \in \mathbb{F}_2^n$ by $e_i = [a(\alpha_i) = 0]$. Then $g^2 b - a' \in ak[x]$, where a' is the derivative of a. Furthermore $\operatorname{wt} e = \deg a$ and*

$$\sum_i \left(\frac{g(\alpha_i)^2 B(\alpha_i)}{A'(\alpha_i)} - e_i \right) \frac{A}{x - \alpha_i} \in g^2 k[x].$$

Compared to the case $G = g^2$ of Theorem 5.1.3, Theorem 7.1 adds the hypothesis that $g(\alpha_i)^2 B(\alpha_i)/A'(\alpha_i) \in \mathbb{F}_2$, but Theorem 7.1 obtains $g^2 b - a' \in ak[x]$ as a conclusion rather than requiring it as a hypothesis.

Part of the proof of Theorem 7.1 is essentially the calculation in Theorem 5.4.1. This paper's formally verified proofs (see Appendix C) factor the main overlap into shared lemmas. It wouldn't be surprising if Theorem 7.1, or at least part of the proof beyond Theorem 5.4.1, is already in the literature, but various searches and discussions with colleagues have not found a reference.

Proof. By assumption $\prod_i (x - \alpha_i) = A \in ak[x]$. By unique factorization, $a = \lambda \prod_{i \in S} (x - \alpha_i)$ for some $\lambda \in k^*$ and some $S \subseteq \{1, \ldots, n\}$. Now $i \in S$ if and only if $a(\alpha_i) = 0$, so $a = \lambda \prod_{i:a(\alpha_i)=0} (x - \alpha_i)$.

It suffices to show that α_i is a root of $g^2 b - a'$ for each i with $a(\alpha_i) = 0$. Indeed, this implies $g^2 b - a' \in (\prod_{i:a(\alpha_i)=0}(x - \alpha_i))k[x] = ak[x]$. By Theorem 5.1.3, wt $e = \deg a$ and

$$\sum_i \left(\frac{g(\alpha_i)^2 B(\alpha_i)}{A'(\alpha_i)} - e_i \right) \frac{A}{x - \alpha_i} \in g^2 k[x]$$

as claimed.

So fix j with $a(\alpha_j) = 0$; this implies $t \geq 1$ since $t \geq \deg a$. Write $q = a/(x - \alpha_j)$; then $q(\alpha_j) = a'(\alpha_j)$ by Bernoulli's rule. The rest of the proof will show that $(g^2 b)(\alpha_j) = q(\alpha_j)$, so α_j is a root of $g^2 b - a'$ as desired.

Define $r_i = (g^2 B)(\alpha_i)/A'(\alpha_i)$ for each i. By hypothesis $r_i \in \mathbb{F}_2$; i.e., $r_i^2 = r_i$.

For any $\rho \in k[x]$, abbreviate $\rho(x + \alpha_j)$ as $\overline{\rho}$. Define $D = \overline{A}$, and define $\delta_i = \alpha_i - \alpha_j$. Then $\delta_1, \ldots, \delta_n$ are distinct elements of k; $D = \prod_i(x + \alpha_j - \alpha_i) = \prod_i(x - \delta_i)$; and $D'(\delta_i) = A'(\delta_i + \alpha_j) = A'(\alpha_i)$.

Define $R = \sum_i(r_i/g^2(\alpha_i))D/(x - \delta_i)$. Then $\deg R < n$, and, by Theorem 2.1.2, $R(\delta_i) = (r_i/g^2(\alpha_i))D'(\delta_i)$. Substitute $r_i = (g^2 B)(\alpha_i)/A'(\alpha_i)$ and $D'(\delta_i) = A'(\alpha_i)$ to see that $R(\delta_i) = B(\alpha_i)$. The difference $R - \overline{B}$ has each δ_i as a root, so $R - \overline{B} \in Dk[x]$.

Define $Q = \sum_i(r_i/g^2(\alpha_i))\delta_i^{2t}D/(x - \delta_i)$. Then $\deg Q < n$, and

$$x^{2t} R - Q = \sum_i \frac{r_i}{g^2(\alpha_i)}(x^{2t} - \delta_i^{2t})\frac{D}{x - \delta_i} = D \sum_i \frac{r_i}{g^2(\alpha_i)} \sum_{0 \leq s < 2t} x^{2t-1-s}\delta_i^s.$$

Consider any $\varphi \in k[x]$ with $\deg \varphi < 2t$. Write φ_e for the coefficient of x^e in φ. Then $\varphi = \sum_{0 \leq e < 2t} \varphi_e x^e$, so

$$\varphi \frac{x^{2t} R - Q}{D} = \sum_{0 \leq e < 2t} \varphi_e x^e \sum_i \frac{r_i}{g^2(\alpha_i)} \sum_{0 \leq s < 2t} x^{2t-1-s}\delta_i^s$$

$$= \sum_{i,e,s:0 \leq e < 2t, 0 \leq s < 2t} \frac{\varphi_e \delta_i^s r_i}{g^2(\alpha_i)} x^{e+2t-1-s}.$$

The coefficient of x^{2t-1} in $\varphi \cdot (x^{2t} R - Q)/D$ is thus

$$\sum_{i,e,s:0 \leq e < 2t, 0 \leq s < 2t, e+2t-1-s=2t-1} \frac{\varphi_e \delta_i^s r_i}{g^2(\alpha_i)} = \sum_{i,e:0 \leq e < 2t} \frac{\varphi_e \delta_i^e r_i}{g^2(\alpha_i)} = \sum_i \frac{\varphi(\delta_i) r_i}{g^2(\alpha_i)}.$$

More specifically, consider any $h \in k[x]$ with $\deg h \leq t$, define $\varphi = \overline{hq}$, and define $H = \overline{hq} \cdot (x^{2t} R - Q)/D$. Then $\deg \varphi < 2t$ since $\deg q = \deg a - 1 \leq t - 1$, so the coefficient of x^{2t-1} in H is $\sum_i \varphi(\delta_i) r_i/g^2(\alpha_i) = \sum_i(hq)(\alpha_i) r_i/g^2(\alpha_i)$.

Next note that $\deg(aB - bA) < n - 2t + \deg a \leq n - t$, so $\deg((x - \alpha_j)hqB - hbA) = \deg(haB - hbA) < n$, so $\deg((x - \alpha_j)^{2t}hqB - (x - \alpha_j)^{2t-1}hbA) < n + 2t - 1$. Hence $\deg(x^{2t}\overline{hqB} - x^{2t-1}\overline{hbD}) < n + 2t - 1$. Also $\deg Q < n$ so $\deg(\overline{hq}Q) < n + 2t - 1$.

Now rewrite H as

$$x^{2t-1}\overline{hb} + \frac{(x^{2t}\overline{hqB} - x^{2t-1}\overline{hbD}) - \overline{hq}Q}{D} + x^{2t}\overline{hq}\frac{R - \overline{B}}{D}.$$

The second term has degree at most $2t - 2$, so the coefficient of x^{2t-1} in that term is 0. The third term is in $x^{2t}k[x]$, so the coefficient of x^{2t-1} in that term is also 0. The coefficient of x^{2t-1} in H is thus the coefficient of x^{2t-1} in $x^{2t-1}\overline{hb}$; i.e., the coefficient of x^0 in $\overline{hb}$; i.e., $(\overline{hb})(0)$; i.e., $(hb)(\alpha_j)$.

Recap: any $h \in k[x]$ with $\deg h \leq t$ has $(hb)(\alpha_j) = \sum_i(hq)(\alpha_i) r_i/g^2(\alpha_i)$. In particular, $(gb)(\alpha_j) = \sum_i(gq)(\alpha_i) r_i/g^2(\alpha_i) = \sum_i q(\alpha_i) r_i/g(\alpha_i)$, and $(qb)(\alpha_j) = \sum_i q^2(\alpha_i) r_i/g^2(\alpha_i)$.

One has $2 = 0$ in k since k contains $\mathbb{F}_2$, so $(v + w)^2 = v^2 + 2vw + w^2 = v^2 + w^2$ for all $v, w \in k$: in short, squaring maps sums to sums. In particular,

$$(gb)(\alpha_j)^2 = \sum_i \frac{q(\alpha_i)^2 r_i^2}{g(\alpha_i)^2} = \sum_i \frac{q^2(\alpha_i) r_i}{g^2(\alpha_i)} = (qb)(\alpha_j)$$

since $r_i^2 = r_i$. Finally, $b(\alpha_j) \neq 0$ since $\gcd\{a, b\} = 1$, so $(g^2 b)(\alpha_j) = q(\alpha_j)$. $\qquad\square$

8   McEliece decryption

The reader is presumed to be interested specifically in Classic McEliece [BernCLM+17], although without much work one can also cover other versions of the McEliece cryptosystem.

8.1   Ciphertexts

The basic goal of the cryptosystem is for Alice to communicate to Bob a secret vector $e \in \mathbb{F}_2^n$ with wt $e = t$. Alice encodes e as a shorter "ciphertext" $H(e) \in \mathbb{F}_2^{mt}$; shorter means $mt < n$. The function H is determined by Bob's "public key" and has three critical properties listed below.

Section 8.2 explains how Bob "decrypts" $H(e)$, recovering e from $H(e)$ using Bob's "secret key". Meanwhile an attacker sees the ciphertext $H(e)$ and Bob's *public* key, and hopefully has trouble recovering e. One aspect of attacks is within scope of this paper and is covered in Section 8.3 below.

Here are the three critical properties of the function $H : \mathbb{F}_2^n \to \mathbb{F}_2^{mt}$:

- **Linear:** The function is $\mathbb{F}_2$-linear. (This allows the function to be concisely communicated as a matrix. That matrix is Bob's public key.)

- **Goppa:** Each $c \in \mathbb{F}_2^n$ has $H(c) = 0$ if and only if $\sum_i c_i A/(x - \alpha_i) \in gk[x]$. Here k is a field with $\#k = 2^m$, and $\alpha_1, \ldots, \alpha_n, g$ are as in Section 6, as usual with $A = \prod_i (x - \alpha_i)$. (Bob's secret key is $(\alpha_1, \ldots, \alpha_n, g)$. There is a standard public choice of k.)

- **Systematic:** The composition $H \circ \iota : \mathbb{F}_2^{mt} \to \mathbb{F}_2^{mt}$ is the identity map, where ι is the injection $\mathbb{F}_2^{mt} \to \mathbb{F}_2^n$ that simply appends $n - mt$ zeros to the input. In other words, the first $mt \times mt$ block of the matrix is an identity matrix. (Obviously the identity matrix can then be omitted from the public key, saving some space; less obviously, this reduces the cost of optimized decoding from $n^{2+o(1)}$ to $n^{1+o(1)}$.)

For each $k, \alpha_1, \ldots, \alpha_n, g$ there is at most one H satisfying these properties. The obvious way for Bob to construct this H, if it exists, is to convert $\sum_i c_i A/(x - \alpha_i) \in gk[x]$ into a system of $\mathbb{F}_2$-linear equations (a "parity-check matrix") using Theorem 5.4.1, and then row-reduce the equations to obtain systematic form. Conjecturally, this succeeds about 30% of the time. In case of failure, the traditional response is to try again with a new $(\alpha_1, \ldots, \alpha_n, g)$; Chou's "semi-systematic form" options (see [BernCLM+19]) instead apply a limited permutation to $(\alpha_1, \ldots, \alpha_n)$; [AlbCPTT17] had instead applied an arbitrary permutation to $(\alpha_1, \ldots, \alpha_n)$. See [BernCLM+19] for step-by-step algorithms.

8.2   Decryption

Bob decrypts a ciphertext $H(e)$ as follows. Define $c = \iota(H(e)) - e \in \mathbb{F}_2^n$. One has $H(\iota(H(e))) = H(e)$ by the systematic-form property of H, so $H(c) = 0$ by linearity. One then has $\sum_i c_i A/(x - \alpha_i) \in gk[x]$ by the Goppa property of H. Recovering e from $H(e)$ is thus a simple matter of appending $n - mt$ zeros to obtain $\iota(H(e)) = e + c$, and then

recovering $e, c \in \mathbb{F}_2^n$ from $e + c$ by the decoding algorithm from Section 5 (with $G = g^2$ as explained in Section 6). The quantities $\alpha_1, \ldots, \alpha_n, g$ used in the decoding algorithm are available in Bob's secret key.

8.3 Rigidity

At this point one could define a "public-key cryptosystem" consisting of

- a key-generation algorithm that randomly generates a secret key $(\alpha_1, \ldots, \alpha_n, g)$ and the corresponding public key (the matrix representing H),

- an encryption algorithm that takes a "plaintext" e and the public key and outputs the ciphertext $H(e)$, and

- a decryption algorithm that takes $H(e)$ and the secret key and outputs e.

Classic McEliece is actually a "key-encapsulation mechanism" (KEM) in which e is chosen randomly and the final output is the result of applying a "cryptographic hash function" to e. There are also further steps that protect against various types of attacks.

One type of attack is relevant to this paper: "chosen-ciphertext attacks" in which an attacker tries another ciphertext and sees how Bob responds. The protections against these attacks (see [BernCLM+17]) rely critically on Bob recognizing invalid input vectors to the decryption process. An input vector $\sigma \in \mathbb{F}_2^{mt}$ is by definition valid exactly when it is in $\{H(e) : e \in \mathbb{F}_2^n, \mathrm{wt}\, e = t\}$.

One way to recognize invalid input vectors is as follows:

- Feed σ through any decoding algorithm that works for valid inputs. More precisely, apply some function $D : \mathbb{F}_2^{mt} \to \mathbb{F}_2^n$ with the following property: all $e \in \mathbb{F}_2^n$ with $\mathrm{wt}\, e = t$ have $D(H(e)) = e$.

- In all cases, whatever the output $e \in \mathbb{F}_2^n$ is, check that $\mathrm{wt}\, e = t$. If this fails, the input vector is invalid.

- "Reencrypt" to double-check validity of σ: compute $H(e)$ and check whether $H(e) = \sigma$. If this fails, the input vector is invalid.

Handling the matrix for H in the last step incurs similar costs to encryption. Consider, e.g., [Pate19] saying that this "necessitates the inclusion of the public key as part of the private key and increases the running time of decapsulation", although to save space one could instead take time to "regenerate the public key from the private key when needed".

A more efficient approach, already noted in [BernCLM+17, Section 2.5] and used in the software accompanying [BernCLM+17], checks whether $H(e) = \sigma$ "without using quadratic space", and in particular without storing or recomputing the matrix for H. The point is that the following properties are equivalent:

- $\sigma = H(e)$;

- $H(\iota(\sigma)) = H(e)$, by the systematic-form property of H;

- $H(c) = 0$ for $c = \iota(\sigma) - e$, by linearity;

- $\sum_i c_i A/(x - \alpha_i) \in gk[x]$, by the Goppa property of H.

This last condition, checking that $c = \iota(\sigma) - e$ is a codeword, no longer involves H: it is simply some extra polynomial arithmetic, the same type of arithmetic that is being carried out anyway.

A third approach is to inspect the details of decoding, relying not just on Theorem 5.1.2 to decode valid inputs but also Theorem 5.1.3 to identify invalid inputs. Specifically, after interpolating B with $B(\alpha_i)g(\alpha_i)^2/A'(\alpha_i) = \iota(\sigma)_i$ and finding an approximant b/a to B/A at degree t, one checks

- that $\deg a = t$ (this also forces $\deg(aB - bA) < n - 2t + \deg a$, since an approximant by definition has $\deg(aB - bA) < n - t$);

- that $A \in ak[x]$ (i.e., that a has exactly t roots among $\alpha_1, \dots, \alpha_n$); and

- that $g^2 b - a' \in ak[x]$ (i.e., that $g^2 b - a'$ vanishes on each of the roots of a).

If all of these checks succeed then $\text{wt}\, e = t$ and $H(e) = \sigma$ where $e_i = [a(\alpha_i) = 0]$. Otherwise σ is invalid.

It is not clear that the condition $g^2 b - a' \in ak[x]$ is more efficient to evaluate than the condition $\sum_i c_i A/(x - \alpha_i) \in gk[x]$. See generally the discussion of fast "syndrome" computation in [BernCS13].

A fourth approach is to interpolate, find an approximant b/a, check that $\deg a = t$, and check that $A \in ak[x]$, skipping the check that $g^2 b - a' \in ak[x]$. This relies on Theorem 7.1 and the fact that $\iota(\sigma) \in \mathbb{F}_2^n$.

8.4   Robust system design

There are several reasons to recommend the second approach from Section 8.3, the approach taken in Classic McEliece, even if it is not quite as efficient as the fourth approach.

What happens if there's a mistake in the extra logic in this paper leading to Theorem 7.1, or in the handling of invalid inputs in software implementing a decoding algorithm? Appendix C includes a formalization of Theorem 7.1 having a computer-verified proof, but this does not directly address the software question. Software is normally tested on many *valid* inputs; this doesn't provide any assurance that *invalid* inputs are correctly recognized.

A separate reencryption step, whether expressed as testing $H(e) = \sigma$ or more efficiently as testing that $c = \iota(\sigma) - e$ is a codeword, splits the decryption task into two simpler tasks. The task of decoding is to correctly handle valid inputs. The task of reencryption is to reject invalid inputs. Reencryption is redundant if the decoder also rejects invalid inputs, but having the separate reencryption step means that the requirements on the decoder are reduced.

As an illustration of the value of reencryption, consider the efficient chosen-ciphertext attack from Chou [Cho20] breaking both specified versions (namely [AlbCPTT17] and [AlbCPTT19]) of "NTS-KEM", a McEliece variant that skipped reencryption.

Recall that Berlekamp–Massey polynomials are extended-gcd polynomials but with coefficients in reverse order. Reversing polynomials loses information if one does not attach extra information (a "formal degree") to each polynomial: for example, both $3 + x + 4x^2$ and $3x + x^2 + 4x^3$ have the same reversal, namely $4 + x + 3x^2$. The NTS-KEM decoding algorithms are shown in [Cho20] to sometimes find a polynomial ax of degree t when they should instead find a polynomial a of degree $t - 1$. This often leaks information if the attacker modifies a ciphertext $H(e)$ in a way that corresponds to flipping one bit of e.

As further illustrations of how the decoding details matter, [Cho20] identifies bugs (deviations from the specification) in the decoding algorithms in each of the four official NTS-KEM implementations (`ref`, `opt`, `sse2`, `avx2`); these bugs stop the attack from working against one implementation (`ref`), although the attack works against the other three implementations.

Reencrypting the incorrect weight-t error vector obtained from ax would have detected the mismatch with σ and would have stopped this attack. A different way to stop this attack would be to require computer verification of proofs that

- decoding algorithms decode correctly, including cases of weight below t, and

- decoding software correctly implements those algorithms.

Reencryption has the advantage of being easier. Verification has the advantage of also ensuring that valid ciphertexts are handled correctly.

8.5 History

McEliece's original cryptosystem [Mce78] had a different ciphertext shape: the secret message being sent was encoded as some c with $H(c) = 0$ (i.e., some Goppa codeword), and then transmitted as $e + c$ for a secret e with wt $e = t$. Niederreiter [Nie86] introduced the idea of sending just $H(e)$ as a ciphertext, with e as the message. In both [Mce78] and [Nie86], the decoder handled matrices of similar size to the public key.

McEliece started with a generator matrix for the Goppa code, meaning a matrix with row space $\{c \in \mathbb{F}_2^n : \sum_i c_i A/(x - \alpha_i) \in gk[x]\}$. McEliece said that this matrix "could be in canonical, for example row-reduced echelon, form". Row-reduced echelon form is easily compressed into less space than a random matrix, especially if one requires row-reduced echelon form specifically with no skipped columns, i.e., systematic form.

But McEliece didn't use this canonical matrix as the public key: McEliece used a random generator matrix. McEliece also randomly permuted the output positions; this is equivalent to randomly permuting $(\alpha_1, \ldots, \alpha_n)$.

Eventually it was understood that, after permuting $(\alpha_1, \ldots, \alpha_n)$, one can safely use a canonical generator matrix (or, equivalently, a canonical parity-check matrix), such as a systematic matrix. Canteaut and Chabaud [CanC95, page 4, note 1] said that "most of the bits of the plain-text would be revealed" by a systematic generator matrix but that using a random generator matrix "has no cryptographic function". Canteaut and Sendrier [CanS98, pages 188–189] said that the Niederreiter variant "allows a public key in systematic form at no cost for security whereas this would reveal a part of the plaintext in McEliece system". As noted by Overbeck and Sendrier [OveS09, page 98], the partial-plaintext problem is eliminated by various McEliece variants designed for security against chosen-ciphertext attacks: in these variants, the plaintext looks completely random, and the attacker is faced with the problem of finding *all* of the bits of the plaintext.

The fact that one can decrypt using $n^{1+o(1)}$ time and space, including an optimized version of a reencryption step to check $H(e) = \sigma$, appeared in [BernCLM+17]. This relies on systematic form

- to reduce decryption of σ to decoding of $\iota(\sigma)$; and, symmetrically,

- to reduce testing $H(e) = \sigma$ to testing that $\iota(\sigma) - e$ is a codeword.

The first reduction had already appeared in the McEliece context in [BernCS13, Section 6], which in turn says that the choice of $\iota(\sigma)$ as a decoder input was recommended to the authors by Sendrier.

References

[@1998]   — (no editor), *39th annual symposium on foundations of computer science, FOCS '98, November 8–11, 1998, Palo Alto, California, USA*, IEEE Computer Society, 1998. DOI: 10.1109/SFCS.1998. See [GurS98].

[@2000]   — (no editor), *Proceedings of the 32nd annual ACM symposium on theory of computing*, Association for Computing Machinery, New York, 2000. ISBN 1-58113-184-4. See [Bon00].

[AbrMKS22]   Oskar Abrahamsson, Magnus O. Myreen, Ramana Kumar, Thomas Sewell, *Candle: A verified implementation of HOL Light*, in ITP 2022 [AndM22] (2022), 3:1–3:17. DOI: 10.4230/LIPICS.ITP.2022.3. Citations in this document: §C.1, §C.1.

[AlbCPTT17]   Martin Albrecht, Carlos Cid, Kenneth G. Paterson, CJ Tjhai, Martin Tomlinson, *NTS-KEM* (2017); see also newer version [AlbCPTT19]. URL: `https://web.archive.org/web/20180202040600/https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions`. Citations in this document: §8.1, §8.4, §B.2, §B.4, §B.4, §B.4.

[AlbCPTT19]   Martin Albrecht, Carlos Cid, Kenneth G. Paterson, CJ Tjhai, Martin Tomlinson, *NTS-KEM: second round submission* (2019); see also older version [AlbCPTT17]. URL: `https://web.archive.org/web/20200310165056/https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions`. Citations in this document: §8.4, §B.2, §B.4, §B.4, §B.4.

[AndM22]   June Andronick, Leonardo de Moura (editors), *13th international conference on interactive theorem proving, ITP 2022, August 7–10, 2022, Haifa, Israel*, 237, Schloss Dagstuhl—Leibniz-Zentrum für Informatik, 2022. ISBN 978-3-95977-252-5. See [AbrMKS22].

[Berl68]   Elwyn R. Berlekamp, *Algebraic coding theory*, McGraw-Hill, 1968. Citations in this document: §4.4.

[Bern08a]   Daniel J. Bernstein, *Fast multiplication and its applications*, in [BuhS08] (2008), 325–384. URL: `https://cr.yp.to/papers.html#multapps`. Citations in this document: §2.2, §3.2, §4.4.

[Bern08b]   Daniel J. Bernstein, *Reducing lattice bases to find small-height values of univariate polynomials*, in [BuhS08] (2008), 421–446. URL: `https://cr.yp.to/papers.html#smallheight`. Citations in this document: §3.4.

[Bern11a]   Daniel J. Bernstein, *List decoding for binary Goppa codes*, in IWCC 2011 [CheeGLS+11] (2011), 62–80. URL: `https://cr.yp.to/papers.html#goppalist`. DOI: `10.1007/978-3-642-20901-7_4`. Citations in this document: §3.4.

[Bern11b]   Daniel J. Bernstein, *Simplified high-speed high-distance list decoding for alternant codes*, in PQCrypto 2011 [Yan11] (2011), 200–216. URL: `https://cr.yp.to/papers.html#simplelist`. DOI: `10.1007/978-3-642-25405-5_13`. Citations in this document: §3.4.

[Bern20]   Daniel J. Bernstein, *Verified fast formulas for control bits for permutation networks* (2020). URL: `https://cr.yp.to/papers.html#controlbits`. Citations in this document: §2.2.

[BernBD09]   Daniel J. Bernstein, Johannes Buchmann, Erik Dahmen (editors), *Post-quantum cryptography*, Springer, 2009. ISBN 978-3-540-88701-0. DOI: `10.1007/978-3-540-88702-7`. See [OveS09].

[BernCLM+17]   Daniel J. Bernstein, Tung Chou, Tanja Lange, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Wen Wang, *Classic McEliece: conservative code-based cryptography*, "Supporting Documentation" (2017); see also newer version [BernCLM+19]. URL: `https://classic.mceliece.org/nist.html`. Citations in this document: §8, §8.3, §8.3, §8.3, §8.5.

[BernCLM+19]   Daniel J. Bernstein, Tung Chou, Tanja Lange, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Wen Wang, *Classic McEliece: conservative code-based cryptography*, "Supporting Documentation" (2019); see also older version [BernCLM+17]. URL: https://classic.mceliece.org/nist.html. Citations in this document: §8.1, §8.1.

[BernCS13]   Daniel J. Bernstein, Tung Chou, Peter Schwabe, *McBits: fast constant-time code-based cryptography*, in [BertC13] (2013), 250–272. URL: https://cr.yp.to/papers.html#mcbits. DOI: 10.1007/978-3-642-40349-1_15. Citations in this document: §1.1, §8.3, §8.5.

[BernLP11]   Daniel J. Bernstein, Tanja Lange, Christiane Peters, *Wild McEliece*, in SAC 2010 [BirGS11] (2011), 143–158. URL: https://eprint.iacr.org/2010/410. DOI: 10.1007/978-3-642-19574-7_10. Citations in this document: §1.1.

[BernY19]   Daniel J. Bernstein, Bo-Yin Yang, *Fast constant-time gcd computation and modular inversion*, IACR Transactions on Cryptographic Hardware and Embedded Systems **2019.3** (2019), 340–398. URL: https://gcd.cr.yp.to/papers.html. DOI: 10.46586/tches.v2019.i3.340-398. Citations in this document: §3.2, §3.2, §4.4.

[BertC13]   Guido Bertoni, Jean-Sébastien Coron (editors), *Cryptographic hardware and embedded systems—CHES 2013—15th international workshop, Santa Barbara, CA, USA, August 20–23, 2013, proceedings*, 8086, Springer, 2013. ISBN 978-3-642-40348-4. DOI: 10.1007/978-3-642-40349-1. See [BernCS13].

[BhaPTY03]   Vijay K. Bhargava, H. Vincent Poor, Vahid Tarokh, Seokho Yoon (editors), *Communications, information and network security. With a foreword by Richard E. Blahut*, Springer, 2003. ISBN 978-1-4020-7251-2; 978-1-4419-5318-6; 978-1-4757-3789-9. DOI: 10.1007/978-1-4757-3789-9. See [Gao03].

[BirGS11]   Alex Biryukov, Guang Gong, Douglas R. Stinson (editors), *Selected areas in cryptography—17th international workshop, SAC 2010, Waterloo, Ontario, Canada, August 12–13, 2010, revised selected papers*, Lecture Notes in Computer Science, 6544, Springer, 2011. DOI: 10.1007/978-3-642-19574-7. See [BernLP11].

[BlaH20]   Jasmin Blanchette, Catalin Hritcu (editors), *Proceedings of the 9th ACM SIGPLAN international conference on certified programs and proofs, CPP 2020, New Orleans, LA, USA, January 20–21, 2020*, ACM, 2020. ISBN 978-1-4503-7097-4. DOI: 10.1145/3372885. See [Mat20].

[Bon00]   Dan Boneh, *Finding smooth integers in short intervals using CRT decoding*, in STOC 2000 [@2000] (2000), 265–272; see also newer version [Bon02]. DOI: 10.1145/335305.335337.

[Bon02]   Dan Boneh, *Finding smooth integers in short intervals using CRT decoding*, Journal of Computer and System Sciences **64** (2002), 768–784; see also older version [Bon00]. ISSN 0022-0000. URL: https://crypto.stanford.edu/~dabo/abstracts/CRTdecode.html. DOI: 10.1006/jcss.2002.1827. Citations in this document: §3.4.

[BraCPR22] Martin Brain, Carlos Cid, Rachel Player, Wrenna Robson, *Verifying Classic McEliece: examining the role of formal methods in post-quantum cryptography standardisation*, in CBCrypto 2022 [Den23] (2022), 21–36. URL: https://eprint.iacr.org/2023/010. DOI: 10.1007/978-3-031-29689-5_2. Citations in this document: §C.

[Bre81] Claude Brezinski, *The long history of continued fractions and Padé approximants*, in [BruR81] (1981), 1–27. DOI: 10.1007/BFb0095574. Citations in this document: §3.4.

[BruR81] Marcel G. de Bruin, Herman van Rossum (editors), *Padé approximation and its applications, Amsterdam 1980, proceedings of a conference held in Amsterdam, the Netherlands, October 29–31, 1980*, Lecture Notes in Mathematics, 888, Springer, 1981. ISSN 0075-8434. DOI: 10.1007/BFb0095573. See [Bre81].

[BuhS08] Joe P. Buhler, Peter Stevenhagen (editors), *Surveys in algorithmic number theory*, Mathematical Sciences Research Institute Publications, 44, Cambridge University Press, New York, 2008. URL: https://www.math.leidenuniv.nl/~psh/ANTproc/filesheet.shtml. See [Bern08a], [Bern08b].

[CanC95] Anne Canteaut, Florent Chabaud, *Improvements of the attacks on cryptosystems based on error-correcting codes*, report LIENS-95-21 (1995). URL: https://www.di.ens.fr/reports/1995/liens-95-21.A4.pdf. Citations in this document: §8.5.

[CanS98] Anne Canteaut, Nicolas Sendrier, *Cryptanalysis of the original McEliece cryptosystem*, in Asiacrypt '98 [OhtP98] (1998), 187–199. URL: https://www.rocq.inria.fr/secret/Anne.Canteaut/Publications/Canteaut_Sendrier98.pdf. DOI: 10.1007/3-540-49649-1_16. Citations in this document: §8.5.

[CheeGLS+11] Yeow Meng Chee, Zhenbo Guo, San Ling, Fengjing Shao, Yuansheng Tang, Huaxiong Wang, Chaoping Xing (editors), *Coding and cryptology—third international workshop, IWCC 2011, Qingdao, China, May 30–June 3, 2011, proceedings*, Lecture Notes in Computer Science, 6639, Springer, 2011. ISBN 978-3-642-20900-0. DOI: 10.1007/978-3-642-20901-7. See [Bern11a].

[ChenC21] Ming-Shing Chen, Tung Chou, *Classic McEliece on the ARM Cortex-M4*, IACR Transactions on Cryptographic Hardware and Embedded Systems **2021.3** (2021), 125–148. URL: https://tungchou.github.io/papers/cm-m4.pdf. DOI: 10.46586/tches.v2021.i3.125-148. Citations in this document: §1.1.

[Cho17] Tung Chou, *McBits revisited*, in CHES 2017 [FisH17] (2017), 213–231; see also newer version [Cho18]. URL: https://tungchou.github.io/papers/mcbits_revisited.pdf. DOI: 10.1007/978-3-319-66787-4_11.

[Cho18] Tung Chou, *McBits revisited: toward a fast constant-time code-based KEM*, Journal of Cryptographic Engineering **8** (2018), 95–107; see also older version [Cho17]. DOI: 10.1007/s13389-018-0186-9. Citations in this document: §1.1.

[Cho20]   Tung Chou, *An IND-CCA2 attack against the 1st- and 2nd-round versions of NTS-KEM*, in SecITC 2020 [MaiOS21] (2020), 165–184. URL: https://tungchou.github.io/papers/ntskem_cca2.pdf. DOI: 10.1007/978-3-030-69255-1_11. Citations in this document: §8.4, §8.4, §8.4, §B.2, §B.4.

[CohH15]   Henry Cohn, Nadia Heninger, *Ideal forms of Coppersmith's theorem and Guruswami-Sudan list decoding*, Advances in Mathematics of Communications **9** (2015), 311–339. URL: https://arxiv.org/abs/1008.1284. DOI: 10.3934/amc.2015.9.311. Citations in this document: §3.4.

[Com01]   Douglas E. Comer, *How to criticize computer scientists: or, avoiding ineffective deprecation and making insults more pointed* (2001). URL: https://web.archive.org/web/20010111213900/https://www.cs.purdue.edu/homes/dec/essay.criticize.html. Citations in this document: §B.2.

[Den23]   Jean-Christophe Deneuville (editor), *Code-based cryptography—10th international workshop, CBCrypto 2022, Trondheim, Norway, May 29–30, 2022, revised selected papers*, 13839, Springer, 2023. ISBN 978-3-031-29688-8. DOI: 10.1007/978-3-031-29689-5. See [BraCPR22].

[Dor87]   Jean Louis Dornstetter, *On the equivalence between Berlekamp's and Euclid's algorithms*, IEEE Transactions on Information Theory **33** (1987), 428–431. DOI: 10.1109/TIT.1987.1057299. Citations in this document: §4.4.

[Emd13]   Peter van Emde Boas, *The correspondence between Donald E. Knuth and Peter van Emde Boas on priority deques during the spring of 1977* (2013). URL: https://staff.fnwi.uva.nl/p.vanemdeboas/knuthnote.pdf. Citations in this document: §B.1.

[Euc300BC]   Euclid, *Elements*, about 300 B.C. URL: https://www.claymath.org/library/historical/euclid/files/elem.7.2.html. Citations in this document: §3.4.

[FisH17]   Wieland Fischer, Naofumi Homma (editors), *Cryptographic hardware and embedded systems—CHES 2017—19th international conference, Taipei, Taiwan, September 25–28, 2017, proceedings*, 10529, Springer, 2017. ISBN 978-3-319-66786-7. DOI: 10.1007/978-3-319-66787-4. See [Cho17].

[For65a]   G. David Forney, Jr., *Concatenated codes* (1965). URL: https://dspace.mit.edu/bitstream/handle/1721.1/4303/RLE-TR-440-04743368.pdf. Citations in this document: §4.4.

[For65b]   G. David Forney, Jr., *On decoding BCH codes*, IEEE Transactions on Information Theory **11** (1965), 549–557. DOI: 10.1109/TIT.1965.1053825. Citations in this document: §4.4.

[Gao03]   Shuhong Gao, *A new algorithm for decoding Reed-Solomon codes*, in [BhaPTY03] (2003), 55–68. URL: https://www.math.clemson.edu/~sgao/papers/RS.pdf. DOI: 10.1007/978-1-4757-3789-9_5. Citations in this document: §4.4.

[Gau1801]   Carl Friedrich Gauss, *Disquisitiones arithmeticae*, 1801. URL: https://archive.org/details/disquisitionesa00gaus. Citations in this document: §3.4.

[GhoV14]   Santosh Ghosh, Ingrid Verbauwhede, *BLAKE-512-based 128-bit CCA2 secure timing attack resistant McEliece cryptoprocessor*, IEEE Transactions on Computers **63** (2014), 1124–1133. URL: https://www.esat.kuleuven.be/cosic/publications/article-2447.pdf. DOI: 10.1109/TC.2012.271. Citations in this document: §1.1.

[Gop70]   Valerii D. Goppa, *A new class of linear correcting codes*, Problemy Peredachi Informatsii **6** (1970), 24–30. URL: http://www.mathnet.ru/links/95a131fb57f5ed88dd732c324798a36a/ppi1748.pdf. Citations in this document: §1.1, §5.4, §6.1.

[GorZ61]   Daniel Gorenstein, Neal Zierler, *A class of error-correcting codes in p^m symbols*, Journal of the Society for Industrial and Applied Mathematics **9** (1961), 207–214. URL: https://epubs.siam.org/doi/10.1137/0109020. DOI: 10.1137/0109020. Citations in this document: §4.4.

[GurS98]   Venkatesan Guruswami, Madhu Sudan, *Improved decoding of Reed-Solomon and algebraic-geometry codes*, in FOCS 1998 [@1998] (1998), 28–39; see also newer version [GurS99]. URL: https://madhu.seas.harvard.edu/papers.html. Citations in this document: §1.1.

[GurS99]   Venkatesan Guruswami, Madhu Sudan, *Improved decoding of Reed-Solomon and algebraic-geometry codes*, IEEE Transactions on Information Theory **45** (1999), 1757–1767; see also older version [GurS99]. ISSN 0018-9448. URL: https://madhu.seas.harvard.edu/papers.html. DOI: 10.1109/18.782097.

[Har96]   John Harrison, *HOL Light: A tutorial introduction*, in FMCAD 1996 [SriC96] (1996), 265–269. DOI: 10.1007/BFb0031814. Citations in this document: §C.

[HeiHT92]   G. H. L. M. Heideman, Fokke W. Hoeksema, Henk E. P. Tattje (editors), *Proceedings of the 13th symposium on information theory in the Benelux*, Werkgemeenschap voor Informatie- en Communicatietheorie, 1992. See [PreBGV92].

[Jus76]   Jørn Justesen, *On the complexity of decoding Reed–Solomon codes*, IEEE Transactions on Information Theory **22** (1976), 237–238. DOI: 10.1109/TIT.1976.1055516. Citations in this document: §4.4.

[Knu74]   Donald E. Knuth, *Structured programming with go to statements*, Computing Surveys **6** (1974), 261–301. DOI: 10.1145/356635.356640. Citations in this document: §1.1.

[Knu97]   Donald E. Knuth, *The art of computer programming, volume 2: seminumerical algorithms*, 3rd edition, Addison-Wesley, 1997. ISBN 0-201-89684-2. Citations in this document: §3.4.

[Kro1881]   Leopold Kronecker, *Zur Theorie der Elimination einer Variablen aus zwei algebraischen Gleichungen*, Monatsberichte der Königlich Preussischen Akademie der Wissenschaften zu Berlin (1881). URL: https://archive.org/details/werkehrsgaufvera02kronuoft/page/114/mode/2up. Citations in this document: §3.4.

[Lag1773]   Joseph-Louis Lagrange, *Recherches d'arithmétique*, Nouveaux Mémoires de l'Académie royale des Sciences et Belles-Lettres de Berlin (1773). URL: https://gallica.bnf.fr/ark:/12148/bpt6k229222d/f696. Citations in this document: §3.4.

[Lag1776]   Joseph-Louis Lagrange, *Sur l'usage des fractions continues dans le calcul intégral*, Nouveaux Mémoires de l'Académie royale des Sciences et Belles-Lettres de Berlin (1776). URL: https://gallica.bnf.fr/ark:/12148/bpt6k229223s/f303. Citations in this document: §3.4.

[MaiOS21]   Diana Maimut, Andrei-George Oprina, Damien Sauveron (editors), *Innovative security solutions for information technology and communications—13th international conference, SecITC 2020, Bucharest, Romania, November 19–20, 2020, revised selected papers*, 12596, Springer, 2021. ISBN 978-3-030-69254-4. DOI: 10.1007/978-3-030-69255-1. See [Cho20].

[Mas69]   James Massey, *Shift-register synthesis and BCH decoding*, IEEE Transactions on Information Theory **15** (1969), 122–127. ISSN 0018-9448. DOI: 10.1109/TIT.1969.1054260. Citations in this document: §4.4.

[Mat20]   The mathlib community, *The Lean mathematical library*, in CPP 2020 [BlaH20] (2020), 367–381. URL: https://arxiv.org/abs/1910.09336. DOI: 10.1145/3372885.3373824. Citations in this document: §C.

[Mce78]   Robert J. McEliece, *A public-key cryptosystem based on algebraic coding theory*, JPL DSN Progress Report (1978), 114–116. URL: https://ipnpr.jpl.nasa.gov/progress_report2/42-44/44N.PDF. Citations in this document: §1.1, §8.5, §8.5.

[Mil75]   William H. Mills, *Continued fractions and linear recurrences*, Mathematics of Computation **29** (1975), 173–180. URL: https://www.ams.org/journals/mcom/1975-29-129/S0025-5718-1975-0369276-7/. DOI: 10.1090/S0025-5718-1975-0369276-7. Citations in this document: §4.4.

[MouU21]   Leonardo de Moura, Sebastian Ullrich, *The Lean 4 theorem prover and programming language*, in CADE 28 [PlaS21] (2021), 625–635. DOI: 10.1007/978-3-030-79876-5_37. Citations in this document: §C, §C.1.

[Nie86]   Harald Niederreiter, *Knapsack-type cryptosystems and algebraic coding theory*, Problems of Control and Information Theory **15** (1986), 159–166. Citations in this document: §8.5, §8.5.

[OhtP98]   Kazuo Ohta, Dingyi Pei (editors), *Advances in cryptology—ASIACRYPT'98: proceedings of the international conference on the theory and application of cryptology and information security held in Beijing*, Lecture Notes in Computer Science, 1514, Springer, 1998. ISBN 3-540-65109-8. DOI: 10.1007/3-540-49649-1. See [CanS98].

[Orm19]   Tavis Ormandy, *Issue 1804: cryptoapi: SymCrypt modular inverse algorithm* (2019). URL: https://bugs.chromium.org/p/project-zero/issues/detail?id=1804. Citations in this document: §B.2.

[OveS09]   Raphael Overbeck, Nicolas Sendrier, *Code-based cryptography*, in [BernBD09] (2009), 95–145. DOI: 10.1007/978-3-540-88702-7_4. Citations in this document: §1.1, §8.5.

[Pad1892]   Henri Padé, *Sur la représentation approchée d'une fonction par des fractions rationnelles*, Annales scientifiques de l'École normale supérieure **9** (1892), 3–93. URL: `https://web.archive.org/web/20180718235117/http://www.numdam.org/article/ASENS_1892_3_9__S3_0.pdf`. DOI: `10.24033/asens.378`. Citations in this document: §3.4.

[Pate19]   Kenneth G. Paterson, *New version of NTS-KEM* (2019). URL: `https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/Gf5ucjoDok4/m/YciNWTraAwAJ`. Citations in this document: §8.3.

[Patt75]   Nicholas J. Patterson, *The algebraic decoding of Goppa codes*, IEEE Transactions on Information Theory **21** (1975), 203–207. DOI: `10.1109/TIT.1975.1055350`. Citations in this document: §1.1.

[Pet60]   W. Wesley Peterson, *Encoding and error-correction procedures for the Bose-Chaudhuri codes*, Transactions of the Institute of Radio Engineers **6** (1960), 459–470. DOI: `10.1109/TIT.1960.1057586`. Citations in this document: §4.4.

[PlaS21]   André Platzer, Geoff Sutcliffe (editors), *Automated deduction—CADE 28—28th international conference on automated deduction, virtual event, July 12–15, 2021, proceedings*, 12699, Springer, 2021. ISBN 978-3-030-79875-8. DOI: `10.1007/978-3-030-79876-5`. See [MouU21].

[Pra62]   Eugene Prange, *The use of information sets in decoding cyclic codes*, IRE Transactions on Information Theory **IT-8** (1962), S5–S9. DOI: `10.1109/TIT.1962.1057777`. Citations in this document: §4.4.

[PreBGV92] Bart Preneel, Antoon Bosselaers, René Govaerts, Joos Vandewalle, *A software implementation of the McEliece public-key cryptosystem*, in [HeiHT92] (1992), 119–126. URL: `https://www.esat.kuleuven.be/cosic/publications/article-267.pdf`. Citations in this document: §1.1.

[ReeS60]   Irving S. Reed, Gustave Solomon, *Polynomial codes over certain finite fields*, Journal of the Society for Industrial and Applied Mathematics **8** (1960), 300–304. URL: `https://epubs.siam.org/doi/10.1137/0108018`. DOI: `10.1137/0108018`. Citations in this document: §4.4.

[Sar77]   Dilip V. Sarwate, *On the complexity of decoding Goppa codes*, IEEE Transactions on Information Theory **23** (1977), 515–516. URL: `https://core.ac.uk/download/pdf/158319337.pdf`. DOI: `10.1109/TIT.1977.1055732`. Citations in this document: §4.4.

[Shi89]   Akira Shiozaki, *Decoding of redundant residue polynomial codes using Euclid's algorithm*, IEEE Transactions on Information Theory **34** (1989), 1351–1354. DOI: `10.1109/18.21269`. Citations in this document: §4.4.

[SriC96]   Mandayam K. Srivas, Albert John Camilleri (editors), *Formal methods in computer-aided design, first international conference, FMCAD '96, Palo Alto, California, USA, November 6–8, 1996, proceedings*, Lecture Notes in Computer Science, 1166, Springer, 1996. ISBN 3-540-61937-2. DOI: `10.1007/BFB0031795`. See [Har96].

[Ste1585]   Simon Stevin, *L'arithmétique*, Imprimerie de Christophle Plantin, 1585. URL: https://web.archive.org/web/20190430054513/http://www.dwc.knaw.nl/pub/bronnen/Simon_Stevin-[II_B]_The_Principal_Works_of_Simon_Stevin,_Mathematics.pdf. Citations in this document: §3.4.

[Stra69]   Volker Strassen, *Gaussian elimination is not optimal*, Numerische Mathematik **13** (1969), 354–356. ISSN 0029-599X. DOI: 10.1007/BF02165411. Citations in this document: §4.4.

[Stru63]   Dirk J. Struik, *The origin of L'Hôpital's rule*, The Mathematics Teacher **56** (1963), 257–260. URL: https://www.jstor.org/stable/27956806. DOI: 10.5951/MT.56.4.0257. Citations in this document: §A.21.

[Sud97]   Madhu Sudan, *Decoding of Reed Solomon codes beyond the error-correction bound*, Journal of Complexity **13** (1997), 180–193. ISSN 0885-064X. URL: https://madhu.seas.harvard.edu/papers.html. DOI: 10.1006/jcom.1997.0439. Citations in this document: §1.1.

[SugKHN75]   Yasuo Sugiyama, Masao Kasahara, Shigeichi Hirasawa, Toshihiko Namekawa, *A method for solving key equation for decoding Goppa codes*, Information and Control **27** (1975), 87–99. DOI: 10.1016/S0019-9958(75)90090-X. Citations in this document: §4.4.

[The20]   The Sage Developers (editor), *SageMath, the Sage Mathematics Software System (Version 9.2)*, 2020. URL: https://www.sagemath.org. Citations in this document: §1.2.

[Til93]   Henk C. A. van Tilborg, *Coding theory, a first course*, 1993. URL: https://www.win.tue.nl/~henkvt/images/CODING.pdf. Citations in this document: §1.1.

[War1779]   Edward Waring, *Problems concerning interpolations*, Philosophical Transactions of the Royal Society **69** (1779), 59–67. URL: https://royalsocietypublishing.org/doi/pdf/10.1098/rstl.1779.0008. DOI: 10.1098/rstl.1779.0008. Citations in this document: §2.1.

[WelS79]   Lloyd R. Welch, Robert A. Scholtz, *Continued fractions and Berlekamp's algorithm*, IEEE Transactions on Information Theory **25** (1979), 19–27. DOI: 10.1109/TIT.1979.1055987. Citations in this document: §4.4.

[Win19]   Davey Winder, *Warning: Google researcher drops Windows 10 zero-day security bomb* (2019). URL: https://www.forbes.com/sites/daveywinder/2019/06/12/warning-windows-10-crypto-vulnerability-outed-by-google-researcher-before-microsoft-can-fix-it/. Citations in this document: §B.2.

[Yan11]   Bo-Yin Yang (editor), *Post-quantum cryptography—4th international workshop, PQCrypto 2011, Taipei, Taiwan, November 29–December 2, 2011, proceedings*, 7071, Springer, 2011. ISBN 978-3-642-25404-8. DOI: 10.1007/978-3-642-25405-5. See [Bern11b].

A   Polynomials

This appendix reviews the definition of the polynomial ring $k[x]$ over a field k, and the properties of polynomials used in this paper.

A.1   Commutative rings

A **commutative ring** is a set R with elements $0, 1 \in R$, a unary operation $- : R \to R$, and binary operations $+, \cdot : R \times R \to R$ satisfying the identities $r+s = s+r$; $r+(s+t) = (r+s)+t$; $r + (-r) = 0$; $0 + r = r$; $r \cdot s = s \cdot r$; $r \cdot (s \cdot t) = (r \cdot s) \cdot t$; $r \cdot (s+t) = (r \cdot s) + (r \cdot t)$; $1 \cdot r = r$.

These identities imply all of the identities satisfied by $\mathbb{Z}$, the set of integers with its usual $0, 1, -, +, \cdot$.

Normally $r \cdot s$ is abbreviated rs, and $r + (-s)$ is abbreviated $r - s$.

A.2   Ring morphisms

A **ring morphism from R to S**, where R and S are commutative rings, is a function from R to S preserving $0, 1, -, +, \cdot$: i.e., a function $\varphi : R \to S$ with $\varphi(0) = 0$, $\varphi(1) = 1$, $\varphi(-r) = -\varphi(r)$, $\varphi(r + s) = \varphi(r) + \varphi(s)$, and $\varphi(rs) = \varphi(r)\varphi(s)$.

This is the universal-algebra definition of a ring morphism. This is equivalent to a shorter definition that omits the conditions $\varphi(0) = 0$ and $\varphi(-r) = -\varphi(r)$.

A ring morphism maps every $0, 1, -, +, \cdot$ formula in the inputs to the same formula in the outputs: e.g., $\varphi(r + st) = \varphi(r) + \varphi(s)\varphi(t)$ and $\varphi(\sum_i r_i) = \sum_i \varphi(r_i)$.

A.3   Multiples

Let R be a commutative ring. The notation uR, for $u \in R$, means the set $\{uq : q \in R\}$. The notation $uR + vR$, for $u, v \in R$, means the set $\{uq + vr : q, r \in R\}$.

A.4   Units

The notation R^* means $\{u \in R : 1 \in uR\}$; i.e., $u \in R^*$ exactly when some $v \in R$ satisfies $uv = 1$. The elements of R^* are called the **units of R**.

A.5   Fields

One calls R a **field** if $R^* = \{u \in R : u \neq 0\}$. In other words, an element of a field is a unit and if only if it is nonzero.

For example, the set $\{0, 1\}$ with $-, +, \cdot$ defined as arithmetic modulo 2 is a field, denoted $\mathbb{F}_2$. As another example, the set $\mathbb{Q}$ of rational numbers with its usual $0, 1, -, +, \cdot$ is a field.

A.6   Vector spaces

Let k be a field. A **k-vector space** is a set V with an element 0, a unary operation $-$, a binary operation $+$, and, for each $\alpha \in k$, a unary operation $v \mapsto \alpha \cdot v$ such that $v + w = w + v$; $u + (v + w) = (u + v) + w$; $0 + v = v$; $v + (-v) = 0$; $1 \cdot v = v$; $\alpha \cdot (v + w) = \alpha \cdot v + \alpha \cdot w$; $(\alpha\beta) \cdot v = \alpha \cdot (\beta \cdot v)$; and $(\alpha + \beta) \cdot v = (\alpha \cdot v) + (\beta \cdot v)$ for all $u, v, w \in V$ and $\alpha, \beta \in k$.

A.7   The standard n-dimensional vector space

Let n be a nonnegative integer. The set $k^n = \{(v_0, v_1, \ldots, v_{n-1}) : v_0, v_1, \ldots, v_{n-1} \in k\}$ is a k-vector space under the following operations:

- 0 is $(0, 0, \ldots, 0)$.

- $-(v_0, v_1, \ldots, v_{n-1})$ is $(-v_0, -v_1, \ldots, -v_{n-1})$.

- $(v_0, v_1, \ldots, v_{n-1}) + (w_0, w_1, \ldots, w_{n-1})$ is $(v_0 + w_0, v_1 + w_1, \ldots, v_{n-1} + w_{n-1})$.

- $\alpha \cdot (v_0, v_1, \ldots, v_{n-1})$ is $(\alpha v_0, \alpha v_1, \ldots, \alpha v_{n-1})$.

A.8 Linear maps

Let k be a field, and let V, W be k-vector spaces. A **k-linear map from V to W** is a function from V to W preserving $0, -, +, \cdot$: i.e., a function φ satisfying $\varphi(0) = 0$, $\varphi(-v) = -\varphi(v)$, $\varphi(u + v) = \varphi(u) + \varphi(v)$, and $\varphi(\alpha \cdot v) = \alpha \cdot \varphi(v)$ for all $u, v \in V$ and all $\alpha \in k$.

This is the universal-algebra definition of a k-linear map as a k-vector-space morphism. This is equivalent to a shorter definition that omits the conditions $\varphi(0) = 0$ and $\varphi(-v) = -\varphi(v)$.

If $n, m \in \mathbb{Z}$ with $n > m \geq 0$ then any k-linear map from k^n to k^m must map some nonzero input to zero.

A.9 Polynomials

Let k be a field. By definition $k[x]$ is the set of vectors $(f_0, f_1, \dots)$ with all nonnegative integers as indices, $f_i \in k$ for each nonnegative integer i, and $\{i : f_i \neq 0\}$ finite.

If one drops the requirement that $\{i : f_i \neq 0\}$ is finite then one obtains the power-series ring $k[[x]]$, but the reader can safely focus on $k[x]$ for this paper.

A.10 The ring structure of polynomials

The set $k[x]$ is a commutative ring under the following operations:

- 0 is the vector $(0, 0, \dots)$.

- 1 is the vector $(1, 0, \dots)$.

- Negation maps $(f_0, f_1, \dots)$ to $(-f_0, -f_1, \dots)$.

- Addition maps $(f_0, f_1, \dots), (g_0, g_1, \dots)$ to $(f_0 + g_0, f_1 + g_1, \dots)$.

- Multiplication maps $(f_0, f_1, f_2, f_3, \dots), (g_0, g_1, g_2, g_3, \dots)$ to the "convolution" vector $(f_0 g_0, f_0 g_1 + f_1 g_0, f_0 g_2 + f_1 g_1 + f_2 g_0, f_0 g_3 + f_1 g_2 + f_2 g_1 + f_3 g_0, \dots)$.

A.11 The k-algebra structure of polynomials

The map $\alpha \mapsto (\alpha, 0, 0, \dots)$ from k to $k[x]$ is a ring morphism. This map is injective, so one can view k as a subset of $k[x]$.

A.12 Units of $k[x]$

The units of $k[x]$ are exactly the elements $(\alpha, 0, 0, \dots)$ where $\alpha \in k^*$.

A.13 The k-vector structure of polynomials

The set $k[x]$ is a vector space under the following operations: $0, -, +$ are as defined above; $\alpha \cdot (f_0, f_1, \dots)$, for $\alpha \in k$, is defined as $(\alpha f_0, \alpha f_1, \dots)$.

This k-vector structure matches the k-algebra structure: $(\alpha f_0, \alpha f_1, \dots)$ is the same as the product $(\alpha, 0, 0, \dots)(f_0, f_1, \dots)$.

A.14 Powers of x

The vector $(0, 1, 0, \dots) \in k[x]$ is abbreviated x. One then has $x^0 = (1, 0, 0, \dots) = 1$, $x^1 = (0, 1, 0, \dots)$, $x^2 = (0, 0, 1, \dots)$, etc.

Any $f = (f_0, f_1, \dots) \in k[x]$ equals the finite sum $\sum_{i : f_i \neq 0} f_i x^i$. One can also write f as the infinite sum $\sum_{i \geq 0} f_i x^i$; only finitely many terms here are nonzero.

A.15   Coefficients

If $f = (f_0, f_1, \dots) \in k[x]$ and $i \in \mathbb{Z}$ then the **coefficient of x^i in f** means the entry f_i for $i \geq 0$, or 0 for $i < 0$. (The case $i < 0$ arises in the proof of Theorem 3.1.2 if $t > \deg A$.)

One conventionally hides the formal definition of a polynomial as a vector: rather than constructing a polynomial f as $(f_0, f_1, \dots)$ and referring to f_i as the entry at position i in f, one constructs f as $\sum_i f_i x^i$ and refers to f_i as the coefficient of x^i in f.

A.16   Degree

If $f = (f_0, f_1, \dots) \in k[x]$ then the **degree of f**, written $\deg f$, is $-\infty$ for $f = 0$, and otherwise the largest i such that $f_i \neq 0$.

If $f, g \in k[x]$ then $\deg fg = \deg f + \deg g$ and $\deg(f \pm g) \leq \max\{\deg f, \deg g\}$.

A.17   Monic polynomials

An element $f = (f_0, f_1, \dots) \in k[x]$ is called **monic** if $f \neq 0$ and $f_{\deg f} = 1$; i.e., $f \neq 0$ and the coefficient of $x^{\deg f}$ in f is 1.

A.18   Evaluation

If $f = (f_0, f_1, \dots) \in k[x]$ and $\alpha \in k$ then the **value of f at α**, denoted $f(\alpha)$, is $\sum_{i \geq 0} f_i \alpha^i$, i.e., $\sum_{i : f_i \neq 0} f_i \alpha^i$. This is an element of k. Beware the ambiguity of concatenation being used to express both multiplication and evaluation: $(\alpha + \beta)f$, $(\alpha + \beta) \cdot f$, and $f \cdot (\alpha + \beta)$ refer to products in $k[x]$, while $f(\alpha + \beta)$ refers to a value in k.

For each $\alpha \in k$, the map $f \mapsto f(\alpha)$ from $k[x]$ to k is a ring morphism. In other words, for $f, g \in k[x]$ one has $f(\alpha) = 0$ if $f = 0$, $f(\alpha) = 1$ if $f = 1$, $(-f)(\alpha) = -f(\alpha)$, $(f + g)(\alpha) = f(\alpha) + g(\alpha)$, and $(f \cdot g)(\alpha) = f(\alpha) \cdot g(\alpha)$.

A.19   Roots

For $f \in k[x]$ and $\alpha \in k$, saying that α is a **root of f** means that $f(\alpha) = 0$. This is equivalent to $f = (x - \alpha)q$ for some $q \in k[x]$, i.e., $f \in (x - \alpha)k[x]$.

A.20   Vandermonde invertibility

If $f \neq 0$ then f has at most $\deg f$ roots. Equivalently: if $\alpha_1, \alpha_2, \dots, \alpha_n \in k$ are distinct, and $f_0, f_1, \dots, f_{n-1} \in k$ satisfy $\sum_i f_i \alpha_j^i = 0$ for all $j \in \{1, 2, \dots, n\}$, then $(f_0, f_1, \dots, f_{n-1}) = (0, 0, \dots, 0)$.

A.21   Derivatives

If $f = (f_0, f_1, f_2, f_3, \dots) \in k[x]$ then the **derivative of f** is $(f_1, 2f_2, 3f_3, \dots)$. In other words, the derivative of $\sum_i f_i x^i$ is $\sum_{i \geq 1} i f_i x^{i-1}$.

If $f, g \in k[x]$ then $(fg)' = fg' + f'g$, where $f', g', (fg)'$ are the derivatives of f, g, fg respectively; this is the **product rule**.

One consequence of the product rule is **Bernoulli's rule** that if $\alpha \in k$ and $f(\alpha) = 0$ then $(fg)'(\alpha) = f'(\alpha) \cdot g(\alpha)$. Bernoulli's rule is typically described as a rule for evaluating some "0/0" expressions: if $f(\alpha) = 0$ and $f'(\alpha) \neq 0$ then the ratio $(fg/f)(\alpha)$ is $(fg)'(\alpha)/f'(\alpha)$. Bernoulli's rule is often called L'Hôpital's rule, for reasons explained in [Stru63].

As an example of Bernoulli's rule, if $\alpha_1, \dots, \alpha_n \in k$ and $A = \prod_{1 \leq j \leq n}(x - \alpha_j)$ then $A'(\alpha_h) = \prod_{1 \leq j \leq n, j \neq h}(\alpha_h - \alpha_j)$.

A.22   Shifts

If $f = (f_0, f_1, \dots) \in k[x]$ and $\alpha \in k$ then the **shift of f by α**, denoted $f(x + \alpha)$, means the element $\sum_{i \geq 0} f_i(x + \alpha)^i$ of $k[x]$.

The map $f \mapsto f(x + \alpha)$ from $k[x]$ to $k[x]$ is a ring morphism. This map preserves degrees: $\deg f(x + \alpha) = \deg f$. This map also preserves derivatives: the derivative of $f(x + \alpha)$ is $f'(x + \alpha)$, where f' is the derivative of f.

One can unify evaluation and shifts into a more general evaluation operation. This generality is not necessary for the main body of this paper, but is used in Appendix C.

A.23   Quotients and remainders

If $f, g \in k[x]$ and $g \neq 0$ then there are unique $q, r \in k[x]$ such that $f = gq + r$ and $\deg r < \deg g$. If $r = 0$ then the notation f/g means q.

A.24   Unique factorization

The ring $k[x]$ is a unique-factorization domain. In particular, if $f \in k[x]$ has roots $\alpha_1, \dots, \alpha_n \in k$, and $\alpha_1, \dots, \alpha_n$ are distinct, then $f \in (x - \alpha_1) \cdots (x - \alpha_n)k[x]$.

A.25   Greatest common divisors

If $f, g \in k[x]$ are not both 0 then there is a unique monic $d \in k[x]$ such that $dk[x] = fk[x] + gk[x]$. This is called the **greatest common divisor of f and g**, written $\gcd\{f, g\}$. One has $f, g \in dk[x]$ and $k[x] = (f/d)k[x] + (g/d)k[x]$, so $\gcd\{f/d, g/d\} = 1$.

A.26   Squarefreeness

A nonzero element $f \in k[x]$ is called **squarefree** if it has the following property: $g^2 \in fk[x]$ implies $g \in fk[x]$. Equivalently, f is not divisible by the square of any irreducible element of $k[x]$. Equivalently, $\gcd\{f, f'\} = 1$ where f' is the derivative of f.

B   Random tests

B.1   Test scripts

> *Beware of bugs in the above code; I have only proved it correct, not tried it.*
> —Knuth [Emd13, page 11 in cited PDF]

Figures B.1.1, B.1.2, B.1.3, and B.1.4 are Sage scripts to test Algorithms 2.1.1, 3.1.1, 4.1.1, and 5.1.1 respectively on random inputs.

B.2   Test-development principles

The primary design objective of random tests is, for any given amount of CPU time spent on testing, to minimize the chance that bugs will avoid the tests. The obvious baseline is to ensure that tests catch every *known* bug in the subroutine being tested. Beyond this, one can try to proactively catch further bugs, extrapolating from what is known about the processes by which people make mistakes.

Bug patterns are a central topic in the literature on software engineering. There is far less attention to bugs in the literature on algorithms. If one is trying to test, for example, an extended-gcd algorithm, then how does one evaluate whether tests reach the baseline of catching every known extended-gcd bug, never mind proactively catching further bugs?

```
from interpolator import interpolator

for q in range(100):
  q = ZZ(q)
  if not q.is_prime_power(): continue
  print('interp %d' % q)
  sys.stdout.flush()
  k = GF(q)
  for loop in range(100):
    n = randrange(q+1)
    a = list(k)
    shuffle(a)
    a = a[:n]
    r = [k.random_element() for j in range(n)]
    phi = interpolator(n,k,a,r)
    assert phi.degree() < n
    assert all(phi(aj) == rj for aj,rj in zip(a,r))
    kpoly = phi.parent()
    assert phi == kpoly.lagrange_polynomial(zip(a,r))
```

Figure B.1.1: Random tests for Algorithm 2.1.1.

Occasionally a bug will be highlighted because it has been shown to have security consequences. For example, Section 8.4 described an exploitable bug pointed out by Chou [Cho20] in the Goppa decoder from [AlbCPTT17] and [AlbCPTT19]. As another example, Ormandy [Orm19] discovered that some inputs would cause an extended-gcd algorithm in a Microsoft cryptography library to enter an infinite loop; this meant that an attacker could trivially cause a server to stop responding, something that [Win19] called a "Windows 10 zero-day security bomb".

However, this information is generally not indexed by algorithm. Furthermore, the baseline goal is to catch every known bug—not merely the bugs already shown to have security consequences. From an engineering perspective, one would expect much more serious efforts to track what has previously gone wrong.

Comer's introduction [Com01] to the differences between two computer-science cultures, namely the mathematical culture and the engineering culture, lists algorithms solely within the mathematical culture. Certainly most algorithm papers are like most mathematics papers in viewing proofs as the primary goal. A typical algorithm paper includes a proof that an algorithm works; the paper is expected to avoid reminding readers that proofs are often wrong, and, in particular, is expected to avoid taking any steps other than a proof to address the risk that the algorithm is wrong. This position is defensible for the occasional computer-verified proofs, but most proofs in the literature are not computer-verified, and the systematic lack of attention to bugs makes test development unnecessarily difficult.

This paper's computer-verified proofs (see Appendix C) reduce, but do not eliminate, the risk of bugs in this paper's algorithms. The theorems cover the main mathematical content of the algorithms, but they do not directly state that the algorithms compute the specified functions. Such a statement would require the proof system to have a definition of algorithms, and a definition of the Sage instructions used in this paper's algorithms; formalizing such definitions is beyond the scope of this paper.

```
from approximant import approximant

for q in range(100):
  q = ZZ(q)
  if not q.is_prime_power(): continue
  print('approximant %d' % q)
  sys.stdout.flush()
  k = GF(q)
  kpoly.<x> = k[]
  for loop in range(100):
    Adeg = randrange(100)
    A = kpoly([k.random_element() for j in range(Adeg)]+[1])
    if Adeg == 0:
      B = kpoly(0)
    else:
      Bdeg = randrange(Adeg)
      B = kpoly([k.random_element() for j in range(Bdeg+1)])
      # note that B could actually have lower degree
    t = randrange(Adeg+3)
    a,b = approximant(t,k,A,B)
    assert gcd(a,b) == 1
    assert a.degree() <= t
    assert b.degree() < t
    assert a != 0
    assert a*B-b*A == 0 or (a*B-b*A).degree() < A.degree()-t
```

Figure B.1.2: Random tests for Algorithm 3.1.1.

B.3   General shape of these tests

The element $0 \in k$ plays a special role in linear algebra, the definition of polynomials, etc. The tests here try small fields k so that 0 will often appear at various positions in the computation. Hopefully this means that any mishandling of 0 will be triggered by the tests.

Half of the tests of Reed–Solomon decoding in Figure B.1.3 are tests aimed at checking correct behavior on decodable inputs. These tests use input vectors r generated as $e + c$ where $c = (f(\alpha_1), \ldots, f(\alpha_n))$ and e has weight at most t (often chosen to be below t). These tests check whether the decoder finds f.

The other half of the decoding tests are aimed at checking correct behavior on non-decodable inputs. These tests use uniform random input vectors r. If the decoder finds some f then the tests check that $\text{wt}(r-c) \leq t$ where $c = (f(\alpha_1), \ldots, f(\alpha_n))$. If the decoder returns `None`, there is no check whether the decoder should actually have found some f; a bug here should be caught more efficiently by the first type of test.

Figure B.1.4 has an analogous split between testing decodable inputs and testing non-decodable inputs for Goppa decoding. There is no similar split in Figures B.1.1 and B.1.2, since those algorithms handle all inputs successfully.

For Figures B.1.1, B.1.3, and B.1.4, n is chosen randomly between 0 and q; for Figure B.1.2, $\deg A$ is chosen randomly between 0 and 99. Similarly, t is chosen randomly in Figures B.1.2, B.1.3, and B.1.4; in each case, the range of t covered by the tests is slightly beyond the range of t useful for applications.

```
from rs import interpolator_with_errors

for q in range(100):
  q = ZZ(q)
  if not q.is_prime_power(): continue
  print('interpolator_with_errors %d' % q)
  sys.stdout.flush()
  k = GF(q)
  kpoly.<x> = k[]
  for loop in range(100):
    n = randrange(q+1)
    t = randrange(3+n//2)
    a = list(k)
    shuffle(a)
    a = a[:n]
    for known in True,False:
      if known:
        f = kpoly([k.random_element() for j in range(n-2*t)])
        r = list(map(f,a))
        e = [k.random_element() for j in range(t)]+[0]*(n-t)
        shuffle(e)
        assert len([ej for ej in e if ej != 0]) <= t
        for j in range(n): r[j] += e[j]
      else:
        f = 'unknown' # cut off data flow from previous iteration
        r = [k.random_element() for j in range(n)]
      f2 = interpolator_with_errors(n,t,k,a,r)
      if f2 == None:
        assert not known
      else:
        assert f2 == 0 or f2.degree() < n-2*t
        if known: assert f2 == f
        assert len([j for j in range(n) if f2(a[j]) != r[j]]) <= t
```

Figure B.1.3: Random tests for Algorithm 4.1.1.

B.4   How the tests catch various bugs

The bug in the Goppa decoder from [AlbCPTT17] and [AlbCPTT19] is triggered when the correct error vector e has weight $t - 1$ and has $e_z = 0$ where $\alpha_z = 0$. Figure B.1.4 is intended to catch this: the tests generate uniform random sequences $(\alpha_1, \ldots, \alpha_n)$ of distinct field elements, and often use weight $t - 1$ for the error vector e; often α_z will be 0 for some z, and often e_z will also be 0.

In an experiment that modified Algorithm 5.1.1 to imitate what [Cho20] described, Figure B.1.4 immediately caught the bug. One could directly test the algorithms from [AlbCPTT17] and [AlbCPTT19] by translating the algorithms from pseudocode to real code. One could directly test the software accompanying [AlbCPTT17] and [AlbCPTT19] by extracting the Goppa-decoding portions of that software and providing a shim layer to support the `goppa_errors` interface.

The extended-gcd bug mentioned above in Microsoft's cryptography library was that a modular-inversion algorithm continued to loop until finding gcd 1—which would always happen for inputs with modular inverses, but the attacker could provide a non-invertible

```
from goppa import goppa_errors

for m in range(1,10):
  q = 2^m
  print('goppa_errors %d' % q)
  sys.stdout.flush()
  k = GF(q)
  kpoly.<x> = k[]
  for loop in range(100):
    while True:
      n = randrange(q+1)
      t = randrange(3+n//m)
      if t >= n: t = n
      a = list(k)
      shuffle(a)
      a = a[:n]
      G = kpoly([k.random_element() for j in range(2*t)]+[1])
      if all(G(aj) != 0 for aj in a):
        break

    assert G.degree() == 2*t
    A = kpoly(prod(x-aj for aj in a))
    Aprime = A.derivative()
    for aj in a: assert Aprime(aj) != 0

    for known in True,False:
      if known:
        f = kpoly([k.random_element() for j in range(n-2*t)])
        r = [(f*G)(aj)/Aprime(aj) for aj in a]
        if randrange(2):
          e = [1]*t+[0]*(n-t)
        else:
          actualweight = randrange(t+1)
          e = [1]*actualweight+[0]*(n-actualweight)
        shuffle(e)
        assert len([ej for ej in e if ej != 0]) <= t
        for j in range(n): r[j] += e[j]
      else:
        e = 'unknown' # cut off data flow from previous iteration
        r = [k.random_element() for j in range(n)]
      e2 = goppa_errors(n,t,k,a,G,r)
      if e2 == None:
        assert not known
      else:
        assert len(e2) == n
        if known: assert e2 == e
        assert len([ej for ej in e2 if ej != 0]) <= t
        assert G.divides(sum((r[i]-e2[i])*A//(x-a[i]) for i in range(n)))
```

Figure B.1.4: Random tests for Algorithm 5.1.1.

input, triggering an infinite loop. In the decoding context, an extended-gcd computation is the normal way to compute approximants, and one can imagine someone

- starting with an extended-gcd algorithm that computes all remainders,

- augmenting the algorithm to record (a, b) for the first remainder $aB - bA$ of degree below $\deg A - t$, and

- not optimizing away the pointless computation of subsequent remainders,

so there could still be an infinite-loop bug. In these tests, because k is small, some input positions will often be 0, forcing $\gcd\{A, B\} \neq 1$, so if there is an infinite loop for that case then the tests will trigger it.

Another easy bug to imagine in Reed–Solomon decoders and Goppa decoders is testing $\deg(aB - bA)$ against $n - t$ rather than $n - 2t + \deg a$, although this does not matter in an application that requires $\deg a = t$. An experiment with eight runs of Figure B.1.3 consistently caught this bug; each run already caught the bug with $\#k = 2$. Another experiment with eight runs of Figure B.1.4 also consistently caught this bug; here the eight runs caught the bug with $\#k = 8$, $\#k = 16$, $\#k = 4$, $\#k = 8$, $\#k = 8$, $\#k = 4$, $\#k = 32$, $\#k = 4$ respectively. The variation in $\#k$ here suggests running more repetitions of the tests for reliability, or adding tests specifically for this case.

C   Computer-verified formalizations of the theorems

This appendix presents formalizations of this paper's theorems using two proof assistants, namely HOL Light and Lean 4; compares the formalized statements to the theorem statements earlier in this paper; and explains how to verify proofs of the formalized theorems. The proofs, `lightgoppa-20230818.ml` and `leangoppa-20230818.tar.gz`, are supplements available at the URLs shown below.

See [Har96] for an introduction to HOL Light, [MouU21] for an introduction to Lean 4, and [Mat20] for an introduction to the Lean math library. This appendix says "HOL Light" and "Lean" to refer to the full proof assistants available when these formalizations began, including the math libraries.

See [BraCPR22, Section 4] for a report of previous progress towards formalizing this paper's theorems in Lean. Theorems on direct interpolation were already in Lean.

C.1   Risks

The claim that a computer has verified a proof does not mean that the user is safe. On the contrary, various risks should be kept in mind.

First, as noted above, this paper's theorem statements are not stating that any particular software works correctly. The theorem statements capture the mathematical content of what some decoding software is *intended* to compute, but perhaps the software actually computes something else. This appendix does not directly address this risk. Appendix B directly addresses this risk, but only for a limited set of inputs.

Second, there can be mismatches between the formalized theorems and the theorems claimed earlier in this paper. Such mismatches could allow a claimed theorem to be false despite a computer-verified proof of the formalized theorem. Appendix C.6 addresses this risk by comparing theorem statements.

Third, beyond the risks of mismatches in the theorem statements, there are risks of mismatches in the underlying definitions. This paper does not review the complete chain of definitions, but Appendix C.5 reviews the definitions that seem most likely to cause problems.

Fourth, bugs are sometimes discovered in proof-verification software. An erroneous proof could slip past verifier bugs, even in the absence of malice. Proof software typically addresses this by delegating verification to a "relatively small trusted kernel", in the words of [MouU21], so the problem is then simply to audit that kernel. The HOL Light kernel is particularly small, and [AbrMKS22] includes a proof of correctness of the kernel.

Fifth, HOL Light is written in a general-purpose programming language, namely OCaml; Lean is itself a general-purpose programming language; and, for both HOL Light and Lean, proofs are software written in the same languages. Buggy code or malicious code inside any of this software—not just the kernel responsible for verifying proofs—can spoil verification, destroy files, etc. This appendix recommends running verification inside a virtual machine, but does not otherwise address the risk of malice.

Candle from [AbrMKS22] is a port of HOL Light to the verified CakeML compiler, and is backed by a theorem stating that the resulting machine code releases only correct theorems (where "release" refers to a particular output mechanism; the Candle user can, with some work, disable other output mechanisms). However, Candle's official version does not yet include HOL Light's ring-theory library.

C.2 Verifying the proofs of the HOL Light formalizations

Here is how to run `lightgoppa` in HOL Light. These instructions assume a Debian virtual machine with at least 2GB free disk space.

A few instructions are run as root in the virtual machine:

```
apt install git opam wget -y
adduser --disabled-password --gecos lightgoppa lightgoppa
su - lightgoppa
```

All remaining steps are within the `lightgoppa` account:

```
time opam init -a
time opam switch create 4.05.0
eval `opam env`
time opam pin add camlp5 7.10 -y
time opam install num camlp-streams ocamlfind -y

git clone https://github.com/jrh13/hol-light
cd hol-light
git checkout 29b3e114f5c166584f4fbcfd1e1f9b13a25b7349
make

wget https://cr.yp.to/2023/lightgoppa-20230818.ml
time ocaml -I `camlp5 -where` camlp5o.cma -init hol.ml \
< lightgoppa-20230818.ml > lightgoppa-20230818.out
grep Error lightgoppa-20230818.out
```

The timed commands were observed to take 16, 273, 59, 114, and 1168 seconds respectively on a dual AMD EPYC 7742 server running at 2.245 GHz, mostly using just 1 of the server's 128 cores. The resulting `lightgoppa-20230818.out` has 40792 lines, including copies of all definitions and proven theorems. The first 80% of the lines are from various HOL Light libraries. The main `lightgoppa` theorem statements are collected at the end. Proof errors would instead produce `Error` lines.

C.3 Verifying the proofs of the Lean formalizations

Here is how to run `leangoppa` in Lean. These instructions assume a Debian virtual machine with at least 8GB free disk space.

A few instructions are run as root in the virtual machine:

```
apt install git curl -y
adduser --disabled-password --gecos leangoppa leangoppa
su - leangoppa
```

All remaining steps are within the `leangoppa` account:

```
GH=raw.githubusercontent.com
curl -sS -o elan-init.sh \
  https://$GH/leanprover/elan/master/elan-init.sh
sh elan-init.sh -y
source $HOME/.elan/env
wget https://cr.yp.to/2023/leangoppa-20230818.tar.gz
tar -xf leangoppa-20230818.tar.gz
cd leangoppa-20230818
time lake exe cache get
time lake build
```

The `lake exe cache get` was observed to take 42 seconds on the machine mentioned above. The `lake build` was observed to take 162 seconds, producing some `Building` lines (and timing information from `time` at the end). Proof errors would instead produce `error` lines.

C.4   Examples of the underlying definitions

Typing "`field;;`" in a HOL Light session prints out

```
val it : thm =
  |- !r. field r <=>
         ~(ring_1 r = ring_0 r) /\
         (!x. x IN ring_carrier r /\ ~(x = ring_0 r)
              ==> (?y. y IN ring_carrier r /\ ring_mul r x y = ring_1 r))
```

which is a translation of the following statement into the HOL Light language: a commutative ring r is a field if and only if

- $1 \neq 0$ in r and

- for each $x \in r$ such that $x \neq 0$ in r: there exists $y \in r$ such that $xy = 1$ in r.

HOL Light reserves the word "ring" for commutative rings; this matches common terminology in commutative algebra, although it does not match common terminology in non-commutative algebra.

As this translation illustrates, HOL Light uses ASCII versions of various logic symbols: `|-` for $\vdash$ (proves), `<=>` for $\Leftrightarrow$ (if and only if), `==>` for $\Rightarrow$ (implies), `/\` for $\wedge$ (and), `!` for $\forall$ (for all), `?` for $\exists$ (there exists), `~` for $\neg$ (not). Also, if a set S has elements 0 and 1 and operations $-, +, \cdot$ satisfying the usual identities, and if `r` is defined as the ring $(S, 0, 1, -, +, \cdot)$, then `ring_carrier r` means S and `ring_0 r` means 0 and so on.

More subtly, using the syntax `ring_1 r`, `ring_0 r`, etc. adds an implicit hypothesis that r is a `ring`. One can make this more explicit in theorem statements by replacing `!r` with `!r:A ring`, where S is a subset of `A`.

HOL Light's definition of `ring` takes longer to read:

```
let ring_tybij =
  let eth = prove
   (`?s (z:A) w n a m.
           z IN s /\
           w IN s /\
           (!x. x IN s ==> n x IN s) /\
           (!x y. x IN s /\ y IN s ==> a x y IN s) /\
           (!x y. x IN s /\ y IN s ==> m x y IN s) /\
           (!x y. x IN s /\ y IN s ==> a x y = a y x) /\
           (!x y z. x IN s /\ y IN s /\ z IN s ==> a x (a y z) = a (a x y) z) /\
           (!x. x IN s ==> a z x = x) /\
           (!x. x IN s ==> a (n x) x = z) /\
           (!x y. x IN s /\ y IN s ==> m x y = m y x) /\
           (!x y z. x IN s /\ y IN s /\ z IN s ==> m x (m y z) = m (m x y) z) /\
           (!x. x IN s ==> m w x = x) /\
           (!x y z. x IN s /\ y IN s /\ z IN s
                    ==>  m x (a y z) = a (m x y) (m x z))`,
     MAP_EVERY EXISTS_TAC
      [`{ARB:A}`; `ARB:A`; `ARB:A`; `(\x. ARB):A->A`;
       `(\x y. ARB):A->A->A`; `(\x y. ARB):A->A->A`] THEN
     REWRITE_TAC[IN_SING] THEN MESON_TAC[]) in
  new_type_definition "ring" ("ring","ring_operations")
   (GEN_REWRITE_RULE DEPTH_CONV [EXISTS_UNPAIR_THM] eth);;
```

Part of the length comes directly from the number of rules in the usual mathematical definition of $(S, 0, 1, -, +, \cdot)$ being a commutative ring. For example, the line "`!x y. x IN s /\ y IN s ==> m x y = m y x`" states that each x, y with $x \in S$ and $y \in S$ has $\cdot(x, y) = \cdot(y, x)$, i.e., that multiplication is commutative; this is one of eight identities in the definition, after five lines saying that S is closed under $0, 1, -, +, \cdot$.

The surrounding lines are showing that an object satisfying these rules exists (namely, a ring of one element), and are then saying that any tuple $(S, 0, 1, -, +, \cdot)$ satisfying the rules, or more precisely $(S, (0, (1, (-, (+, \cdot)))))$, is referred to as the `ring_operations` of a `ring`.

The field definition quoted above relies on further definitions that extract S etc. from such a tuple. For example, the following definition says that if `r` is a `ring` then `ring_carrier r` is defined as the first component of the `ring_operations` tuple, i.e., that the carrier of the ring $(S, 0, 1, -, +, \cdot)$ is S:

```
let ring_carrier = new_definition
 `(ring_carrier:(A)ring->A->bool) =
    \r. FST(ring_operations r)`;;
```

Inside this definition, `X->Y` means the set of functions from `X` to `Y`; `A->bool` in particular means the set of subsets of `A`, modeled as the set of functions from `A` to $\{\text{True}, \text{False}\}$; `(A)ring` means the set of rings $(S, \ldots)$ whose first components S are subsets of `A`; and `(A)ring->A->bool` means the set of functions mapping such rings to subsets of `A`. This definition specifies `ring_carrier` as one such function, namely the function that maps $(S, \ldots)$ to S: the tuple $(S, 0, 1, -, +, \cdot)$ is modeled via pairs as $(S, (0, (1, (-, (+, \cdot)))))$, and `FST` means the first component of a pair.

Reading the complete definition of a field in Lean takes much longer, and only fragments of the definition are displayed below. At the top there is

```
class Field (K : Type u) extends CommRing K, DivisionRing K
```

which sounds easy enough—a field is a commutative division ring. The complications begin at the next layer, the definition of a division ring:

```
class DivisionRing (K : Type u) extends Ring K, DivInvMonoid K, Nontrivial K, RatCast K where
  /-- For a nonzero `a`, `a⁻¹` is a right multiplicative inverse. -/
  protected mul_inv_cancel : ∀ (a : K), a ≠ 0 → a * a⁻¹ = 1
  /-- We define the inverse of `0` to be `0`. -/
  protected inv_zero : (0 : K)⁻¹ = 0
  protected ratCast := Rat.castRec
  /-- However `ratCast` is defined, propositionally it must be equal to `a * b⁻¹`. -/
  protected ratCast_mk : ∀ (a : ℤ) (b : ℕ) (h1 h2), Rat.cast (a, b, h1, h2) = a * (b : K)⁻¹ := by
    intros
    rfl
  /-- Multiplication by a rational number. -/
  protected qsmul : ℚ → K → K := qsmulRec Rat.cast
  /-- However `qsmul` is defined,
  propositionally it must be equal to multiplication by `ratCast`. -/
  protected qsmul_eq_mul' : ∀ (a : ℚ) (x : K), qsmul a x = Rat.cast a * x := by
    intros
    rfl
```

This defines a division ring as a tuple $(K, Q, X, Z, Q', Q'', Q''')$, where

- K is a nontrivial ring;

- K is also a "division-inversion monoid";

- Q is a function from $\mathbb{Q}$ to K;

- X is the fact that each nonzero $a \in K$ has $aa^{-1} = 1$ in K;

- Z is the fact that $0^{-1} = 0$ in K;

- Q' is the fact that each $a \in \mathbb{Z}$ and $b \in \mathbb{N} = \{0, 1, \ldots\}$ has $Q(a/b) = ab^{-1}$ in K;

- Q'' is a function from $\mathbb{Q} \times K$ to K; and

- Q''' is the fact that each $a \in \mathbb{Q}$ and $x \in K$ has $Q''(a, x) = Q(a)x$ in K.

These definitions rely on the usual map from $\mathbb{Z}$ to K having already been specified, on 0^{-1} being defined as 0, and on the underlying `DivInvMonoid` definition—which has its own complications and is not displayed here—allowing division by 0. HOL Light also defines $0^{-1} = 0$.

One can see from the above definition that Lean uses the syntax $0 : K$ to refer to 0 in K, and the syntax `x*y` for the product of elements x and y of K. The conciseness of this notation improves legibility compared to HOL Light's notation `ring_0 K` and `ring_mul r x y`, especially in long formulas. Lean also automatically deduces that the second 0 in the equation $(0 : K)^{-1} = 0$ is intended as $0 : K$.

Similarly, Lean uses the syntax $q : K$ to refer to $Q(q)$, where q is a rational number and Q is the function provided as part of defining the division ring K. Note that a reader checking Lean's definitions for an application using fields is confronted with the Q, Q', Q'', Q''' complications whether or not the application uses this syntax, and has to figure out whether these complications restrict the mathematical concept of a field.

Both HOL Light and Lean include various theorems about fields, such as HOL Light's theorem `FIELD_INTEGER_MOD_RING` saying

```
!n. field (integer_mod_ring n) <=> prime n
```

(for each n, the ring $\mathbb{Z}/n$ is a field if and only if n is prime) and Lean's unnamed construction

```
variable (p : ℕ) [Fact p.Prime]
...
/-- Field structure on `ZMod p` if `p` is prime. -/
instance : Field (ZMod p) :=
...
```

(the weaker statement that, for each prime p, the ring $\mathbb{Z}/p$ is a field). There are further theorems saying, e.g., that 2 is a prime, so fields exist with the definitions of both systems; the definitions are not accidentally vacuous. Also, the theorem

```
variable (p : ℕ) [h_prime : Fact p.Prime] (n : ℕ)
...
theorem card (h : n ≠ 0) : Fintype.card (GaloisField p n) = p ^ n := by
...
```

shows (in conjunction with the fact that any `GaloisField` is a `Field`) that Lean's definition of fields allows fields of any prime-power cardinality. Such theorems reduce the risk of improper definitions slipping through, although a reviewer is still happier when the definitions per se are clean and easy to review.

C.5   High-risk definitions

A complete review would check many more definitions and theorems in HOL Light and Lean, building confidence that both systems are defining concepts equivalent to various standard mathematical concepts. This appendix does not include a complete review. There are, however, two reasons to study some definitions more closely.

First, both systems make some dangerous choices of notation. For example, the function $m, n \mapsto \max\{0, m - n\}$ from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}$, where $\mathbb{N} = \{0, 1, 2, \ldots\}$, is given the surprisingly short name "$-$" in both HOL Light and Lean. This creates an obvious risk of confusion with the mathematician's much more important subtraction function from $\mathbb{Z} \times \mathbb{Z}$ to $\mathbb{Z}$, which is also denoted "$-$" in these systems and in the broader literature. The two functions have incompatible semantics in the sense that applying the usual map from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{Z} \times \mathbb{Z}$, and then applying "$-$", does not always produce the same results as applying "$-$" and then the usual map from $\mathbb{N}$ to $\mathbb{Z}$. The reader expects $2 - 3$ to be -1, not 0. As another example, Lean defines `a/b` to mean what mathematicians would write as $\lfloor a/b \rfloor$; this is usually not the same as a/b.

Second, for a paper formalizing additional definitions, there is a clear need to carefully check those definitions. The formalized theorem statements below rely on the following 8 definitions in `leangoppa` and the following 21 definitions in `lightgoppa`. The reason there are more definitions in `lightgoppa` is that, for some of the following concepts, `lightgoppa` develops the concepts from scratch while `leangoppa` uses concepts already built into Lean.

C.5.1   Squarefree elements of rings

In `lightgoppa`, `ring_squarefree` is defined as follows:

```
let ring_squarefree = new_definition `
  ring_squarefree(r:R ring) a
  <=>
  (!b. b IN ring_carrier r ==>
   ring_divides r a (ring_mul r b b) ==>
   ring_divides r a b
  )
`;;
```

This says that an element a of a (commutative) ring r is squarefree if and only if each b in r with a dividing b^2 also has a dividing b. The `ring_divides` notion is already provided by HOL Light; the definition is not repeated here.

Note that $X \Rightarrow (Y \Rightarrow Z)$ is logically equivalent to $(X \wedge Y) \Rightarrow Z$. Conventionally, and in HOL Light, syntax is defined so that $X \Rightarrow Y \Rightarrow Z$ means $X \Rightarrow (Y \Rightarrow Z)$. But one can instead write $X \wedge Y \Rightarrow Z$ (i.e., `X /\ Y ==> Z`) if desired, meaning $(X \wedge Y) \Rightarrow Z$, as illustrated by the field definition quoted above.

As an example of the importance of checking definitions, consider the fact that a sufficiently severe misdefinition of squarefreeness could make a formalization of Theorem 6.1.1 content-free. As extra evidence that the above definition of `squarefree` matches the expected notion, `lightgoppa` includes a proof that any prime element of r is squarefree—

```
let ring_squarefree_if_prime = prove(`
  !(r:R ring) p.
  ring_prime r p ==> ring_squarefree r p
`,
...
```

—and a proof that, more generally, a product of prime elements is squarefree if the prime elements do not divide each other.

Lean already provides a `Squarefree` definition, so `leangoppa` simply uses that. Lean's `Squarefree` syntax is more concise than the `ring_squarefree` syntax: it does not mention the ring in its name or as an argument.

C.5.2 Univariate polynomials

HOL Light has some general development of multivariate polynomials over a ring. The following definition in `lightgoppa` extracts the special case of univariate polynomials:

```
let x_ring = new_definition `
  x_ring (r:R ring) = poly_ring r {0}
`;;
```

The notation here is deceptively concise, as the following paragraphs explain.

In both HOL Light and Lean, the universe is partitioned into disjoint "types". There are various ways to construct types, such as the type $\mathbb{N} = \{0, 1, 2, \ldots\}$ (denoted `num` in HOL Light) and the type $X \rightarrow Y$ of functions from type X to type Y. Subsets of X are formalized as functions from X to $\{\text{True}, \text{False}\}$, as noted above.

HOL Light's $\mathbb{Z}$ type is not a superset of $\mathbb{N}$—this would violate the disjointness rule. Instead of having elements `0` and `1` and so on, HOL Light's $\mathbb{Z}$ (denoted `int`) has elements `&0` and `&1` and so on, along with negative integers. (Similarly, Lean's 0 in $\mathbb{N}$ is a different object from Lean's 0 in $\mathbb{Z}$, even though the Lean syntax often allows the map from $\mathbb{N}$ to $\mathbb{Z}$ to be left implicit,) What matters here is that there is only one type containing 0, namely $\mathbb{N}$.

Mathematically, multivariate polynomials over a ring r are functions from exponent vectors to r satisfying certain finiteness conditions. For example, the polynomial $2x^3y^4 + 5x^6y^7$ is the function that maps $(3, 4)$ to 2, maps $(6, 7)$ to 5, and maps everything else to 0. The exponent vector $(3, 4)$, in turn, can be viewed as a function producing 3 on one input and 4 on another input. HOL Light formalizes exponent vectors as functions from any *type* V to $\mathbb{N}$; HOL Light then defines `poly_ring r S`, where S is any *subset* of V, as the polynomials with exponent vectors supported on S.

Defining `x_ring r` as `poly_ring r {0}` is thus extracting the polynomials with exponent vectors supported on $\{0\}$ out of the ring of polynomials with exponent vectors supported on $\mathbb{N}$, since $\mathbb{N}$ is the unique type containing 0: in other words, extracting $r[x_0]$

out of $r[x_0, x_1, x_2, \ldots]$. The exponent vectors here are defined as functions from $\mathbb{N}$ to $\mathbb{N}$, and polynomials are defined as functions mapping exponent vectors to r. In HOL Light syntax, the type of an exponent vector is `num->num` in this case, and the type of a polynomial is `(num->num)->R`, when the carrier of r is a subset of R.

Given this setup, the definition of coefficient extraction from a univariate polynomial is factored into two definitions in `lightgoppa`:

```
let map0to = new_definition `
  map0to (d:num) = \v. if v = 0 then d else 0
`;;


let coeff = new_definition `
  coeff (d:num) (p:(num->num)->R)
  = p(map0to d)
`;;
```

The backslash is HOL Light's ASCII version of the λ notation for defining functions: `map0to d` is defined as the function mapping v to d if $v = 0$ and to 0 otherwise. HOL Light automatically figures out that this function has type `num->num`, since 0 is used as an input of the function. It would figure this out even if `(d:num)` were abbreviated as `d`, since 0 is also used as an output of the function.

The point of `map0to d` is that it is the exponent vector of the monomial $x_0^d x_1^0 x_2^0 \ldots$ in the polynomial ring $r[x_0, x_1, x_2, \ldots]$. Viewing a polynomial as a function from exponent vectors, and evaluating this polynomial at $(d, 0, 0, \ldots)$, extracts the coefficient of x_0^d from the polynomial. This is what `coeff d p` does.

(Appendix A.15 defined coefficients more generally for $d \in \mathbb{Z}$. Both `lightgoppa` and `leangoppa` include a corresponding `zcoeff` definition. This is a convenient tool inside the *proof* of Theorem 3.1.2, but someone reviewing the *theorem* statements displayed below can skip it. Similarly, polynomial shifts as in Appendix A.22 appear in the proof of Theorem 7.1 but not in the theorem statement.)

Note that someone using univariate polynomial rings generally does not want to know implementation details such as the variable being labeled 0 and the ring being carved out of $r[x_0, x_1, \ldots]$. Most of the definitions and theorem statements below are expressed in terms of `coeff` etc., and it would be useful to add syntax to abstract away the occasional `(num->num)->R`.

Even with a suitable abstraction layer, obtaining univariate polynomial rings as a specialization of multivariate polynomial rings means that someone reviewing the full definition is faced with the complications of multivariate polynomial rings even for applications not using those complications. On the other hand, having `lightgoppa` define its own univariate polynomial rings would mean more high-risk definitions to review. Presumably what will be best in the long run is a formalized version of what appears in the mathematical literature: a simple definition of univariate polynomial rings, a more complicated definition of multivariate polynomial rings, and theorems regarding the relationship between the definitions.

For Lean, `leangoppa` reuses Lean's existing definition of `k[X]` as the ring of univariate polynomials over k, and Lean's definition of `p.coeff d` as the coefficient of X^d in p. The formal definitions of these objects in Lean are not reviewed here.

C.5.3   Special polynomials

Several important polynomials are given names in `lightgoppa`:

```
let x_pow = new_definition `
  x_pow (r:R ring) (d:num) =
  \m. if m = map0to d then ring_1 r else ring_0 r
`;;
```

This defines `x_pow r d` as the polynomial x^d in $r[x]$, i.e., the polynomial x_0^d in $r[x_0, x_1, \ldots]$: formally, the function that maps the exponent vector $(d, 0, 0, \ldots)$ to 1 and maps everything else to 0.

```
let poly_x = new_definition `
  poly_x (r:R ring) = x_pow r 1
`;;
```

This defines `poly_x r` as the polynomial x^1 in $r[x]$.

```
let const_x_pow = new_definition `
  const_x_pow (r:R ring) c d =
    ring_mul(x_ring r) (poly_const r c) (x_pow r d)
`;;
```

This defines `const_x_pow r c d` as the polynomial cx^d; `poly_const` is already provided by HOL Light as the usual map from r to $r[x]$. Lean defines an equivalent function under the name `monomial`, but supports the more concise syntax `(C c)*X^d`; `C` is defined as the usual map from r to $r[X]$.

```
let x_minus_const = new_definition `
  x_minus_const (r:R ring) (c:R)
  = ring_sub(x_ring r) (poly_x r) (poly_const r c)
`;;
```

This defines `x_minus_const r c` as the polynomial $x - c$. In Lean, writing `X - C c` is more concise.

C.5.4   Degrees of polynomials

To avoid working with $-\infty$, `lightgoppa` works with the map $f \mapsto 2^{\deg f}$, denoted `twodeg`, from $r[x]$ to $\mathbb{N}$. If r is a field (or, more generally, a domain) then this maps multiplication to multiplication. The definition of `twodeg` is factored into the following three layers.

```
let x_support = new_definition `
  x_support (r:R ring) (p:(num->num)->R)
  = {d | ~(coeff d p = ring_0 r)}
`;;
```

This defines `x_support r p` as the set of natural numbers d such that the coefficient of x^d in p is nonzero. Recall that `~` in HOL Light means "not". HOL Light automatically figures out that d ranges through $\mathbb{N}$, since `coeff` requires its input to be in $\mathbb{N}$.

```
let maximum = new_definition `
  maximum S
  = @m:num. m IN S /\ (!n. m < n ==> ~(n IN S))
`;;
```

This defines notation for maximal elements of sets of natural numbers. (HOL Light already defines notation for minimal elements of sets of natural numbers.) In HOL Light, "`@m. P m`" means a choice of m satisfying $P(m)$, or an arbitrary choice of m (within whichever type) if no choice of m satisfies $P(m)$. This definition says that `maximum S` is some $m \in \mathbb{N}$ such that $m \in S$ and each $n > m$ has $n \notin S$, or, if no such m exists, an arbitrary $m \in \mathbb{N}$.

```
let twodeg = new_definition `
  twodeg (r:R ring) (p:(num->num)->R) =
  if p = ring_0(x_ring r)
  then 0
  else 2 EXP (maximum (x_support r p))
`;;
```

This defines `twodeg r p` as 0 if $p = 0$, or as 2^m (denoted `2 EXP m` in HOL Light) if $p \neq 0$, where m is the maximum element of `x_support r p`. Both cases match the usual concept of $2^{\deg p}$.

In Lean, `p.degree` is the usual mathematical degree of p. This degree is in a "WithBot $\mathbb{N}$" type that can be viewed as $\mathbb{N} \cup \{-\infty\}$, although it actually has a separate copy of $\mathbb{N}$ to follow the disjointness rule mentioned above. The Lean syntax automatically maps $\mathbb{N}$ to WithBot $\mathbb{N}$, so one does not see WithBot named in any of the theorem statements below. The semantics of the theorem statements still depend on WithBot.

Lean also provides `natDegree` producing results in $\mathbb{N}$, in particular mapping the polynomial 0 to 0. This avoids WithBot but would force various theorem statements to treat the polynomial 0 specially.

C.5.5   Polynomial evaluation

The "more general evaluation operation" mentioned in Appendix A.22 is as follows: any ring morphism from r to s can be extended to a ring morphism from $r[x]$ to s; the extension is unique if one specifies the image of x in s.

Even more generally, write $r[V]$ for the multivariate polynomial ring with polynomial variables indexed by V. For any ring morphism $f : r \to s$ and any function $e : V \to s$, there is a unique ring morphism from $r[V]$ to s compatible with f (i.e., f is the composition of the usual map $r \to r[V]$ and this morphism $r[V] \to s$) and compatible with e (i.e., this morphism takes each variable $v \in V$ to $e(v)$).

HOL Light defines `poly_extend r s f e` as this general morphism. The following definition specializes this to the case of univariate polynomial evaluation, or, more precisely, evaluating all variables at the same point:

```
let poly_eval = new_definition `
  poly_eval (r:R ring) (a:R)
  = poly_extend(r,r) I (\v:num. a)
`;;
```

The ring morphism $f : r \to r$ here is `I`, which HOL Light defines as the identity function. The function $e : \mathbb{N} \to r$ maps all inputs to a given element $a \in r$. The resulting `poly_eval r a` is then a function mapping $r[x_0, x_1, \ldots]$ to r, specifically the evaluation map that takes each x_j to a. In `lightgoppa`, this is applied to various elements of $r[x] = r[x_0]$.

Reviewers checking this definition and the underlying `poly_extend` definition, to see that this is the expected polynomial-evaluation function on $r[x]$, are again faced with multivariate complications. However, reviewers can instead inspect the following theorem, which completely characterizes `poly_eval r a p` for $a \in r$ and $p \in r[x]$:

```
let poly_eval_expand = prove(`
  !(r:A ring) a p.
  a IN ring_carrier r ==>
  p IN ring_carrier(x_ring r) ==>
  poly_eval r a p =
  ring_sum r
    (x_support r p)
    (\d. ring_mul r (coeff d p) (ring_pow r a d))
`,
...
```

This still relies on HOL Light's definition of `ring_pow r a d` as a^d in r, and its definition of `ring_sum r S f` as the sum in r of $f(s)$ for all $s \in S$.

C.5.6   Derivatives of polynomials

The definition of derivative in `lightgoppa` is factored into the following two layers:

```
let monomial_shift = new_definition `
  monomial_shift (m:num->num)
  = (\v. if v = 0 then m v + 1 else m v)
`;;
```

This maps exponent vector $(m_0, m_1, m_2, \ldots)$ to $(m_0 + 1, m_1, m_2, \ldots)$.

```
let x_derivative = new_definition `
  x_derivative (r:R ring) (p:(num->num)->R)
  = (\m. ring_mul r
          (ring_of_num r (m 0 + 1))
          (p (monomial_shift m)))
`;;
```

This defines `x_derivative r p` as the function that maps an exponent vector $m = (m_0, m_1, \ldots)$ to $(m_0 + 1)p_{m_0+1,m_1,\ldots}$. HOL Light defines `ring_of_num r` to map $\mathbb{N}$ to r in the usual way.

The case that matters for univariate polynomials is that `x_derivative r p` maps exponent vector $(d, 0, \ldots)$ to $(d+1)p_{d+1,0,\ldots}$. In other words, if p has coefficients $(p_0, p_1, p_2, \ldots)$ then `x_derivative r p` has coefficients $(1p_1, 2p_2, 3p_3, \ldots)$, as in Appendix A.21. Instead of checking these definitions, reviewers can rely on the following characterization:

```
let x_derivative_expand_twodeg = prove(`
  !(r:R ring) p H.
  p IN ring_carrier(x_ring r) ==>
  twodeg r p <= H ==>
  x_derivative r p
  = ring_sum (x_ring r)
      {d | 2 EXP d <= H}
      (\d. const_x_pow r
            (ring_mul r (ring_of_num r d) (coeff d p))
            (d-1))
`,
...
```

This says that if $p \in r[x]$ and $2^{\deg p} \leq H$ then `x_derivative r p` is $\sum_{d:2^d \leq H}(dp_d)x^{d-1}$, where p_d is the coefficient of x^d in p. Beware that, as noted above, HOL Light's `d-1` means 0 if $d = 0$ rather than -1; fortunately, the result for $d = 0$ is multiplied by 0 in r.

In `leangoppa`, there is a `diff` definition providing a thin wrapper around Lean's existing `derivative` definition:

```
variable {R: Type _} [Semiring R]

noncomputable def diff: R[X] → R[X] := derivative.toFun
```

Then `p.diff` is the derivative of `p`. The `toFun` refers to the fact that, in Lean, `derivative` is not a function from $R[X]$ to $R[X]$, but rather a tuple that includes such a function along with extra facts about derivatives, such as the fact that derivatives are compatible with multiplication by scalars in R. (Formally, Lean builds up these facts in multiple layers, and one `toFun` still leaves the additive structure of derivatives, but Lean is able to figure out what the pure `diff` function means.) One can write `p.derivative` or `derivative p` in some—but not all—of the same situations as `p.diff`, with Lean automatically using the desired function.

C.5.7 Interpolator

The following notation is defined in both `lightgoppa` and `leangoppa` to shorten various theorem statements. The definitions are juxtaposed here for comparison.

There is nothing special about the indices $1, \ldots, n$ for a list $\alpha_1, \ldots, \alpha_n$ of distinct elements of k. Lean encourages finite vectors to be indexed by an arbitrary finite set for generality. Accordingly, in `leangoppa`'s formalization, α is an injective function from an arbitrary finite set S to k. Other vectors indexed by $1, \ldots, n$ in this paper's theorems are similarly indexed by S in the formalization.

In `lightgoppa`'s formalization, S is instead specifically a finite subset of k; α is the identity map and is suppressed. Other vectors indexed by $1, \ldots, n$ in this paper's theorems are again indexed by S in the formalization.

```
let monic_vanishing_at = new_definition `
  monic_vanishing_at (r:R ring) (S:R->bool)
  = ring_product(x_ring r)
      S (\s. x_minus_const r s)
`;;
```

This gives a name to $\prod_{s \in S}(x - s)$.

```
variable {k: Type _} [Field k]
variable {U: Type _} [DecidableEq U]

noncomputable def monic_vanishing_at (α: U → k) (S: Finset U): k[X] :=
  ∏ s in S, (X - C (α s))
```

This gives a name to $\prod_{s \in S}(X - \alpha_s)$. Note the conciseness of Lean's version of this formula.

```
let monic_vanishing_at_except = new_definition `
  monic_vanishing_at_except (r:R ring) S s
  = monic_vanishing_at r (S DELETE s)
`;;
```

This gives a name to $\prod_{t \in S-\{s\}}(x - t)$, i.e., $\prod_{t \in S: t \neq s}(x - t)$.

```
variable {k: Type _} [Field k]
variable {U: Type _} [DecidableEq U]

noncomputable def monic_vanishing_at_except
  (α: U → k) (S: Finset U) (s: U): k[X] :=
  monic_vanishing_at α (S.erase s)
```

This gives a name to $\prod_{t \in S - \{s\}}(X - \alpha_t)$.

```
let vanishing_at_except_1_at = new_definition `
  vanishing_at_except_1_at (k:K ring) S s
  = ring_mul(x_ring k)
      (monic_vanishing_at_except k S s)
      (poly_const k
        (ring_inv k
          (poly_eval k s
            (monic_vanishing_at_except k S s))))
`;;
```

This gives a name to $(\prod_{t \in S - \{s\}}(x - t))/(\prod_{t \in S - \{s\}}(s - t))$, where the second product is expressed as the value at s of $(\prod_{t \in S - \{s\}}(x - t))$.

```
variable {k: Type _} [Field k]
variable {U: Type _} [DecidableEq U]

noncomputable def vanishing_at_except_1_at
  (α: U → k) (S: Finset U) (s: U): k[X] :=
  (monic_vanishing_at_except α S s)
  * C (((monic_vanishing_at_except α S s).eval (α s))⁻¹)
```

This gives a name to $(\prod_{t \in S - \{s\}}(X - \alpha_t))/(\prod_{t \in S - \{s\}}(\alpha_s - \alpha_t))$.

```
let interpolator = new_definition `
  interpolator (k:K ring) S v
  = ring_sum(x_ring k) S
      (\s. ring_mul(x_ring k)
             (poly_const k (v s))
             (vanishing_at_except_1_at k S s))
`;;
```

This gives a name to $\sum_{s \in S} \left(v_s (\prod_{t \in S - \{s\}}(x - t))/(\prod_{t \in S - \{s\}}(s - t)) \right)$.

```
variable {k: Type _} [Field k]
variable {U: Type _} [DecidableEq U]

noncomputable def interpolator (α: U → k) (S: Finset U) (r: U → k): k[X] :=
  ∑ s in S, C (r s) * vanishing_at_except_1_at α S s
```

This gives a name to $\sum_{s \in S} \left(v_s (\prod_{t \in S - \{s\}}(X - \alpha_t))/(\prod_{t \in S - \{s\}}(\alpha_s - \alpha_t)) \right)$.

C.5.8 Approximant

The following notation is also defined in both `lightgoppa` and `leangoppa`.

```
let approximant = new_definition `
  approximant (k:K ring) A B t a b
  <=> A IN ring_carrier(x_ring k)
   /\ B IN ring_carrier(x_ring k)
   /\ a IN ring_carrier(x_ring k)
   /\ b IN ring_carrier(x_ring k)
   /\ ring_coprime(x_ring k) (a,b)
   /\ twodeg k a <= 2 EXP t
   /\ 2 EXP t * twodeg k (ring_sub(x_ring k)
                            (ring_mul(x_ring k) a B)
                            (ring_mul(x_ring k) b A))
       < twodeg k A
`;;
```

This defines `approximant k A B t a b` to mean that all of the following are true:
A, B, a, b are in $k[x]$; a, b are coprime in $k[x]$; $2^{\deg a} \leq 2^t$, i.e., $\deg a \leq t$; and $2^t 2^{\deg(aB - bA)} < 2^{\deg A}$, i.e., $t + \deg(aB - bA) < \deg A$.

HOL Light's definition of `ring_coprime` says that the inputs a, b are in the ring (so the `approximant` definition could have skipped saying $a \in k[x]$ and $b \in k[x]$, although this would not necessarily increase clarity) and that each common divisor of a and b is a unit. Various theorems in `lightgoppa` say that, for fields k, `ring_coprime(x_ring k) a b` is equivalent to $ak[x] + bk[x] = k[x]$, which in turn is equivalent to saying that some $u, v \in k[x]$ have $au + bv = 1$.

```
variable {k: Type _} [Field k]

def approximant (A B: k[X]) (t: ℕ) (a b: k[X]): Prop :=
  IsCoprime a b ∧ a.degree ≤ t ∧ t + (a*B-b*A).degree < A.degree
```

This more concisely defines `approximant A B t a b`, where A, B, a, b have type $k[X]$, to mean that all of the following are true: a, b are coprime in $k[X]$; $\deg a \leq t$; and $t + \deg(aB - bA) < \deg A$. Lean defines `IsCoprime a b` as the existence of u, v with $ua + vb = 1$.

Recall that Section 3 said $\deg(aB - bA) < \deg A - t$. The formalizations in `lightgoppa` and `leangoppa` shift t to the other side of the inequality, avoiding division and subtraction respectively.

```
let small_approximant = new_definition `
  small_approximant (k:K ring) A B t a b
  <=> approximant k A B t a b
   /\ twodeg k b < 2 EXP t
`;;
```

This defines `small_approximant k A B t a b` the same way as above but with the extra condition $2^{\deg b} < 2^t$, i.e., $\deg b < t$.

```
variable {k: Type _} [Field k]

def small_approximant (A B: k[X]) (t: ℕ) (a b: k[X]): Prop :=
  approximant A B t a b ∧ b.degree < t
```

Similarly, this defines `small_approximant A B t a b` the same way as above but with the extra condition $\deg b < t$.

C.5.9 Hamming weight

The final definitions listed here are the `lightgoppa` and `leangoppa` definitions of the Hamming weight of a vector.

```
let hamming_weight = new_definition(`
  hamming_weight (r:R ring) (e:X->R) S
  = CARD {x | x IN S /\ ~(e x = ring_0 r)}
`);;
```

This defines `hamming_weight r e S` as the cardinality of the set of $x \in S$ with $e_x \neq 0$ in r.

```
variable {k: Type _} [Field k] [DecidableEq k]
variable {U: Type _} [DecidableEq U]

def hamming_weight (e: U → k) (S: Finset U) :=
  (S.filter (fun s ↦ e s ≠ 0)).card
```

This defines `hamming_weight` similarly: the `fun` builds the function that maps s to True exactly when the s entry in the vector is nonzero, and the `filter` extracts the corresponding subset of S.

C.6 Comparing this paper's theorems to the formalizations

Finally, the following pages compare each theorem statement in the main body of this paper to (1) the formalized theorem statement in HOL Light and (2) the formalized theorem statement in Lean.

This comparison accompanies each step in each formalized theorem statement by a nearby note (in blue for HOL Light or red for Lean) translating the step into normal mathematical language. It is then easy to match up the notes to the original theorem statement at the top of the page. Sometimes the formalized theorem statement skips an assumption from the original theorem statement, making it more general; also, see Appendix C.5.7 regarding how the vector indices $\{1, \ldots, n\}$ are handled in the two formalizations.

A special comment is required for Theorem 4.1.2. One of the stated conclusions is $f = B - bA/a$; with the mathematician's normal interpretation, this implies $a \neq 0$. The proof of $a \neq 0$ uses an earlier conclusion that $A \in ak[x]$, along with a hypothesis saying $A = \prod_i (x - \alpha_i)$. For the formalization in `leangoppa`, the conclusion `f = B-b*(A/a)` (recall that this means $f = B - b\lfloor A/a \rfloor$ in Lean) does not imply $a \neq 0$, since Lean defines division by 0; so the formalization includes a separate conclusion that $a \neq 0$. The formalization in `lightgoppa` similarly has separate conclusions saying $a \neq 0$ and $af = aB - bA$.

Theorem 2.1.2 (direct interpolation). *Let n be a nonnegative integer. Let k be a field. Let $\alpha_1, \ldots, \alpha_n$ be distinct elements of k. Let $r_1, \ldots, r_n$ be elements of k. Define*

$$\varphi = \sum_i r_i \prod_{j \neq i} \frac{x - \alpha_j}{\alpha_i - \alpha_j}.$$

Then $\{f \in k[x] : \deg f < n, (f(\alpha_1), \ldots, f(\alpha_n)) = (r_1, \ldots, r_n)\} = \{\varphi\}.$

```
let interpolator_main = prove(`
  !(k:K ring) S r.
  field k ==>
  S SUBSET ring_carrier k ==>
  FINITE S ==>
  (!s. s IN S ==> r s IN ring_carrier k) ==>
  {f |    f IN ring_carrier(x_ring k)
      /\ twodeg k f < 2 EXP CARD S
      /\ (!t. t IN S ==> poly_eval k t f = r t)}
  = {interpolator k S r}
`, ...
```

<div align="right">

if
$k \subseteq K$ is a ring
and k is a field
and $S \subseteq k$
and S is finite
and $r_s \in k$ for each $s \in S$
then the set of $f \in k[x]$
with $\deg f < \#S$
and $f(t) = r_t$ for each $t \in S$
is the set consisting solely of
$\sum_{s \in S} \left(r_s \left(\prod_{t \in S - \{s\}} (x - t) \right) / \left(\prod_{t \in S - \{s\}} (s - t) \right) \right)$

</div>

```
variable {k: Type _} [Field k]
variable {U: Type _} [DecidableEq U]

theorem interpolator_main
  {α: U → k}
  {S: Finset U}
  {r: U → k}
  (injective: Set.InjOn α S)
: {f:k[X] | degree f < S.card
           ∧ (∀ s, (s ∈ S → f.eval (α s) = r s))}
  = {interpolator α S r} := ...
```

<div align="right">

if k is a field

and α is a function from U to k
and S is a finite subset of U
and r is a function from U to k
and α is injective on S
then the set of $f \in k[X]$ with $\deg f < \#S$
and $f(\alpha_s) = r_s$ for each $s \in S$
is the set consisting solely of
$\sum_{s \in S} \left(r_s \left(\prod_{t \in S - \{s\}} (X - \alpha_t) \right) / \left(\prod_{t \in S - \{s\}} (\alpha_s - \alpha_t) \right) \right)$

</div>

Theorem 3.1.2 (approximants). *Let t be a nonnegative integer. Let k be a field. Let A, B be elements of $k[x]$ with $\deg A > \deg B$. Then there exist $a, b \in k[x]$ such that $\gcd\{a, b\} = 1$, $\deg a \leq t$, $\deg b < t$, and $\deg(aB - bA) < \deg A - t$.*

```
let small_approximant_exists = prove(`                                           if
  !(k:K ring) A B t:num.                           k ⊆ K is a ring and t ∈ ℕ
  field k ==>                                                 and k is a field
  A IN ring_carrier(x_ring k) ==>                              and A ∈ k[x]
  B IN ring_carrier(x_ring k) ==>                              and B ∈ k[x]
  twodeg k B < twodeg k A ==>                       and deg B < deg A then
  ?a b. small_approximant k A B t a b          there exist a, b with a, b ∈ k[x];
`, ...                    gcd{a, b} = 1; deg a ≤ t; t + deg(aB − bA) < deg A; deg b < t
```

```
variable {k: Type _} [Field k]                              if k is a field

theorem small_approximant_exists
  {A B: k[X]}                                              and A, B ∈ k[X]
  (degAB: A.degree > B.degree)                       and deg A > deg B
  {t: ℕ}                                                       and t ∈ ℕ
: ∃ a b, small_approximant A B t a b := ...    then there exist a, b with a, b ∈ k[X];
                          gcd{a, b} = 1; deg a ≤ t; t + deg(aB − bA) < deg A; deg b < t
```

Theorem 3.1.3 (the best-approximation property of approximants). *Let t be a nonnegative integer. Let k be a field. Let A, B, a, b, c, d be elements of $k[x]$ such that $\gcd\{a, b\} = 1$, $\deg a \le t$, $\deg(aB - bA) < \deg A - t$, $\deg c \le t$, and $\deg(cB - dA) < \deg A - t$. Then $(c, d) = (\lambda a, \lambda b)$ for some $\lambda \in k[x]$.*

```
let approximant_best = prove(`                                        if
  !(k:K ring) A B t a b c d.                           k ⊆ K is a ring
  field k ==>                                         and k is a field
  approximant k A B t a b ==>          and t ∈ ℕ; A, B, a, b ∈ k[x]; gcd{a, b} = 1;
  c IN ring_carrier(x_ring k) ==>             deg a ≤ t; t + deg(aB − bA) < deg A
  d IN ring_carrier(x_ring k) ==>                     and c, d ∈ k[x]
  twodeg k c <= 2 EXP t ==>                             and deg c ≤ t
  2 EXP t * twodeg k (ring_sub(x_ring k)
                     (ring_mul(x_ring k) c B)     and t + deg(cB − dA) < deg A
                     (ring_mul(x_ring k) d A))
  < twodeg k A ==>                                          then
  ?L.                                            there exists L with
     L IN ring_carrier(x_ring k)                          L ∈ k[x]
  /\ c = ring_mul(x_ring k) a L                        and c = aL
  /\ d = ring_mul(x_ring k) b L                        and d = bL
`, ...
```

```
variable {k: Type _} [Field k]                            if k is a field

theorem approximant_best
  {A B a b c d: k[X]}                              and A, B, a, b, c, d ∈ k[X]
  {t: ℕ}                                                and t ∈ ℕ
  (appr: approximant A B t a b)                   and gcd{a, b} = 1; deg a ≤ t;
  (cdeg: c.degree ≤ t)                      t + deg(aB − bA) < deg A and deg c ≤ t
  (cBdAdeg: t + (c*B-d*A).degree < A.degree)    and t + deg(cB − dA) < deg A
: ∃ (L: k[X]),                                  then there exists L ∈ k[X] with
  (c = a*L ∧ d = b*L) := ...                          c = aL and d = bL
```

Theorem 4.1.2 (interpolation with errors). *Let n, t be nonnegative integers. Let k be a field. Let $\alpha_1, \ldots, \alpha_n$ be distinct elements of k. Define $A = \prod_i (x - \alpha_i)$. Let B, a, b, f be elements of $k[x]$ with $\gcd\{a, b\} = 1$, $\deg a \le t$, $\deg(aB - bA) < n - t$, and $\deg f < n - 2t$. Define $e = (B(\alpha_1) - f(\alpha_1), \ldots, B(\alpha_n) - f(\alpha_n))$. Assume $\mathrm{wt}\, e \le t$. Then $A \in ak[x]$; $f = B - bA/a$; $\deg(aB - bA) < n - 2t + \deg a$; and $\{i : e_i \ne 0\} = \{i : a(\alpha_i) = 0\}$.*

```
let interpolation_with_errors = prove(`                    if
 !(k:K ring) S A B t:num a b f e aBbA.        k ⊆ K is a ring and t ∈ ℕ
 field k ==>                                        and k is a field
 S SUBSET ring_carrier k ==>                     and S is a subset of k
 FINITE S ==>                                       and S is finite
 A = monic_vanishing_at k S ==>        and A = ∏_{s∈S}(x − s) and
 approximant k A B t a b ==>        t ∈ ℕ; A, B, a, b ∈ k[x]; gcd{a, b} = 1; deg a ≤ t;
 f IN ring_carrier(x_ring k) ==>       t + deg(aB − bA) < deg A and f ∈ k[x]
 2 EXP (2*t) * twodeg k f < twodeg k A ==>       and 2t + deg f < deg A
 (!s. s IN S ==>                         and each s ∈ S has e_s = B(s) − f(s)
  e s = ring_sub k (poly_eval k s B) (poly_eval k s f)) ==>
 aBbA = ring_sub(x_ring k)
         (ring_mul(x_ring k) a B)                   (notation for aB − bA)
         (ring_mul(x_ring k) b A) ==>
 hamming_weight k e S <= t ==>                and wt(e on S) ≤ t then
    ring_divides(x_ring k) a A                         A ∈ ak[x]
 /\ ~(a = ring_0(x_ring k))                          and a ≠ 0
 /\ ring_mul(x_ring k) a f = aBbA             and af = aB − bA and
 /\ 2 EXP (2*t) * twodeg k aBbA                  2t + deg(aB − bA)
    < twodeg k A * twodeg k a                       < deg A + deg a
 /\ {s | s IN S /\ ~(e s = ring_0 k)}          and {s ∈ S : e_s ≠ 0}
    = {s | s IN S /\ poly_eval k s a = ring_0 k}     = {s ∈ S : a(s) = 0}
`, ...
```

```
variable {k: Type _} [Field k] [DecidableEq k]            if k is a field
variable {U: Type _} [DecidableEq U]

theorem interpolation_with_errors
  {α e: U → k}                             and α, e are functions from U to k
  {S: Finset U}                                  and S is a finite subset of U
  {A B a b f: k[X]}                                    and A, B, a, b, f ∈ k[X]
  {t: ℕ}                                                    and t ∈ ℕ
  (injective: Set.InjOn α S)                          and α is injective on S
  (Adef: A = monic_vanishing_at α S)       and A = ∏_{s∈S}(X − α_s) and gcd{a, b} = 1;
  (appr: approximant A B t a b)                deg a ≤ t; t + deg(aB − bA) < deg A
  (fdeg: 2*t + f.degree < A.degree)               and 2t + deg f < deg A
  (edef: (∀ s, s ∈ S →                      and each s ∈ S has e_s = B(α_s) − f(α_s)
              e s = B.eval (α s) - f.eval (α s)))
  (wt: hamming_weight e S ≤ t)                      and wt(e on S) ≤ t
: (  a | A                                            then A ∈ ak[X]
  ∧ a ≠ 0                                               and a ≠ 0
  ∧ f = B-b*(A/a)                                 and f = B − b⌊A/a⌋
  ∧ 2*t + (a*B-b*A).degree < S.card + a.degree     and 2t + deg(aB − bA)
  ∧ { s ∈ S | e s ≠ 0 } = { s ∈ S | a.eval (α s) = 0 }     < #S + deg a
  ) := ...                                  and {s ∈ S : e_s ≠ 0} = {s ∈ S : a(α_s) = 0}
```

Theorem 4.1.3 (checking interpolation with errors). *Let n, t be nonnegative integers. Let k be a field. Let $\alpha_1, \ldots, \alpha_n$ be distinct elements of k. Define $A = \prod_i (x - \alpha_i)$. Let B, a, b, f be elements of $k[x]$ such that $\deg a \le t$, $A \in ak[x]$, $\deg(aB - bA) < n - 2t + \deg a$, and $af = aB - bA$. Then $a \ne 0$; $\deg f < n - 2t$; and $\mathrm{wt}(B(\alpha_1) - f(\alpha_1), \ldots, B(\alpha_n) - f(\alpha_n)) \le t$.*

```
let checking_interpolation_with_errors = prove(`                                    if
  !(k:K ring) S A B t:num a b f.                             k ⊆ K is a ring and t ∈ ℕ
  field k ==>                                                        and k is a field
  S SUBSET ring_carrier k ==>                                           and S ⊆ k
  FINITE S ==>                                                       and S is finite
  A = monic_vanishing_at k S ==>                            and A = ∏_{s∈S}(x − s)
  B IN ring_carrier(x_ring k) ==>                                      and B ∈ k[x]
  a IN ring_carrier(x_ring k) ==>                                      and a ∈ k[x]
  b IN ring_carrier(x_ring k) ==>                                      and b ∈ k[x]
  f IN ring_carrier(x_ring k) ==>                                      and f ∈ k[x]
  twodeg k a <= 2 EXP t ==>                                        and deg a ≤ t
  ring_divides(x_ring k) a A ==>                                   and A ∈ ak[x]
  aBbA = ring_sub(x_ring k)
          (ring_mul(x_ring k) a B)                         (notation for aB − bA)
          (ring_mul(x_ring k) b A) ==>           and 2t + deg(aB − bA) < deg A + deg a
  2 EXP (2*t) * twodeg k aBbA < twodeg k A * twodeg k a ==>                     and
  ring_mul(x_ring k) a f = aBbA ==>                              af = aB − bA then
  (    ~(a = ring_0(x_ring k))                                             a ≠ 0
   /\ 2 EXP (2*t) * twodeg k f < twodeg k A            and 2t + deg f < deg A
   /\ hamming_weight k                            and wt(s ∈ S ↦ B(s) − f(s)) ≤ t
       (\s. ring_sub k (poly_eval k s B) (poly_eval k s f))
       S <= t
  )
`, ...
```

```
variable {k: Type _} [Field k] [DecidableEq k]                          if k is a field
variable {U: Type _} [DecidableEq U]

theorem checking_interpolation_with_errors
  {α: U → k}                                           and α is a function from U to k
  {S: Finset U}                                            and S is a finite subset of U
  {A B a b f: k[X]}                                       and A, B, a, b, f ∈ k[X]
  {t: ℕ}                                                               and t ∈ ℕ
  (injective: Set.InjOn α S)                              and α is injective on S
  (Adef: A = monic_vanishing_at α S)                  and A = ∏_{s∈S}(X − α_s)
  (adeg: a.degree ≤ t)                                             and deg a ≤ t
  (aA: a | A)                                                      and A ∈ ak[X]
  (aBbAdeg: 2*t + (a*B-b*A).degree < S.card + a.degree)                        and
  (af: a*f = a*B-b*A)                        2t + deg(aB − bA) < #S + deg a and af = aB − bA
: (   a ≠ 0                                                          then a ≠ 0
   ∧ 2*t + f.degree < S.card                                 and 2t + deg f < #S
   ∧ hamming_weight (fun s ↦ B.eval (α s) - f.eval (α s)) S ≤ t
  ) := ...                                             and wt(s ∈ S ↦ B(α_s) − f(α_s)) ≤ t
```

Theorem 5.1.2 (Goppa decoding). *Let n, t be nonnegative integers. Let k be a field with $\mathbb{F}_2 \subseteq k$. Let $\alpha_1, \ldots, \alpha_n$ be distinct elements of k. Define $A = \prod_i (x - \alpha_i)$. Let G be an element of $k[x]$ such that $\deg G = 2t$ and $\gcd\{G, A\} = 1$. Let B, a, b be elements of $k[x]$ with $\gcd\{a, b\} = 1$, $\deg a \leq t$, and $\deg(aB - bA) < n - t$. Let A', a' be the derivatives of A, a respectively. Let e be an element of $\mathbb{F}_2^n$ such that $\operatorname{wt} e \leq t$ and*

$$\sum_i \left(\frac{G(\alpha_i) B(\alpha_i)}{A'(\alpha_i)} - e_i \right) \frac{A}{x - \alpha_i} \in Gk[x].$$

Then $e_i = [a(\alpha_i) = 0]$ for all i; $\operatorname{wt} e = \deg a$; $A \in ak[x]$; $Gb - a' \in ak[x]$; and $\deg(aB - bA) < n - 2t + \deg a$.

```
let goppa_decoding = prove(`                                              if
 !(k:K ring) S G A B t a b Aprime aprime aBbA e.            k ⊆ K is a ring
 field k ==>                                                 and k is a field
 S SUBSET ring_carrier k ==>                                     and S ⊆ k
 FINITE S ==>                                                and S is finite
 A = monic_vanishing_at k S ==>                      and A = ∏_{s∈S}(x − s)
 G IN ring_carrier(x_ring k) ==>                              and G ∈ k[x]
 twodeg k G = 2 EXP (2*t) ==>              and deg G = 2t and gcd{G, A} = 1
 ring_coprime(x_ring k) (G,A) ==>           and t ∈ ℕ; A, B, a, b ∈ k[x]; gcd{a, b} = 1;
 approximant k A B t a b ==>                     deg a ≤ t; t + deg(aB − bA) < deg A
 (!s. s IN S ==> (e s = ring_0 k \/ e s = ring_1 k)) ==>                    and
 hamming_weight k e S <= t ==>          each s ∈ S has e_s ∈ {0, 1}, and wt(e on S) ≤ t
 Aprime = x_derivative k A ==>                              (notation for A′)
 aprime = x_derivative k a ==>                              (notation for a′)
 aBbA = ring_sub(x_ring k)
         (ring_mul(x_ring k) a B)                      (notation for aB − bA)
         (ring_mul(x_ring k) b A) ==>
 ring_divides(x_ring k) G (                                                 and
   ring_sum(x_ring k) S
     (\s. ring_mul(x_ring k)
           (poly_const k (
              ring_sub k
                (ring_div k
                   (ring_mul k
                      (poly_eval k s G)
                      (poly_eval k s B))
                   (poly_eval k s Aprime))
                (e s)))
           (monic_vanishing_at_except k S s))) ==>                        then
(   (!s. s IN S ==>                                   each s ∈ S has e_s = [a(s) = 0]
     e s = if poly_eval k s a = ring_0 k
           then ring_1 k else ring_0 k)
 /\ 2 EXP (hamming_weight k e S) = twodeg k a          and wt(e on S) = deg a
 /\ ring_divides(x_ring k) a A                                and A ∈ ak[x]
 /\ ring_divides(x_ring k) a                             and Gb − a′ ∈ ak[x]
     (ring_sub(x_ring k) (ring_mul(x_ring k) G b) aprime)
 /\ 2 EXP (2*t) * twodeg k aBbA < twodeg k A * twodeg k a                   and
 )                                              2t + deg(aB − bA) < deg A + deg a
`, ...
```

Theorem 5.1.2 (Goppa decoding). *Let n, t be nonnegative integers. Let k be a field with $\mathbb{F}_2 \subseteq k$. Let $\alpha_1, \ldots, \alpha_n$ be distinct elements of k. Define $A = \prod_i (x - \alpha_i)$. Let G be an element of $k[x]$ such that $\deg G = 2t$ and $\gcd\{G, A\} = 1$. Let B, a, b be elements of $k[x]$ with $\gcd\{a, b\} = 1$, $\deg a \leq t$, and $\deg(aB - bA) < n - t$. Let A', a' be the derivatives of A, a respectively. Let e be an element of $\mathbb{F}_2^n$ such that $\mathrm{wt}\, e \leq t$ and*

$$\sum_i \left(\frac{G(\alpha_i)B(\alpha_i)}{A'(\alpha_i)} - e_i \right) \frac{A}{x - \alpha_i} \in Gk[x].$$

Then $e_i = [a(\alpha_i) = 0]$ for all i; $\mathrm{wt}\, e = \deg a$; $A \in ak[x]$; $Gb - a' \in ak[x]$; and $\deg(aB - bA) < n - 2t + \deg a$.

```
variable {U: Type _} [DecidableEq U]                                              if
variable {k: Type _} [Field k] [DecidableEq k]                              k is a field

theorem goppa_decoding
  {α e: U → k}                                              and α, e are functions from U to k
  {S: Finset U}                                                   and S is a finite subset of U
  {A B a b G: k[X]}                                                    and A, B, a, b, G ∈ k[X]
  {t: ℕ}                                                                          and t ∈ ℕ
  (injective: Set.InjOn α S)                                          and α is injective on S
  (Adef: A = monic_vanishing_at α S)                             and A = ∏_{s∈S}(X − α_s)
  (Gdeg: G.degree = 2*t)                                                    and deg G = 2t
  (GA: IsCoprime G A)                                   and gcd{G, A} = 1 and gcd{a, b} = 1;
  (appr: approximant A B t a b)                        deg a ≤ t; t + deg(aB − bA) < deg A
  (einF2: (∀ s, s ∈ S → e s = 0 ∨ e s = 1))                  and each s ∈ S has e_s ∈ {0, 1}
  (wt: hamming_weight e S ≤ t)                                         and wt(e on S) ≤ t
  (codeword: G |
    ∑ s in S,                    and ∑_{s∈S} (( (GB)(α_s)/A'(α_s) − e_s ) ∏_{t∈S−{s}}(X − α_t)) ∈ Gk[X]
      C (((G * B).eval (α s) / A.diff.eval (α s) - e s))
      * monic_vanishing_at_except α S s)
: ((∀ s, s ∈ S → e s = if a.eval (α s) = 0 then 1 else 0)                          then
  ∧ hamming_weight e S = a.degree                         each s ∈ S has e_s = [a(α_s) = 0]
  ∧ a | A                                                          and wt(e on S) = deg a
  ∧ a | G*b - a.diff                                                  and A ∈ ak[X]
  ∧ 2*t + (a*B-b*A).degree < S.card + a.degree                   and Gb − a' ∈ ak[X]
  ) := ...                                                  and 2t + deg(aB − bA) < #S + deg a
```

Theorem 5.1.3 (checking Goppa decoding). *Let n, t be nonnegative integers. Let k be a field with $\mathbb{F}_2 \subseteq k$. Let $\alpha_1, \ldots, \alpha_n$ be distinct elements of k. Define $A = \prod_i (x - \alpha_i)$. Let G be an element of $k[x]$ with $\deg G = 2t$. Let B, a, b be elements of $k[x]$ with $A \in ak[x]$, $\deg(aB - bA) < n - 2t + \deg a$, and $Gb - a' \in ak[x]$, where a' is the derivative of a. Define $e \in \mathbb{F}_2^n$ by $e_i = [a(\alpha_i) = 0]$. Then $\operatorname{wt} e = \deg a$ and*

$$\sum_i \left(\frac{G(\alpha_i)B(\alpha_i)}{A'(\alpha_i)} - e_i \right) \frac{A}{x - \alpha_i} \in Gk[x]$$

where A' is the derivative of A.

```
let goppa_checking = prove(`                                    if
 !(k:K ring) S G A B t:num a b Aprime aprime aBbA e.    k ⊆ K is a ring and t ∈ ℕ
 field k ==>                                                and k is a field
 S SUBSET ring_carrier k ==>                                  and S ⊆ k
 FINITE S ==>                                              and S is finite
 G IN ring_carrier(x_ring k) ==>                           and G ∈ k[x]
 A = monic_vanishing_at k S ==>                      and A = ∏_{s∈S}(x − s)
 B IN ring_carrier(x_ring k) ==>                            and B ∈ k[x]
 a IN ring_carrier(x_ring k) ==>                            and a ∈ k[x]
 b IN ring_carrier(x_ring k) ==>                            and b ∈ k[x]
 twodeg k G = 2 EXP (2*t) ==>                             and deg G = 2t
 Aprime = x_derivative k A ==>                         (notation for A′)
 aprime = x_derivative k a ==>                         (notation for a′)
 aBbA = ring_sub(x_ring k)
         (ring_mul(x_ring k) a B)                     (notation for aB − bA)
         (ring_mul(x_ring k) b A) ==>
 ring_divides(x_ring k) a A ==>                            and A ∈ ak[x]
 2 EXP (2*t) * twodeg k aBbA < twodeg k A * twodeg k a ==>        and
 ring_divides(x_ring k) a                       2t + deg(aB − bA) < deg A + deg a
   (ring_sub(x_ring k) (ring_mul(x_ring k) G b) aprime) ==>   and Gb − a′ ∈ ak[x]
 (!s. s IN S ==>                                                 and
     e s = if poly_eval k s a = ring_0 k then ring_1 k
           else ring_0 k) ==>                    each s ∈ S has e_s = [a(s) = 0] then
 (   2 EXP hamming_weight k e S = twodeg k a                wt(e on S) = deg a
 /\ ring_divides(x_ring k) G (                                    and
      ring_sum(x_ring k) S
        (\s. ring_mul(x_ring k)
              (poly_const k (
                ring_sub k                      ∑_{s∈S} ( ( (GB)(s)/A′(s) − e_s ) ∏_{t∈S−{s}} (x − t) )
                  (ring_div k
                    (poly_eval k s
                      (ring_mul(x_ring k) G B))
                    (poly_eval k s Aprime))
                  (e s)))                                    ∈ Gk[x]
              (monic_vanishing_at_except k S s)))))
`, ...
```

Theorem 5.1.3 (checking Goppa decoding). *Let n, t be nonnegative integers. Let k be a field with $\mathbb{F}_2 \subseteq k$. Let $\alpha_1, \ldots, \alpha_n$ be distinct elements of k. Define $A = \prod_i (x - \alpha_i)$. Let G be an element of $k[x]$ with $\deg G = 2t$. Let B, a, b be elements of $k[x]$ with $A \in ak[x]$, $\deg(aB - bA) < n - 2t + \deg a$, and $Gb - a' \in ak[x]$, where a' is the derivative of a. Define $e \in \mathbb{F}_2^n$ by $e_i = [a(\alpha_i) = 0]$. Then $\operatorname{wt} e = \deg a$ and*

$$\sum_i \left(\frac{G(\alpha_i)B(\alpha_i)}{A'(\alpha_i)} - e_i \right) \frac{A}{x - \alpha_i} \in Gk[x]$$

where A' is the derivative of A.

```
variable {U: Type _} [DecidableEq U]                                          if
variable {k: Type _} [Field k] [DecidableEq k]                       k is a field

theorem goppa_checking
  {α e: U → k}                                   and α, e are functions from U to k
  {S: Finset U}                                        and S is a finite subset of U
  {A B a b G: k[X]}                                          and A, B, a, b, G ∈ k[X]
  {t: ℕ}                                                                and t ∈ ℕ
  (injective: Set.InjOn α S)                            and α is injective on S
  (Adef: A = monic_vanishing_at α S)                   and A = ∏_{s∈S}(X − α_s)
  (Gdeg: G.degree = 2*t)                                          and deg G = 2t
  (aA: a | A)                           and A ∈ ak[X] and 2t + deg(aB − bA) < #S + deg a
  (aBbAdeg: 2*t + (a*B-b*A).degree < S.card + a.degree)
  (Gba: a | G*b - a.diff)                                    and Gb − a' ∈ ak[X]
  (ea: ∀ s, s ∈ S →                                              and each s ∈ S
        e s = if a.eval (α s) = 0 then 1 else 0)              has e_s = [a(α_s) = 0]
: (  hamming_weight e S = a.degree                        then wt(e on S) = deg a
  ∧ G | ∑ s in S,
        C (((G * B).eval (α s) / A.diff.eval (α s) - e s))                  and
        * monic_vanishing_at_except α S s
  ) := ...
```

$$\sum_{s \in S} \left(\left(\frac{(GB)(\alpha_s)}{A'(\alpha_s)} - e_s \right) \prod_{t \in S - \{s\}} (X - \alpha_t) \right) \in Gk[X]$$

Theorem 5.4.1 (Goppa parity checks). *Let n be a nonnegative integer. Let k be a field. Let $\alpha_1, \ldots, \alpha_n$ be distinct elements of k. Define $A = \prod_i (x - \alpha_i)$. Let G be a nonzero element of $k[x]$ with $\gcd\{G, A\} = 1$. Let c be an element of k^n. Then $\sum_i c_i A/(x - \alpha_i) \in Gk[x]$ if and only if $\sum_i c_i \alpha_i^j / G(\alpha_i) = 0$ for all nonnegative integers $j < \deg G$.*

```
let goppa_parity = prove(`                                              if
  !(k:K ring) S G A (c:K->K).                    k ⊆ K is a ring and c is a function from K to K
  field k ==>                                                      and k is a field
  S SUBSET ring_carrier k ==>                                       and S ⊆ k
  FINITE S ==>                                                    and S is finite
  A = monic_vanishing_at k S ==>                          and A = ∏_{s∈S}(x − s)
  G IN ring_carrier(x_ring k) ==>                                and G ∈ k[x]
  ~(G = ring_0(x_ring k)) ==>                                     and G ≠ 0
  ring_coprime(x_ring k) (G,A) ==>                        and gcd{G, A} = 1
  (!s. s IN S ==> c s IN ring_carrier k) ==>          and each s ∈ S has c_s ∈ k then:
  (ring_divides(x_ring k) G
    (ring_sum(x_ring k) S
      (\s. ring_mul(x_ring k)
            (poly_const k (c s))
            (monic_vanishing_at_except k S s)))
  <=>
  (!j:num. 2 EXP j < twodeg k G ==>
    ring_sum k S
      (\s. ring_div k
            (ring_mul k (c s) (ring_pow k s j))
            (poly_eval k s G))
    = ring_0 k))
`, ...
```
$$\sum_{s \in S}\left(c_s \prod_{t \in S - \{s\}}(x - t)\right) \in Gk[x]$$
if and only if
each $j \in \mathbb{N}$ with $j < \deg G$
has $\sum_{s \in S} c_s s^j / G(s) = 0$

```
variable {U: Type _} [DecidableEq U]                                    if
variable {k: Type _} [Field k] [DecidableEq k]                    k is a field

theorem goppa_parity
  {α c: U → k}                                       and α, c are functions from U to k
  {S: Finset U}                                        and S is a finite subset of U
  {A G: k[X]}                                                  and A, G ∈ k[X]
  {d: ℕ}                                                          and d ∈ ℕ
  (injective: Set.InjOn α S)                           and α is injective on S
  (Adef: A = monic_vanishing_at α S)               and A = ∏_{s∈S}(X − α_s)
  (Gdeg: G.degree = d)                                          and deg G = d
  (GA: IsCoprime G A)                                      and gcd{G, A} = 1
: (G | ∑ s in S, C (c s) * monic_vanishing_at_except α S s                 then
  ↦ ∀(j:ℕ), j < d →
        ∑ s in S, (c s / G.eval (α s)) * (α s)^j = 0
  ) := ...
```
$$\sum_{s \in S}\left(c_s \prod_{t \in S - \{s\}}(X - \alpha_t)\right)$$
$$\in Gk[X]$$
if and only if each $j \in \mathbb{N}$ with $j < d$ has $\sum_{s \in S}(c_s/G(\alpha_s))\alpha_s^j = 0$

Theorem 6.1.1 (Goppa squaring). *Let n be a nonnegative integer. Let k be a finite field with $\mathbb{F}_2 \subseteq k$. Let $\alpha_1, \ldots, \alpha_n$ be distinct elements of k. Define $A = \prod_i (x - \alpha_i)$. Let g be a squarefree element of $k[x]$ such that $\gcd\{g, A\} = 1$. Let c be an element of $\mathbb{F}_2^n$. Then $\sum_i c_i A/(x - \alpha_i) \in gk[x]$ if and only if $\sum_i c_i A/(x - \alpha_i) \in g^2 k[x]$.*

```
let goppa_squaring = prove(`                                               if
 !(k:K ring) S A g c Q.                          k ⊆ K is a ring
 field k ==>                                      and k is a field
 FINITE(ring_carrier k) ==>                       and k is finite
 ring_char k = 2 ==>                              and k has characteristic 2
 S SUBSET ring_carrier k ==>                      and S ⊆ k
 A = monic_vanishing_at k S ==>                   and A = ∏_{s∈S}(x − s)
 g IN ring_carrier(x_ring k) ==>                  and g ∈ k[x]
 ring_squarefree(x_ring k) g ==>                  and g is squarefree
 ring_coprime(x_ring k) (g,A) ==>                 and gcd{g, A} = 1
 (!s. s IN S ==> (c s = ring_0 k \/ c s = ring_1 k)) ==>   and each s ∈ S
 Q = ring_sum(x_ring k) S                              has c_s ∈ {0,1}
       (\s. ring_mul(x_ring k)           and Q = ∑_{s∈S}(c_s ∏_{t∈S−{s}}(x − t))
            (poly_const k (c s))
            (monic_vanishing_at_except k S s)) ==>             then:
 (
  ring_divides(x_ring k) g Q                               Q ∈ gk[x]
  <=>                                                    if and only if
  ring_divides(x_ring k) (ring_mul(x_ring k) g g) Q       Q ∈ g²k[x]
 )
`, ...
```

```
variable {U: Type _} [DecidableEq U]                                      if
variable {k: Type _} [Field k] [CharP k 2] [DecidableEq k]    k is a field
                                                    and k has characteristic 2

theorem goppa_squaring
  [Fintype k]                                              and k is finite
  {α c: U → k}                              and α, c are functions from U to k
  {S: Finset U}                              and S is a finite subset of U
  {A Q g: k[X]}                                       and A, Q, g ∈ k[X]
  (Adef: A = monic_vanishing_at α S)              and A = ∏_{s∈S}(X − α_s)
  (sqfree: Squarefree g)                           and g is squarefree
  (gA: IsCoprime g A)                              and gcd{g, A} = 1
  (cinF2: (∀ s, s ∈ S → c s = 0 v c s = 1))   and each s ∈ S has c_s ∈ {0,1}
  (Qdef: Q = (∑ s in S, C (c s) * monic_vanishing_at_except α S s))       and
: (g | Q ↦ g^2 | Q) := ...        Q = ∑_{s∈S}(c_s ∏_{t∈S−{s}}(X − α_t))
                                                                       then:
                                   Q ∈ gk[X] if and only if Q ∈ g²k[X]
```

Theorem 7.1 (checking Goppa decoding for received words in $\mathbb{F}_2^n$). *Let n, t be nonnegative integers. Let k be a field with $\mathbb{F}_2 \subseteq k$. Let $\alpha_1, \ldots, \alpha_n$ be distinct elements of k. Define $A = \prod_i (x - \alpha_i)$. Let g be an element of $k[x]$ such that $\deg g = t$ and $\gcd\{g, A\} = 1$. Let B, a, b be elements of $k[x]$ with $\gcd\{a, b\} = 1$, $\deg a \leq t$, $A \in ak[x]$, and $\deg(aB - bA) < n - 2t + \deg a$. Assume that $g(\alpha_i)^2 B(\alpha_i)/A'(\alpha_i) \in \mathbb{F}_2$ for all i, where A' is the derivative of A. Define $e \in \mathbb{F}_2^n$ by $e_i = [a(\alpha_i) = 0]$. Then $g^2 b - a' \in ak[x]$, where a' is the derivative of a. Furthermore $\operatorname{wt} e = \deg a$ and*

$$\sum_i \left(\frac{g(\alpha_i)^2 B(\alpha_i)}{A'(\alpha_i)} - e_i \right) \frac{A}{x - \alpha_i} \in g^2 k[x].$$

```
let goppa_checking_2 = prove(`                                    if k ⊆ K is a ring
 !(k:K ring) S g G A B t:num a b Aprime aprime aBbA e r.                and t ∈ ℕ
 field k ==>                                                          and k is a field
 ring_char k = 2 ==>                                           and k has characteristic 2
 S SUBSET ring_carrier k ==>                                              and S ⊆ k
 FINITE S ==>                                                           and S is finite
 A = monic_vanishing_at k S ==>                              and A = ∏_{s∈S}(x − s)
 Aprime = x_derivative k A ==>                                        (notation for A′)
 g IN ring_carrier(x_ring k) ==>                                        and g ∈ k[x]
 twodeg k g = 2 EXP t ==>                                              and deg g = t
 G = ring_mul(x_ring k) g g ==>                                     (notation for g²)
 ring_coprime(x_ring k) (g,A) ==>                                    and gcd{g, A} = 1
 B IN ring_carrier(x_ring k) ==>                                        and B ∈ k[x]
 a IN ring_carrier(x_ring k) ==>                                        and a ∈ k[x]
 aprime = x_derivative k a ==>                                       (notation for a′)
 b IN ring_carrier(x_ring k) ==>                                        and b ∈ k[x]
 ring_coprime(x_ring k) (a,b) ==>                                    and gcd{a, b} = 1
 twodeg k a <= 2 EXP t ==>                                             and deg a ≤ t
 ring_divides(x_ring k) a A ==>                                        and A ∈ ak[x]
 aBbA = ring_sub(x_ring k)
        (ring_mul(x_ring k) a B)                              (notation for aB − bA)
        (ring_mul(x_ring k) b A) ==>                          and 2t + deg(aB − bA)
 2 EXP (2*t) * twodeg k aBbA < twodeg k A * twodeg k a ==>         < deg A + deg a
 (!s. s IN S ==>                                                    and each s ∈ S
  r s = ring_div k                                             has r_s = (g²B)(s)/A′(s)
        (poly_eval k s (ring_mul(x_ring k) G B))
        (poly_eval k s Aprime)) ==>
 (!s. s IN S ==> ring_pow k (r s) 2 = r s) ==>           and each s ∈ S has r_s² = r_s
 (!s. s IN S ==>                                     and each s ∈ S has e_s = [a(s) = 0]
     e s = if poly_eval k s a = ring_0 k then ring_1 k
           else ring_0 k) ==>                                                   then
 (   ring_divides(x_ring k) a                                          g²b − a′ ∈ ak[x]
     (ring_sub(x_ring k) (ring_mul(x_ring k) G b) aprime)
 /\ 2 EXP hamming_weight k e S = twodeg k a                   and wt(e on S) = deg a
 /\ ring_divides(x_ring k) G (                                                    and
     ring_sum(x_ring k) S                            ∑_{s∈S}((r_s − e_s) ∏_{t∈S−{s}}(x − t)) ∈ g²k[x]
       (\s. ring_mul(x_ring k)
              (poly_const k (ring_sub k (r s) (e s)))
              (monic_vanishing_at_except k S s))))
`, ...
```

Theorem 7.1 (checking Goppa decoding for received words in $\mathbb{F}_2^n$). *Let n, t be nonnegative integers. Let k be a field with $\mathbb{F}_2 \subseteq k$. Let $\alpha_1, \ldots, \alpha_n$ be distinct elements of k. Define $A = \prod_i (x - \alpha_i)$. Let g be an element of $k[x]$ such that $\deg g = t$ and $\gcd\{g, A\} = 1$. Let B, a, b be elements of $k[x]$ with $\gcd\{a, b\} = 1$, $\deg a \leq t$, $A \in ak[x]$, and $\deg(aB - bA) < n - 2t + \deg a$. Assume that $g(\alpha_i)^2 B(\alpha_i)/A'(\alpha_i) \in \mathbb{F}_2$ for all i, where A' is the derivative of A. Define $e \in \mathbb{F}_2^n$ by $e_i = [a(\alpha_i) = 0]$. Then $g^2 b - a' \in ak[x]$, where a' is the derivative of a. Furthermore $\operatorname{wt} e = \deg a$ and*

$$\sum_i \left(\frac{g(\alpha_i)^2 B(\alpha_i)}{A'(\alpha_i)} - e_i \right) \frac{A}{x - \alpha_i} \in g^2 k[x].$$

```
variable {U: Type _} [DecidableEq U]                                    if
variable {k: Type _} [Field k] [CharP k 2] [DecidableEq k]       k is a field
                                                         and k has characteristic 2

theorem goppa_checking_2
  {α e r: U → k}                                  and α, e, r are functions from U to k
  {S: Finset U}                                          and S is a finite subset of U
  {A B a b g: k[X]}                                          and A, B, a, b, g ∈ k[X]
  {t: ℕ}                                                                 and t ∈ ℕ
  (injective: Set.InjOn α S)                              and α is injective on S
  (Adef: A = monic_vanishing_at α S)                 and A = ∏_{s∈S}(X − α_s)
  (gdeg: g.degree = t)                                            and deg g = t
  (gA: IsCoprime g A)                                       and gcd{g, A} = 1
  (ab: IsCoprime a b)                                       and gcd{a, b} = 1
  (adeg: a.degree ≤ t)                                             and deg a ≤ t
  (aA: a | A)                                 and A ∈ ak[X] and 2t + deg(aB − bA)
  (aBbAdeg: 2*t + (a*B-b*A).degree < S.card + a.degree)    < #S + deg a
  (rdef: ∀ s, s ∈ S →                            and each s ∈ S has r_s = (g²B)(α_s)/A'(α_s)
          r s = (g^2*B).eval (α s) / A.diff.eval (α s))
  (rsq: ∀ s, s ∈ S → (r s)^2 = r s)                  and each s ∈ S has r_s² = r_s
  (ea: (∀ s, s ∈ S →                               and each s ∈ S has e_s = [a(α_s) = 0]
          e s = if a.eval (α s) = 0 then 1 else 0))
: (  a | g^2*b - a.diff                                   then g²b − a' ∈ ak[X]
  ∧ hamming_weight e S = a.degree                   and wt(e on S) = deg a
  ∧ g^2 | ∑ s in S,
          C (((g^2 * B).eval (α s) / A.diff.eval (α s) - e s))       and
          * monic_vanishing_at_except α S s
  ) := ...
```

$$\sum_{s \in S} \left(\left(\frac{(g^2 B)(\alpha_s)}{A'(\alpha_s)} - e_s \right) \prod_{t \in S - \{s\}} (X - \alpha_t) \right) \in g^2 k[X]$$