Understanding binary-Goppa decoding

Daniel J. Bernstein[1,2,3]

[1] Department of Computer Science, University of Illinois at Chicago, USA
[2] Horst Görtz Institute for IT Security, Ruhr University Bochum, Germany
[3] Research Center for Information Technology Innovation, Academia Sinica, Taipei
`djb@cr.yp.to`

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. 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 6.2 and Theorem 6.4 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 $\mathbf{F}_2^n$, 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 3, 4, 5, and 6 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

This work was funded by the Intel Crypto Frontiers Research Center; by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) as part of the Excellence Strategy of the German Federal and State Governments—EXC 2092 CASA—390781972 "Cyber Security in the Age of Large-Scale Adversaries"; by the U.S. National Science Foundation under grant 2037867; and by the Cisco University Research Program. "Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation" (or other funding agencies). Permanent ID of this document: `8561e73dab75d01a6dd6bf542594ddac03cdbe6e`. Date: 2022.03.20.

descriptions appear in, e.g., van Tilborg's coding-theory textbook [77, Section 4.5, "A decoding algorithm"], a Preneel–Bosselaers–Govaerts–Vandewalle paper on a software implementation of the McEliece cryptosystem [67, Section 5.3], a Ghosh–Verbauwhede paper on a constant-time hardware implementation of the cryptosystem [42, Algorithm 3], and the Overbeck–Sendrier survey of code-based cryptography [61, pages 139–140].

All of these sources—and many more—are describing an algorithm introduced by Patterson [64, Section V] to correct t errors for binary Goppa codes defined by squarefree degree-t polynomials. McEliece's paper introducing the McEliece cryptosystem [56] 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 [74] and Guruswami–Sudan [45], 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 [43, 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 [14], Chou [30], and Chen–Chou [28]. 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 [49, 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 [15, Section 5], which also generalizes from $\mathbf{F}_2$ to $\mathbf{F}_q$). 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 I haven't found 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 3.1, 4.1, 5.4, and 6.4), and an algorithm statement (Algorithms 3.3, 4.4, 5.3, and 6.2).

Each algorithm layer is presented as a script in the Sage [76] 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 (`lagrange_polynomial`), the Berlekamp–Massey algorithm, etc.

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. This paper includes various efficient characterizations of vectors having at most t errors (Theorems 5.5, 6.5, and 7.4), and an analysis of safe options for recognizing valid ciphertexts (Sections 8.3 and 8.4).

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

1.3. Acknowledgments. Thanks to Tanja Lange and Alex Pellegrini for their comments.

2 Polynomials

This section reviews the definition of the polynomial ring $k[x]$ over a field k and the necessary properties of polynomials.

2.1. 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 $\mathbf{Z}$, the set of integers with its usual $0, 1, -, +, \cdot$.

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

2.2. 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 $\varphi(\sum_i r_i) = \sum_i \varphi(r_i)$.

2.3. 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 $\{uq + vr : q, r \in R\}$.

2.4. 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***.

2.5. 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 $\mathbf{F}_2$. As another example, the set $\mathbf{Q}$ of rational numbers with its usual $0, 1, -, +, \cdot$ is a field.

2.6. 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 $\alpha, \beta \in k$.

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

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

2.9. Polynomials. Let k be a field. By definition $k[x]$ is the set of vectors $(f_0, f_1, \ldots)$ 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.

2.10. 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 ("convolution") maps $(f_0, f_1, f_2, f_3, \dots), (g_0, g_1, g_2, g_3, \dots)$ to $(f_0g_0, f_0g_1 + f_1g_0, f_0g_2 + f_1g_1 + f_2g_0, f_0g_3 + f_1g_2 + f_2g_1 + f_3g_0, \dots)$.

2.11. 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 $k[x]$.

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

2.13. 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 $(\alpha, 0, 0, \dots)(f_0, f_1, \dots)$.

2.14. 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.

2.15. Coefficients. If $f = (f_0, f_1, \dots) \in k[x]$ and $i \in \mathbf{Z}$ 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 4.1 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.

2.16. 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 $\deg(f \pm g) \leq \max\{\deg f, \deg g\}$.

2.17. 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.

2.18. 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 $(f \cdot g)(\alpha) = f(\alpha) \cdot g(\alpha)$.

2.19. 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., $f \in (x-\alpha)k[x]$.

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

2.21. Transposed Vandermonde invertibility. If $\alpha_1, \ldots, \alpha_n \in k$ are distinct, and $c_1, \ldots, c_n \in k$ satisfy the equations $\sum_j c_j \alpha_j^i = 0$ for all $i \in \{0, 1, \ldots, n-1\}$, then $(c_1, \ldots, c_n) = (0, 0, \ldots, 0)$.

2.22. Derivatives. If $f = (f_0, f_1, f_2, f_3, \ldots) \in k[x]$ then the **derivative of f** is $(f_1, 2f_2, 3f_3, \ldots)$. 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 [73].

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

2.23. 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.

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

2.25. 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 $\gcd\{f/d, g/d\} = 1$.

2.26. 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.

3 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 k. See Section 5 for a generalization that handles as many as t errors in the input vector, at the expense of requiring $\deg f < n - 2t$.

The formula for φ in Theorem 3.1 is usually called the "Lagrange interpolation formula". However, Waring [78] published the same formula earlier.

```
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 3.3. 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; $(r_1, \ldots, r_n) \in k^n$.

Theorem 3.1 (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. □

3.2. An interpolation algorithm. Algorithm 3.3 interpolates a polynomial from its values, using the φ formula in Theorem 3.1.

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.

3.4. More interpolation algorithms. An older interpolation method (due to Newton), the "divided differences" method, recursively interpolates a polynomial $g \in k[x]$ with $(g(\alpha_2), \ldots, g(\alpha_n)) = ((r_2 - r_1)/(\alpha_2 - \alpha_1), \ldots, (r_n - r_1)/(\alpha_n - \alpha_1))$, and then takes $f = r_1 + (x - \alpha_1)g$. This 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 $\sum_{0 \leq i < n} f_i x^i$ by $\sum_{0 \leq i < n} g_i x^i$), division, and other basic operations. See generally [6].

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 [10].

4 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.)

Theorem 4.1 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 4.2, any lattice vector of degree at most $\deg A - t - 1$ must be a multiple of this particular vector.

Why 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 fast algorithms in this area.

Theorem 4.1 (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 $\deg(aB - bA) < \deg A - t$.*

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 $A =$

$A_nx^n + A_{n-1}x^{n-1} + \cdots$ and $B = B_{n-1}x^{n-1} + \cdots$ then the output entries are $a_tB_{n-1} - b_{t-1}A_n$, $a_tB_{n-2} + a_{t-1}B_{n-1} - b_{t-1}A_{n-1} - b_{t-2}A_n$, 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 $(a, b) \neq (0, 0)$, $\deg a \le t$, $\deg b < t$, 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 $\deg(aB - bA)$. □

Theorem 4.2 (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 $\lambda \in k[x]$.*

Equivalently (in the case $c \neq 0$), if $\deg a \le t$, if $\deg c \le t$, $\deg(B/A - b/a) < -t - \deg a$, 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 4.1, one must take larger-degree denominators.

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 $a \neq 0$, or for d/b if $b \neq 0$; in both cases $(c, d) = (a\lambda, b\lambda)$ as claimed. □

4.3. An approximant algorithm. Algorithm 4.4 computes a, b from t, k, A, B. This algorithm works in the same way as the proof of Theorem 4.1, 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.

4.5. 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 [6, Section 21]. Applying a sequence of $2t$ “divsteps”, taking $n = 2t$ in [16, Theorems A.1 and A.2], uses $t^{1+o(1)}$ operations with the “jump” algorithms in [16] while avoiding the timing variability of polynomial division.

4.6. Approximants as ratios. With the following definition, the conclusion of Theorem 4.1 is that there is an approximant to B/A at degree t. This definition would also slightly compress the statement of Theorem 4.2 and the statements of some theorems later in this paper. For the benefit of a reader looking at

```
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 4.4. 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 `[...]+[...]` in Sage is concatenation of lists.

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 5.8 below.

Definition 4.7. *Let k be a field. Let A, B be elements of $k((x^{-1}))$ with $A \ne 0$. Let t be a nonnegative integer. If $(a, b) \in k[x] \times k[x]$ satisfy $\gcd\{a, b\} = 1$, $\deg a \le t$, 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 $\deg(aB - bA)$.

4.8. History. Euclid's subtractive algorithm [36, 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 [50, 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 [71, page 241 of original, page 123 of cited PDF] computed polynomial gcd by iterating $(A, B) \mapsto (B, A \bmod B)$.

According to [22, 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 [53] used convergents to continued fractions of rational functions as small-height approximations to power series. Kronecker [51, 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 4.1 to Lagrange, but the short proof to Kronecker. Small-height approximations to power series are often miscredited to [62] under the name "Padé approximants".

An earlier paper of Lagrange [52, 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 [41] 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 $\deg a \le t$ 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 4.2. For the more complicated setting of list-decoding algorithms, short vectors in arbitrary-rank lattices often appear as an abstraction layer; see, e.g., [21], [7], [32], [8], and [9].

5 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 3, and is used as a subroutine for handling the general case.

5.1. Hamming weight. If $e \in k^n$, where n is a nonnegative integer, then "$\text{wt}\, e$" 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 $\text{wt}(r - c) \le t$, i.e., $\text{wt}(r_1 - f(\alpha_1), \ldots, r_n - f(\alpha_n)) \le t$.

5.2. An interpolation-with-errors algorithm. Algorithm 5.3 recovers f given (n, t, k, α, r), where r is a vector with $\text{wt}(r_1 - f(\alpha_1), \ldots, r_n - f(\alpha_n)) \le t$. The algorithm has three steps:

- Interpolate the input vector into a polynomial $B \in k[x]$ with $\deg B < n$, as in Theorem 3.1.
- Compute an approximant b/a to B/A at degree t as in Theorem 4.1, where $A = \prod_i (x - \alpha_i)$.
- Compute $f = B - bA/a$. Theorem 5.4 says that this works.

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

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 5.5.

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 5.3 includes a separate test for $aB - bA = 0$.

Theorem 5.4 (interpolation with errors). *Let n, t be nonnegative integers. Let k be a field. Let $\alpha_1, \dots, \alpha_n$ 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 $e = (B(\alpha_1) - f(\alpha_1), \dots, B(\alpha_n) - f(\alpha_n))$. Assume $\operatorname{wt} e \le t$. Then $A \in ak[x]$; $f = B - bA/a$; $\deg(aB - bA) < n - 2t + \deg a$; and $\{i : e_i \ne 0\} = \{i : a(\alpha_i) = 0\}$.*

Proof. Define $E = \prod_{i:e_i=0}(x - \alpha_i)$ and $c = \prod_{i:e_i \ne 0}(x - \alpha_i)$. 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 $\alpha_1, \dots, \alpha_n$ are distinct.

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

The conditions of Theorem 4.2 are satisfied: A, B, a, b, c, d are elements of $k[x]$ with $\gcd\{a, b\} = 1$, $\deg a \le t$, $\deg(aB - bA) < \deg A - t$, $\deg c \le t$, and $\deg(cB - dA) < \deg A - t$.

Hence $(c, d) = (\lambda a, \lambda b)$ for some $\lambda \in k[x]$ by Theorem 4.2. By construction $c \ne 0$, so $a \ne 0$. 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 $e_i \ne 0$ exactly when $a(\alpha_i) = 0$: $A(\alpha_i) = 0$ so if $a(\alpha_i) = 0$ then, by the Bernoulli rule, $(A/a)(\alpha_i) = A'(\alpha_i)/a'(\alpha_i) \ne 0$ 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 5.5 (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 be elements of $k[x]$ such that $\gcd\{a, b\} = 1$, $\deg a \leq t$, $A \in ak[x]$, and $\deg(aB - bA) < n - 2t + \deg a$. Define $f = B - bA/a$. Then $f \in k[x]$; $\deg f < n - 2t$; and $\operatorname{wt}(B(\alpha_1) - f(\alpha_1), \ldots, B(\alpha_n) - f(\alpha_n)) \leq t$.*

The condition $\deg(aB - bA) < n - 2t + \deg a$ here cannot be weakened to $\deg(aB - bA) < n - t$. 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 $A \in ak[x]$, and $a \neq 0$ since $A \neq 0$, so $f = B - bA/a \in k[x]$. Also $\deg f = \deg(aB - bA) - \deg a < n - 2t$.

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 $\operatorname{wt}(B(\alpha_1) - f(\alpha_1), \ldots, B(\alpha_n) - f(\alpha_n)) \leq t$. □

5.6. 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 $\mathbf{F}_2 \subseteq k$ and that $e \in \mathbf{F}_2^n$ 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 5.4.

5.7. Reed–Solomon codes. The set of vectors $(f(\alpha_1), \ldots, f(\alpha_n))$ is called a Reed–Solomon code; see Definition 5.8. This is a subspace of the k-vector space k^n. Each vector is called a codeword. With this terminology, Theorem 5.4 recovers a Reed–Solomon codeword from a vector that matches the codeword on at least $n - t$ positions. One can similarly define Goppa codes in Section 6.

Definition 5.8. *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 $(\alpha_1, \ldots, \alpha_n)$**.

5.9. History. Reed–Solomon [68] 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" [66] 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 [38, Chapter 4] (see also [39]) introduced a polynomial-time decoding algorithm for Reed–Solomon codes. Forney's algorithm simplified and extended an algorithm by Gorenstein and Zierler [44], which handled the special case $\{\alpha_1, \ldots, \alpha_n\} = k^*$. The latter algorithm extended an algorithm by Peterson [65], which handled the following special case: $\mathbf{F}_2 \subseteq k$, each $f(\alpha_j)$ is in $\mathbf{F}_2$, and $e \in \mathbf{F}_2^n$.

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 [72], 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 [5] 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 [55] 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 [75] 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 [6, Section 21.6] for history) and for all other necessary subroutines; these algorithms were applied to Reed–Solomon decoding by Justesen [48] and independently Sarwate [69], 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 [57] pointed out that "shift register synthesis" is the same as the problem of finding approximants, the problem of finding (a, b) in Theorem 4.1. See also [79] for how the result after each polynomial division inside an extended-gcd computation appears inside in the Berlekamp–Massey algorithm; [34] for an extended-gcd explanation of all further quantities inside the Berlekamp–Massey algorithm; and [16, 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 5.6 regarding different choices of (A, B); the choice of (A, B) in Theorem 5.4 was published by Shiozaki [70, Section III] and later Gao [40]. For the problem of computing (a, b) in Theorem 4.1, 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.

6 Binary-Goppa decoding

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

6.1. An algorithm to decode binary Goppa codes. Algorithm 6.2 allows any $r \in k^n$ as an input vector, and returns the unique $e \in \mathbf{F}_2^n$ with $\operatorname{wt} e \le t$ 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 $B(\alpha_i) = r_i A'(\alpha_i)/g(\alpha_i)^2$. Here A' is the derivative of A, so $A'(\alpha_j) = \prod_{i \ne j} (\alpha_j - \alpha_i)$.
- Compute an approximant b/a to B/A at degree t as in Theorem 4.1.
- Compute $\{i : e_i = 1\}$ as $\{i : a(\alpha_i) = 0\}$. Theorem 6.4 says that this works.

The algorithm recognizes the `None` case using tests stated in Theorem 6.5: e exists exactly when $A \in ak[x]$, $g^2 b - a' \in ak[x]$, and $\deg(aB - bA) < n - 2t + \deg a$. Here a' is the derivative of a. The test $g^2 b - a' \in ak[x]$ can be skipped for inputs $r \in \mathbf{F}_2^n$; see Section 7.

```
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() == t and g.is_squarefree()
  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])^2 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^2*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 6.2. Algorithm to compute the unique $e \in \mathbf{F}_2^n$ with $\sum_i (r_i - e_i)A/(x - \alpha_i) \in gk[x]$ and $\operatorname{wt} e \le t$, or `None` if no such e exists. Here $A = \prod_i (x - \alpha_i) \in k[x]$. Inputs: integer $n \ge 0$; integer $t \ge 0$; field k containing $\mathbf{F}_2$; $(\alpha_1, \dots, \alpha_n) \in k^n$ with distinct entries; squarefree $g \in k[x]$ with $\deg g = t$ and each $g(\alpha_j)$ nonzero; $r \in k^n$.

Theorem 6.3 (Goppa squaring). *Let n be a nonnegative integer. Let k be a finite field with $\mathbf{F}_2 \subseteq k$. Let $\alpha_1, \dots, \alpha_n$ be distinct elements of k. Define $A = \prod_i (x - \alpha_i)$. Let g be a squarefree element of $k[x]$ such that $\gcd\{g, A\} = 1$. Let c be an element of $\mathbf{F}_2^n$. 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 6.3 in [43, 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 $\mathbf{F}_2$, 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 $C' \in gk[x]$. □

Theorem 6.4 (Goppa decoding). *Let n, t be nonnegative integers. Let k be a finite field with $\mathbf{F}_2 \subseteq k$. Let $\alpha_1, \dots, \alpha_n$ be distinct elements of k. Define*

$A = \prod_i (x - \alpha_i)$. Let g be a squarefree element of $k[x]$ such that $\deg g = t$ and $\gcd\{g, A\} = 1$. Let B, a, b be elements of $k[x]$ with $\gcd\{a, b\} = 1$, $\deg a \le t$, and $\deg(aB - bA) < n - t$. Let A', a' be the derivatives of A, a respectively. Let e be an element of $\mathbf{F}_2^n$ such that $\operatorname{wt} e \le t$ and

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

Then $e_i = [a(\alpha_i) = 0]$ for all i; $\operatorname{wt} e = \deg a$; $A \in ak[x]$; $g^2 b - a' \in ak[x]$; 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 $c_i = (g^2 B)(\alpha_i)/A'(\alpha_i) - e_i$. By assumption $\sum_i c_i A/(x - \alpha_i) \in gk[x]$, so $\sum_i c_i A/(x - \alpha_i) \in g^2 k[x]$ by Theorem 6.3. Write $f = (\sum_i c_i A/(x - \alpha_i))/g^2$. Then $f \in k[x]$ and $\deg f < n - 2t$, since $\deg A = n$ and $\deg g = t$.

Notice that $(g^2 f)(\alpha_i)/A'(\alpha_i) = c_i$. Indeed,

$$\sum_i c_i A'(\alpha_i) \prod_{j \ne i} \frac{x - \alpha_j}{\alpha_i - \alpha_j} = \sum_i c_i \prod_{j \ne i} (x - \alpha_j) = \sum_i c_i \frac{A}{x - \alpha_i} = g^2 f,$$

so $c_i A'(\alpha_i) = (g^2 f)(\alpha_i)$ by Theorem 3.1.

Now $(g^2 B - g^2 f)(\alpha_i)/A'(\alpha_i) = e_i$, so $(B - f)(\alpha_i) = e_i A'(\alpha_i)/g(\alpha_i)^2$, which is nonzero exactly when $e_i \ne 0$, so $\operatorname{wt}((B - f)(\alpha_1), \dots, (B - f)(\alpha_n)) = \operatorname{wt} e \le t$.

By Theorem 5.4, $A \in ak[x]$; $f = B - bA/a$; $\deg(aB - bA) < n - 2t + \deg a$; and $\{i : (B - f)(\alpha_i) \ne 0\} = \{i : a(\alpha_i) = 0\}$. Hence $\{i : e_i \ne 0\} = \{i : a(\alpha_i) = 0\}$. By assumption $e_i \in \mathbf{F}_2$, so $e_i \ne 0$ exactly when $e_i = 1$, so $e_i = [a(\alpha_i) = 0]$. Also, $\operatorname{wt} e$ 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 $a'(\alpha_i) \ne 0$ and, by Bernoulli's rule, $(A/a)(\alpha_i) = A'(\alpha_i)/a'(\alpha_i)$, so

$$1 = e_i = \frac{g(\alpha_i)^2 (B - f)(\alpha_i)}{A'(\alpha_i)} = \frac{g(\alpha_i)^2 (bA/a)(\alpha_i)}{A'(\alpha_i)} = \frac{g(\alpha_i)^2 b(\alpha_i)}{a'(\alpha_i)},$$

so $(g^2 b - a')(\alpha_i) = 0$. Hence $g^2 b - a' \in ak[x]$. □

Theorem 6.5 (checking Goppa decoding). *Let n, t be nonnegative integers. Let k be a finite field with $\mathbf{F}_2 \subseteq k$. Let $\alpha_1, \dots, \alpha_n$ be distinct elements of k. Define $A = \prod_i (x - \alpha_i)$. Let g be an element of $k[x]$ such that $\deg g = t$ and $\gcd\{g, A\} = 1$. Let B, a, b be elements of $k[x]$ with $\gcd\{a, b\} = 1$, $\deg a \le t$, $A \in ak[x]$, $\deg(aB - bA) < n - 2t + \deg a$, and $g^2 b - a' \in ak[x]$, where a' is the derivative of a. Define $e \in \mathbf{F}_2^n$ by $e_i = [a(\alpha_i) = 0]$. Then $\operatorname{wt} e = \deg a$ and*

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

where A' is the derivative of A.

One can replace g^2 in this theorem by any polynomial of degree $2t$ with no roots among $\alpha_1, \ldots, \alpha_n$, but the extra generality is not useful for this paper.

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

Observe that $e_i = (g^2 bA/a)(\alpha_i)/A'(\alpha_i) = (g^2 B - g^2 f)(\alpha_i)/A'(\alpha_i)$:

- If $a(\alpha_i) = 0$ then $a'(\alpha_i) \neq 0$ and $(A/a)(\alpha_i)/A'(\alpha_i) = 1/a'(\alpha_i)$. Also $g^2 b - a' \in ak[x]$ so $(g^2 b)(\alpha_i) = a'(\alpha_i)$. Multiply: $(g^2 bA/a)(\alpha_i)/A'(\alpha_i) = 1 = e_i$.
- If $a(\alpha_i) \neq 0$ then $(g^2 bA/a)(\alpha_i)/A'(\alpha_i) = 0 = e_i$ since $A(\alpha_i) = 0$.

Hence

$$\begin{aligned}\sum_i \left(\frac{(g^2 B)(\alpha_i)}{A'(\alpha_i)} - e_i\right) \frac{A}{x - \alpha_i} &= \sum_i \frac{(g^2 f)(\alpha_i)}{A'(\alpha_i)} \frac{A}{x - \alpha_i} \\ &= \sum_i (g^2 f)(\alpha_i) \prod_{j \neq i} \frac{x - \alpha_i}{\alpha_j - \alpha_i} = g^2 f \in g^2 k[x]\end{aligned}$$

by Theorem 3.1.

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

6.6. 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 5.

The main point of the proof of Theorem 6.4 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 $\beta_j = g(\alpha_j)^2/A'(\alpha_j)$. Any Reed–Solomon decoder can thus be used as a Goppa decoder.

Algorithm 6.2 starts from this approach but streamlines the computation of e, taking advantage of the assumption $e \in \mathbf{F}_2^n$. 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 $e \in \mathbf{F}_2^n$, 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 $\beta_i = g(\alpha_i)^2/A'(\alpha_i)$ to obtain $e_i = g(\alpha_i)^2 b(\alpha_i)/a'(\alpha_i)$ when $a(\alpha_i) = 0$. Streamlining this to $e_i = 1$ might not seem helpful in Algorithm 6.2, since the algorithm checks $g^2 b - a' \in ak[x]$ anyway, and the obvious way to do this is to check $g(\alpha_i)^2 b(\alpha_i) = a'(\alpha_i)$; but Section 7 shows that this check can simply be skipped when the input vector is in $\mathbf{F}_2^n$.

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 $\mathbf{F}_2^n$, not merely in k^n, then the test $g^2b - a' \in ak[x]$ can be removed from Algorithm 6.2.

7.1. Overview of the logic. Theorem 7.2 rewrites $\sum_i c_i A/(x-\alpha_i) \in gk[x]$ as the following system of linear equations: $\sum_i c_i/g(\alpha_i) = 0$, $\sum_i c_i\alpha_i/g(\alpha_i) = 0$, and so on through $\sum_i c_i\alpha_i^{t-1}/g(\alpha_i) = 0$. Ths theorem is from Goppa [43, Section 3], and is used inside the standard method of computing McEliece keys.

Theorem 7.3 uses this system of linear equations to show that any solution $c \in k^n$ with at least $n-t$ entries in $\mathbf{F}_2$ must have all entries in $\mathbf{F}_2$. I wouldn't be surprised if this is already in the literature, but I don't know a reference.

In the context of decoding, skipping the test $g^2b - a' \in ak[x]$ means that one finds $c \in k^n$ given any $e + c \in k^n$ with $\operatorname{wt} e \le t$. If one adds the requirement that the input $e+c$ is in $\mathbf{F}_2^n$ then the resulting c must have at least $n-t$ entries in $\mathbf{F}_2$, so it is in $\mathbf{F}_2^n$, so $e \in \mathbf{F}_2^n$ as desired. Theorem 7.4 spells out the precise conditions.

Theorem 7.2 (Goppa parity checks). *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 g be an element of $k[x]$ such that $\deg g = t$ and $\gcd\{g, A\} = 1$. Let c be an element of k^n. Then $\sum_i c_i A/(x-\alpha_i) \in gk[x]$ if and only if $\sum_i c_i\alpha_i^s/g(\alpha_i) = 0$ for all nonnegative integers $s < t$.*

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 3.1, 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.

By hypothesis $\deg g = t$. If $\deg B < n-t$ 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 $\deg B = \deg(C/g) < n - \deg g = n - t$.

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

$$x^tB - Q = \sum_i \frac{c_i}{g(\alpha_i)}(x^t - \alpha_i^t)\frac{A}{x-\alpha_i} = A\sum_i \frac{c_i}{g(\alpha_i)} \sum_{0 \le s < t} x^{t-1-s}\alpha_i^s.$$

One has $\deg Q < n$, so $\deg B < n-t$ if and only if $\deg(x^tB - Q) < n$, i.e., if and only if $\sum_i (c_i/g(\alpha_i)) \sum_{0\le s<t} x^{t-1-s}\alpha_i^s = 0$, i.e., if and only if $\sum_i (c_i/g(\alpha_i))\alpha_i^s = 0$ for all s with $0 \le s < t$. Hence $C \in gk[x]$ if and only if $\sum_i c_i\alpha_i^s/g(\alpha_i) = 0$ for all s with $0 \le s < t$. □

If the formal structure of this paper allowed $k((x^{-1}))$ then one could replace the last paragraph of the proof with the following: $\deg B < n-t$ if and only if $\deg(B/A) < -t$, i.e., if and only if $\sum_i c_i\alpha_i^s/g(\alpha_i) = 0$ for all nonnegative integers $s < t$, since $B/A = \sum_s x^{-s-1}\sum_i c_i\alpha_i^s/g(\alpha_i)$ as in Section 5.6. The proof given above replaces B/A with the approximation $(x^tB - Q)/x^tA$ so as to work entirely with polynomials.

Theorem 7.3 (Goppa alignment). *Let n, t be nonnegative integers. Let k be a finite field with $\mathbf{F}_2 \subseteq k$. 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 $\deg g = t$ and $\gcd\{g, A\} = 1$. Let c be an element of k^n such that $\sum_i c_i A/(x - \alpha_i) \in gk[x]$. If $\#\{i : c_i \in \mathbf{F}_2\} \geq n - t$ then $c \in \mathbf{F}_2^n$.*

Proof. Write $d_i = c_i^2 - c_i$. Saying $c_i \in \mathbf{F}_2$ is the same as saying $d_i = 0$.

Write $I = \{i : d_i \neq 0\}$. By hypothesis $d_i = 0$ for at least $n - t$ values of i, i.e., $\#I \leq t$. The objective is to show that $d_i = 0$ for all i, i.e., that $I = \{\}$.

By Theorem 7.2, $\sum_i c_i \alpha_i^s / g(\alpha_i) = 0$ for $0 \leq s < t$.

Also $\sum_i c_i A/(x - \alpha_i) \in g^2 k[x]$ by Theorem 6.3. Substitute $(g^2, 2t)$ for (g, t) in Theorem 7.2 to see that $\sum_i c_i \alpha_i^s / g(\alpha_i)^2 = 0$ for $0 \leq s < 2t$.

One has $2 = 0$ in k since k contains $\mathbf{F}_2$, 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. Squaring is also injective: if $v^2 = w^2$ then $(v + w)^2 = v^2 + w^2 = 2v^2 = 0$ so $v + w = 0$ so $v = w$. Consequently $\alpha_1^2, \ldots, \alpha_n^2$ are distinct.

For each s with $0 \leq s < t$, square the equation $\sum_i c_i \alpha_i^s / g(\alpha_i) = 0$ to obtain $\sum_i c_i^2 \alpha_i^{2s} / g(\alpha_i)^2 = 0$. Also $\sum_i c_i \alpha_i^{2s} / g(\alpha_i)^2 = 0$ since $0 \leq 2s < 2t$. Subtract to obtain $\sum_i d_i \alpha_i^{2s} / g(\alpha_i)^2 = 0$, i.e., $\sum_{i \in I} d_i \alpha_i^{2s} / g(\alpha_i)^2 = 0$.

The first $\#I$ of these equations, the equations with $s < \#I$, now imply $d_i / g(\alpha_i)^2 = 0$ for each $i \in I$ by transposed Vandermonde invertibility, since the quantities α_i^2 for $i \in I$ are distinct. Hence $d_i = 0$ for each $i \in I$, but $d_i \neq 0$ for $i \in I$, so $\#I = 0$ as claimed. □

A slightly different proof takes I to be any set of size at most t containing all i for which $d_i \neq 0$. This still reaches the conclusion that $d_i = 0$ for each $i \in I$. One also has $d_i = 0$ for each $i \notin I$ by definition of I, so $d_i = 0$ for all i as claimed.

Theorem 7.4 (checking Goppa decoding for received words in $\mathbf{F}_2^n$). *Let n, t be nonnegative integers. Let k be a finite field with $\mathbf{F}_2 \subseteq k$. 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 $\deg g = t$ 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 $g(\alpha_i)^2 B(\alpha_i)/A'(\alpha_i) \in \mathbf{F}_2$ for all i. Define $e \in \mathbf{F}_2^n$ by $e_i = [a(\alpha_i) = 0]$. Then $\operatorname{wt} e = \deg a$ and*

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

where A' is the derivative of A.

Compared to Theorem 6.5, this adds the condition that $g(\alpha_i)^2 B(\alpha_i)/A'(\alpha_i) \in \mathbf{F}_2$, but removes the condition that $g^2 b - a' \in ak[x]$.

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

Define $c \in k^n$ by $c_i = (g^2 f)(\alpha_i)/A'(\alpha_i)$. Then

$$\sum_i c_i \frac{A}{x-\alpha_i} = \sum_i (g^2 f)(\alpha_i) \prod_{j \neq i} \frac{x-\alpha_i}{\alpha_j - \alpha_i} = g^2 f \in g^2 k[x]$$

by Theorem 3.1.

Define $r_i = (g^2 B)(\alpha_i)/A'(\alpha_i)$. If $a(\alpha_i) \neq 0$ then $(A/a)(\alpha_i) = 0$ so $B(\alpha_i) = f(\alpha_i)$ so $r_i = c_i$. There are at most $\deg a \leq t$ indices i for which $a(\alpha_i) = 0$, so $\#\{i : r_i = c_i\} \geq n - t$. By hypothesis $r_i \in \mathbf{F}_2$, so $\#\{i : c_i \in \mathbf{F}_2\} \geq n - t$. Hence $c \in \mathbf{F}_2^n$ by Theorem 7.3.

Now the difference $r_i - c_i$ is in $\mathbf{F}_2$ for each i. If $e_i = 0$ then $a(\alpha_i) \neq 0$ so again $r_i = c_i$. If $e_i = 1$ then $a(\alpha_i) = 0$, so $r_i - c_i = (g^2 B - g^2 f)(\alpha_i)/A'(\alpha_i) = (g^2 b)(\alpha_i)/a'(\alpha_i) \neq 0$ since $\gcd\{a, b\} = 1$, so $r_i - c_i = 1$. In all cases $r_i - c_i = e_i$. Finally $\sum_i (r_i - e_i)A/(x-\alpha_i) = \sum_i c_i A/(x-\alpha_i) \in g^2 k[x]$, and (as in Theorem 6.4) $\operatorname{wt} e = \#\{i : a(\alpha_i) = 0\} = \deg a$ since $A \in ak[x]$. □

8 McEliece decryption

The reader is presumed to be interested specifically in Classic McEliece [12], although without much work one can also cover other versions of the McEliece cryptosystem.

8.1. Ciphertexts. In this cryptosystem, a secret vector $e \in \mathbf{F}_2^n$ with $\operatorname{wt} e = t$ is encoded as a shorter ciphertext $H(e) \in \mathbf{F}_2^{mt}$. This function $H : \mathbf{F}_2^n \to \mathbf{F}_2^{mt}$ has three critical properties:

- **Linear:** The function is $\mathbf{F}_2$-linear. This allows the function to be concisely communicated as a matrix, the public key.
- **Goppa:** Each $c \in \mathbf{F}_2^n$ 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 $\alpha_1, \dots, \alpha_n, g$ are as in Section 6, as usual with $A = \prod_i (x - \alpha_i)$.
- **Systematic:** The composition $H \circ \iota : \mathbf{F}_2^{mt} \to \mathbf{F}_2^{mt}$ is the identity map, where ι is the injection $\mathbf{F}_2^{mt} \to \mathbf{F}_2^n$ 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 $k, \alpha_1, \dots, \alpha_n, g$ there is at most one H satisfying these properties. One can construct this H, if it exists, by converting $\sum_i c_i A/(x - \alpha_i) \in gk[x]$ into a system of $\mathbf{F}_2$-linear equations (a "parity-check matrix") using Theorem 7.2, and then row-reducing 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 $(\alpha_1, \dots, \alpha_n, g)$; Chou's "semi-systematic form" options (see [13]) instead apply a limited permutation to $(\alpha_1, \dots, \alpha_n)$; [3] had instead applied an arbitrary permutation to $(\alpha_1, \dots, \alpha_n)$. See [13] for step-by-step algorithms.

8.2. Decryption. Decryption of a ciphertext $H(e)$ works as follows. Define $c = \iota(H(e)) - e \in \mathbf{F}_2^n$. 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 $e, c \in \mathbf{F}_2^n$ from $e + c$ as explained in Section 6. This recovery uses $\alpha_1, \ldots, \alpha_n, g$, which are secrets known to the party that generated the public key.

8.3. Rigidity. The cryptosystem includes defenses against chosen-ciphertext attacks. These defenses require, among other things, recognizing invalid input vectors. An input vector $\sigma \in \mathbf{F}_2^{mt}$ is by definition valid exactly when it is in $\{H(e) : e \in \mathbf{F}_2^n, \operatorname{wt} e = t\}$.

One way to handle this is as follows:

- Feed σ through any decoding algorithm that works for valid inputs. More precisely, apply some function $D : \mathbf{F}_2^{mt} \to \mathbf{F}_2^n$ with the following property: all $e \in \mathbf{F}_2^n$ with $\operatorname{wt} e = t$ have $D(H(e)) = e$.
- In all cases, whatever the output $e \in \mathbf{F}_2^n$ is, check that $\operatorname{wt} e = t$. 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., [63] 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 [12, Section 2.5] and used in the software accompanying [12], 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 6.4 to decode valid inputs but also Theorem 6.5 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 $\alpha_1, \ldots, \alpha_n$); and

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

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

It is not clear that the condition $bg^2 - a' \in ak[x]$ 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 [14].

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 $bg^2 - a' \in ak[x]$. This relies on Theorem 7.4 and the fact that $\iota(\sigma) \in \mathbf{F}_2^n$.

8.4. Robust system design. There are several reasons to recommend the second approach, 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 leading to Theorem 7.4, or in the handling of invalid inputs in the software implementing a decoding algorithm? 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 [31] breaking both specified versions (namely [3] and [4]) 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 [31] 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 correponds to flipping one bit of e.

As further illustrations of how the decoding details matter, [31] 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 [56] 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 $\mathrm{wt}\, e = t$. Niederreiter [58] introduced the idea of sending just $H(e)$ as a ciphertext, with e as the message. In both [56] and [58], 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 $\{c \in \mathbf{F}_2^n : \sum_i c_i A/(x - \alpha_i) \in gk[x]\}$. 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 [25, 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 [26, 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 [61, 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 [12]. 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 [14, Section 6], which in turn says that the choice of $\iota(\sigma)$ as a decoder input was recommended to the authors by Sendrier.

References

[1] — (no editor), *39th annual symposium on foundations of computer science, FOCS '98, November 8–11, 1998, Palo Alto, California, USA*, IEEE Computer Society, 1998. See [45].

[2] — (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 [20].

[3] Martin Albrecht, Carlos Cid, Kenneth G. Paterson, CJ Tjhai, Martin Tomlinson, *NTS-KEM* (2017); see also newer version [**4**]. URL: https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions. Citations in this document: §8.1, §8.4, §A.5, §A.7, §A.7, §A.7.

[4] Martin Albrecht, Carlos Cid, Kenneth G. Paterson, CJ Tjhai, Martin Tomlinson, *NTS-KEM: second round submission* (2019); see also older version [**3**]. URL: https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions. Citations in this document: §8.4, §A.5, §A.7, §A.7, §A.7.

[5] Elwyn R. Berlekamp, *Algebraic coding theory*, McGraw-Hill, 1968. Citations in this document: §5.9.

[6] Daniel J. Bernstein, *Fast multiplication and its applications*, in [**24**] (2008), 325–384. URL: https://cr.yp.to/papers.html#multapps. Citations in this document: §3.4, §4.5, §5.9.

[7] Daniel J. Bernstein, *Reducing lattice bases to find small-height values of univariate polynomials*, in [**24**] (2008), 421–446. URL: https://cr.yp.to/papers.html#smallheight. Citations in this document: §4.8.

[8] Daniel J. Bernstein, *List decoding for binary Goppa codes*, in IWCC 2011 [**27**] (2011), 62–80. URL: https://cr.yp.to/papers.html#goppalist. Citations in this document: §4.8.

[9] Daniel J. Bernstein, *Simplified high-speed high-distance list decoding for alternant codes*, in PQCrypto 2011 [**81**] (2011), 200–216. URL: https://cr.yp.to/papers.html#simplelist. Citations in this document: §4.8.

[10] 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: §3.4.

[11] Daniel J. Bernstein, Johannes Buchmann, Erik Dahmen (editors), *Post-quantum cryptography*, Springer, 2009. ISBN 978-3-540-88701-0. See [61].

[12] 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 [**13**]. URL: https://classic.mceliece.org/nist.html. Citations in this document: §8, §8.3, §8.3, §8.5.

[13] 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 [**12**]. URL: https://classic.mceliece.org/nist.html. Citations in this document: §8.1, §8.1.

[14] Daniel J. Bernstein, Tung Chou, Peter Schwabe, *McBits: fast constant-time code-based cryptography*, in [**17**] (2013), 250–272. URL: https://cr.yp.to/papers.html#mcbits. Citations in this document: §1.1, §8.3, §8.5.

[15] Daniel J. Bernstein, Tanja Lange, Christiane Peters, *Wild McEliece*, in SAC 2010 [**19**] (2011), 143–158. URL: https://eprint.iacr.org/2010/410. Citations in this document: §1.1.

[16] 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. Citations in this document: §4.5, §4.5, §5.9.

[17] 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. See [14].

[18] Vijay K. Bhargava, H. Vincent Poor, Vahid Tarokh, Seokho Yoon (editors), *Communications, information and network security. With a foreword by Richard E. Blahut* (2003). ISBN 978-1-4020-7251-2; 978-1-4419-5318-6; 978-1-4757-3789-9. See [40].

[19] 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. See [15].

[20] Dan Boneh, *Finding smooth integers in short intervals using CRT decoding*, in STOC 2000 [**2**] (2000), 265–272; see also newer version [**21**].

[21] 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 [**20**]. ISSN 0022-0000. URL: https://crypto.stanford.edu/~dabo/abstracts/CRTdecode.html. Citations in this document: §4.8.

[22] Claude Brezinski, *The long history of continued fractions and Padé approximants*, in [**23**] (1981), 1–27. Citations in this document: §4.8.

[23] 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* (1981). ISSN 0075-8434. See [22].

[24] Joe P. Buhler, Peter Stevenhagen (editors), *Surveys in algorithmic number theory*, Mathematical Sciences Research Institute Publications, 44, Cambridge University Press, New York, 2008. See [6], [7].

[25] 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.

[26] Anne Canteaut, Nicolas Sendrier, *Cryptanalysis of the original McEliece cryptosystem*, in Asiacrypt '98 [**59**] (1998), 187–199. URL: https://www.rocq.inria.fr/secret/Anne.Canteaut/Publications/Canteaut_Sendrier98.pdf. Citations in this document: §8.5.

[27] 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. See [8].

[28] 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. Citations in this document: §1.1.

[29] Tung Chou, *McBits revisited*, in CHES 2017 [**37**] (2017), 213–231; see also newer version [**30**]. URL: https://tungchou.github.io/papers/mcbits_revisited.pdf.

[30] Tung Chou, *McBits revisited: toward a fast constant-time code-based KEM*, Journal of Cryptographic Engineering **8** (2018), 95–107; see also older version

[30]. URL: https://doi.org/10.1007/s13389-018-0186-9. Citations in this document: §1.1.

[31] Tung Chou, *An IND-CCA2 attack against the 1st- and 2nd-round versions of NTS-KEM*, in SecITC 2020 [54] (2020), 165–184. URL: https://tungchou.github.io/papers/ntskem_cca2.pdf. Citations in this document: §8.4, §8.4, §8.4, §A.5, §A.7.

[32] 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. Citations in this document: §4.8.

[33] 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: §A.5.

[34] Jean Louis Dornstetter, *On the equivalence between Berlekamp's and Euclid's algorithms*, IEEE Transactions on Information Theory **33** (1987), 428–431. Citations in this document: §5.9.

[35] 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: §A.

[36] Euclid, *Elements*, about 300 B.C. URL: https://www.claymath.org/library/historical/euclid/files/elem.7.2.html. Citations in this document: §4.8.

[37] 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. See [29].

[38] 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: §5.9.

[39] G. David Forney, Jr., *On decoding BCH codes*, IEEE Transactions on Information Theory **11** (1965), 549–557. Citations in this document: §5.9.

[40] Shuhong Gao, *A new algorithm for decoding Reed-Solomon codes*, in [18] (2003), 55–68. URL: https://www.math.clemson.edu/~sgao/papers/RS.pdf. Citations in this document: §5.9.

[41] Carl Friedrich Gauss, *Disquisitiones arithmeticae*, 1801. URL: https://archive.org/details/disquisitionesa00gaus. Citations in this document: §4.8.

[42] 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. Citations in this document: §1.1.

[43] 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, §6.1, §7.1.

[44] 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. Citations in this document: §5.9.

[45] Venkatesan Guruswami, Madhu Sudan, *Improved decoding of Reed-Solomon and algebraic-geometry codes*, in FOCS 1998 [1] (1998), 28–39; see also newer version [46]. URL: https://madhu.seas.harvard.edu/papers.html. Citations in this document: §1.1.

[46] 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 [46]. ISSN 0018-9448. URL: https://madhu.seas.harvard.edu/papers.html.

[47] 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 [67].

[48] Jørn Justesen, *On the complexity of decoding Reed–Solomon codes*, IEEE Transactions on Information Theory **22** (1976), 237–238. Citations in this document: §5.9.

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

[50] 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: §4.8.

[51] 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: §4.8.

[52] 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: §4.8.

[53] 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: §4.8.

[54] 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. See [31].

[55] James Massey, *Shift-register synthesis and BCH decoding*, IEEE Transactions on Information Theory **15** (1969), 122–127. ISSN 0018-9448. Citations in this document: §5.9.

[56] 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.

[57] 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/. Citations in this document: §5.9.

[58] 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.

[59] 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. See [26].

[60] 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: §A.5.

[61] Raphael Overbeck, Nicolas Sendrier, *Code-based cryptography*, in [11] (2009), 95–145. Citations in this document: §1.1, §8.5.

[62] 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. Citations in this document: §4.8.

[63] 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.

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

[65] W. Wesley Peterson, *Encoding and error-correction procedures for the Bose-Chaudhuri codes*, Transactions of the Institute of Radio Engineers **6** (1960), 459–470. Citations in this document: §5.9.

[66] Eugene Prange, *The use of information sets in decoding cyclic codes*, IRE Transactions on Information Theory **IT-8** (1962), S5–S9. Citations in this document: §5.9.

[67] Bart Preneel, Antoon Bosselaers, René Govaerts, Joos Vandewalle, *A software implementation of the McEliece public-key cryptosystem*, in [47] (1992), 119–126. URL: https://www.esat.kuleuven.be/cosic/publications/article-267.pdf. Citations in this document: §1.1.

[68] 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. Citations in this document: §5.9.

[69] 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. Citations in this document: §5.9.

[70] Akira Shiozaki, *Decoding of redundant residue polynomial codes using Euclid's algorithm*, IEEE Transactions on Information Theory **34** (1989), 1351–1354. Citations in this document: §5.9.

[71] 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: §4.8.

[72] Volker Strassen, *Gaussian elimination is not optimal*, Numerische Mathematik **13** (1969), 354–356. ISSN 0029-599X. Citations in this document: §5.9.

[73] Dirk J. Struik, *The origin of L'Hôpital's rule*, The Mathematics Teacher **56** (1963), 257–260. URL: https://www.jstor.org/stable/27956806. Citations in this document: §2.22.

[74] 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. Citations in this document: §1.1.

[75] 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. Citations in this document: §5.9.

[76] 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.

[77] 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.

[78] 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`. Citations in this document: §3.

[79] Lloyd R. Welch, Robert A. Scholtz, *Continued fractions and Berlekamp's algorithm*, IEEE Transactions on Information Theory **25** (1979), 19–27. Citations in this document: §5.9.

[80] 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: §A.5.

[81] 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. URL: `https://doi.org/10.1007/978-3-642-25405-5`. See [9].

A Random tests

Beware of bugs in the above code; I have only proved it correct, not tried it. —Knuth [**35**, page 11 in cited PDF]

Figures A.1, A.2, A.3, and A.4 are Sage scripts to test Algorithms 3.3, 4.4, 5.3, and 6.2 respectively on random inputs.

A.5. 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?

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 [**31**] in the Goppa decoder from [**3**] and [**4**]. As another example, Ormandy [**60**] discovered that some inputs would cause an extended-gcd algorithm in a Microsoft cryptography library to enter an infinite loop;

```
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))
```

Fig. A.1. Random tests for Algorithm 3.3.

this meant that an attacker could trivially cause a server to stop responding, something that [80] 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 [33] 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.

A.6. 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 A.3 are tests aimed at checking correct behavior on decodable inputs. These tests use input vectors r

```
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
```

Fig. A.2. Random tests for Algorithm 4.4.

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 $\mathrm{wt}(r - c) \le t$ 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 A.4 has an analogous split between testing decodable inputs and testing non-decodable inputs for Goppa decoding. There is no similar split in Figures A.1 and A.2, since those algorithms handle all inputs successfully.

For Figures A.1, A.3, and A.4, n is chosen randomly between 0 and q; for Figure A.2, $\deg A$ is chosen randomly between 0 and 99. Similarly, t is chosen randomly in Figures A.2, A.3, and A.4; in each case, the range of t covered by the tests is slightly beyond the range of t useful for applications.

A.7. How the tests catch various bugs. The bug in the Goppa decoder from [3] and [4] is triggered when the correct error vector e has weight $t-1$ and has $e_z = 0$ where $\alpha_z = 0$. Figure A.4 is intended to catch this: the tests generate uniform random sequences $(\alpha_1, \ldots, \alpha_n)$ of distinct field elements, and often use

```
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
```

Fig. A.3. Random tests for Algorithm 5.3.

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.

I tried modifying Algorithm 6.2 to imitate what [31] described; Figure A.4 immediately caught the bug. One could directly test the algorithms from [3] and [4] by translating the algorithms from pseudocode to real code. One could directly test the software accompanying [3] and [4] by extracting the Goppa-decoding portions of that software and providing a shim layer to support the `goppa_errors` interface.

The extended-gcd bug 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 input, triggering an infinite loop. In the decoding context, an extended-

```
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(t)]+[1])
      if g.is_squarefree():
        if all(g(aj) != 0 for aj in a):
          break

    assert g.degree() == 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^2)(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)))
```

Fig. A.4. Random tests for Algorithm 6.2.

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 $\deg a = t$. I checked that eight runs of Figure A.3 consistently caught this bug; each run already caught the bug with $\#k = 2$. I also checked that eight runs of Figure A.4 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.