Normal basis exhaustive search: 10 years later

L. Moura[1], D. Panario[2], and D. Thomson[2]

[1] University of Ottawa lucia@site.uottawa.ca
[2] Carleton University {daniel,dthomson}@math.carleton.ca

Abstract. This paper concerns an exhaustive search for low complexity normal bases over finite fields $\mathbb{F}_{2^n}$ over $\mathbb{F}_2$ for $n \leq 46$. This is a followup paper to [11], which appeared one decade ago in 2008 and completed the cases $n \leq 39$. We extend the results in [11] by taking advantage of a combination of algorithmic improvements, more efficient implementations and massive parallelism.

Keywords: finite fields, normal bases, NTL, parallel computing

1 Introduction

For any finite field extension $\mathbb{F}_{2^n}$ over $\mathbb{F}_2$, upon choice of a basis, we can write any field element as an n-long binary vector; that is, $\mathbb{F}_{2^n} \cong \mathbb{F}_2^n$, and the basis makes the isomorphism explicit. When performing finite field arithmetic the choice of representation of elements is critical to the performance of various operations in the system.

For tiny fields, look-up tables can be employed for field arithmetic and for mid-sized fields *Zech logarithms* as a time-memory tradeoff on full lookup tables can be used, see [13, Section 2.1.7.5] for more details. For larger finite fields where lookup tables are no longer feasible, we exploit an explicit isomorphism between the field $\mathbb{F}_{2^n}$ and the vector space $\mathbb{F}_2^n$ by choice of a given basis. Most commonly, field extensions are given by *polynomial* or *power bases*; namely, $\mathbb{F}_{2^n} \cong \mathbb{F}_2[x]/(f)$, where $f \in \mathbb{F}_2[x]$ is an irreducible polynomial of degree n. Here, arithmetic is performed (mod f), and is reasonably efficient in many cases, especially when a sparse choice of f is used. This paper deals with a different kind of basis, namely, *normal bases*.

Definition 1. *Let $\mathcal{B} = \{\alpha, \alpha^2, \ldots, \alpha^{2^{n-1}}\}$. If $\mathcal{B}$ is a basis of $\mathbb{F}_{2^n}$ over $\mathbb{F}_2$ (i.e., if its elements are linearly independent), then $\mathcal{B}$ is a* normal basis *of $\mathbb{F}_{2^n}$ over $\mathbb{F}_2$ and every $\beta \in \mathcal{B}$ is a* normal element *of $\mathbb{F}_{2^n}$ over $\mathbb{F}_2$.*

For any $\alpha \in \mathbb{F}_{2^n}$, the elements α^{2^i}, $0 \leq i \leq n-1$, are the (Galois) *conjugates* of α.

We highlight the usefulness of a normal basis representation of $\mathbb{F}_{2^n}$ over $\mathbb{F}_2$. Suppose $\gamma = \sum_{i=0}^{n-1} g_i \alpha^{2^i}$, then $\gamma^{2^j} = \sum_{i=0}^{n-1} g_{(i-j) \bmod n} \alpha^{2^i}$ for any integer j. Hence, squaring in normal basis representation is given by a cyclic bit shift of its underlying coefficient vector.

The multiplication of elements represented in normal basis receives a similar simplification. Denote by t_{ijk} the *structure constants* of $\mathcal{B}$, given by the relations

$$\alpha^{2^i} \alpha^{2^j} = \sum_{k=0}^{n-1} t_{ijk} \alpha^{2^k}.$$

Since $\alpha^{2^i} \alpha^{2^j} = (\alpha \alpha^{2^{j-i}})^{2^i}$, we have $t_{ijk} = t_{0(j-i)(k-i)}$ for all i, j, k, where subscripts are taken modulo n. The complexity of multiplication of generic elements in $\mathbb{F}_{2^n}$ depends directly on the number of non-zero structure constants. By the above, therefore, the cost of multiplication is directly related to the number of nonzero t_{0jk}.

Definition 2. *Let* $\mathcal{B} = \{\alpha, \alpha^2, \ldots, \alpha^{2^{n-1}}\}$ *be a normal basis of* $\mathbb{F}_{2^n}$ *over* $\mathbb{F}_2$ *and let constants* t_{ij} *be given by the relations*

$$\alpha\alpha^{2^i} = \sum_{j=0}^{n-1} t_{ij}\alpha^{2^j} \quad 0 \le i \le n-1.$$

The complexity *(or* density*) of* $\mathcal{B}$ *is given by* $\mathcal{C}_\mathcal{B} = |\{t_{ij} \neq 0 \colon 0 \le i, j \le n-1\}|$. *The matrix* $T_\mathcal{B} = (t_{ij})$ *is the* multiplication table *of* $\mathcal{B}$.

By *low complexity normal bases*, we mean normal bases whose complexity is bounded by kn for a small integer constant k. Low complexity normal bases are desirable for efficient computations, particularly when the application requires a large number of squarings or exponentiations. For example, the National Institute for Standards in Technology prescribes low complexity normal bases for use in elliptic curve cryptography for curves over $\mathbb{F}_{2^n}$ [14], and they are also prescribed for use in decoding of Gabidulin codes in the rank metric [17]. The following proposition provides an achievable lower-bound on the complexity of normal bases.

Proposition 1. [12] *Let* $\mathcal{B}$ *be a normal basis, then* $2n-1 \le \mathcal{C}_\mathcal{B} \le n^2 - n$.

Bases which meet the lower bound from Proposition 1 are *optimal normal bases*, and were characterized completely in [4]. Optimal normal bases exist only when $n+1$ or $2n+1$ are prime, and generalizations given by *Gauss periods* provide low complexity normal bases only when $kn+1$ is prime for some k; see [13, Section 5.3]. Few other constructions of normal bases are known, nearly all of which construct normal bases in $\mathbb{F}_{2^n}$ from existing normal bases in either subfields or extensions. Hence, low-complexity normal bases are even required in these cases as a starting point. Heuristically, the complexity of a random normal basis is on the order of $n^2/2$, is tightly compacted about the mean, and up to half of all elements of $\mathbb{F}_{2^n}$ are normal, so random search is unlikely to yield low complexity normal bases; see [11].

Exhaustive searches for normal bases of $\mathbb{F}_{2^n}$ over $\mathbb{F}_2$ have been performed previously, for $n < 30$ in [12], for $n < 33$ in [8], and for $n < 40$ by the authors (with A. Masuda) in [11]. Restricting to a subset of normal bases, namely *self-dual* normal bases, allows for efficient search by the transitive action of orthogonal circulant matrices; this was completed for $n \le 47$ in [8] and those results were verified and extended to odd characteristics in [1].

The purpose of this paper is to extend the results in [11] using a combination of algorithmic improvements, efficient implementations and increased availability of computational resources. In [11] we found the minimum complexity normal bases in $\mathbb{F}_{2^n}$ for all $n = 2, \ldots, 39$ using an implementation of Algorithm 1; see Section 3 for details. In this paper, we give an updated algorithm, Algorithm 2, see Section 4, and give a more efficient implementation, see Section 5. Using these improvements, we extend our search to include the cases $n = 40, \ldots, 46$. We give these results and some candidates for future work in Section 6.

2 Some necessary background and notation

In this paper, we use field elements $\alpha \in \mathbb{F}_{2^n}$ interchangeably with their expansion as n-long binary vectors in some implicit underlying basis. When we write matrices as a single row of elements in $\mathbb{F}_{2^n}$ we mean their implicit expansion in the underlying $\mathbb{F}_2$-basis; for example, $P_\alpha = \left(\alpha\ \alpha^2\ \cdots\ \alpha^{2^{n-1}}\right)$ should be interpreted as an $n \times n$ matrix with entries in $\mathbb{F}_2$ where the ith column is the expansion of α^{2^i}, $0 \le i < n$, in the implicit underlying basis.

Operation	Denoted	Cost
Addition in $\mathbb{F}_{2^n}$	$A(n)$	n
Multiplication (classical)	$M(n)$	$4n^2 - 5n + 1$
Matrix multiplication (classical) $C = AB$, $C, A, B \in \mathbb{F}_2^{n \times n}$	$\mathfrak{M}(n)$	$2n^3 - n^2$
Matrix inversion (Gaussian reduction) in $\mathbb{F}_2^{n \times n}$	$I(n)$	$\frac{2}{3}(n^3 + 3n^2 - 4n)$
$\gcd(f, x^n - 1), f \in \mathbb{F}_{2^n}[x]$ (fast methods)	$G(n)$	$\geq nM(n) \log n$

Table 1. Upper bound on the cost of arithmetic operations in $\mathbb{F}_{2^n}$, given in bit operations [6].

Throughout this paper, we denote by e_i the ith standard basis vector for $0 \leq i \leq n-1$; that is $e_i = (e_{ij})$ with

$$e_{ij} = \begin{cases} 1 & j = i, \\ 0 & \text{otherwise.} \end{cases}$$

2.1 Complexity of field arithmetic

In Table 1, we give the computational complexity of the $\mathbb{F}_{2^n}$ arithmetic operations we use in this paper, given in number of $\mathbb{F}_2$ operations.

The quantity $M(n)$ gives the number of $\mathbb{F}_2$-operations for computing the multiplication of two elements of $\mathbb{F}_{2^n}$ over $\mathbb{F}_2$. Classically, this is done by polynomial convolution plus the division by an irreducible modulus of degree n with remainder. The `NTL` documentation states that $\mathbb{F}_{2^n}$ arithmetic is performed "using a combination of classical routines and Karatsuba." From our understanding of the `NTL` internals, for $n < 64$, multiplication is performed by special 128-bit register arithmetic, when available, otherwise classically in an unrolled loop. For moderate sized $n < 512$, explicit unrolled Karatsuba multiplication is performed. Since this is outside of our range of interest, we do not go into the details of Karatsuba multiplication costing here. Asymptotically faster methods are also available, but these are far outside of our range of interest. After multiplication, division plus remainder is performed using special functions for a trinomial or pentanomial modulus, as applicable. We use the `NTL` default modulus for all n.

The `NTL` implementation of matrix inversion uses Gaussian reduction, so we give this cost with the observation that over $\mathbb{F}_2$ there is no row scaling required. Though asymptotically faster methods exist here too, the discussion in the preamble to [6, Chapter 12] indicates that these methods have crossover points also outside of our range of interest.

The cost $G(n)$ of a polynomial gcd in $\mathbb{F}_{2^n}[x]$ is only used as a lower bound to show that the computational complexity of our improved algorithm, Algorithm 2, is less than the original Algorithm 1. For this reason, we state the cost of $G(n)$ using fast arithmetic, regardless of crossover point, since this is enough and the use of classical arithmetic would only increase the savings of Algorithm 2.

3 Original algorithm from [11]

In this section we present Algorithm 1 and explain its various components. Algorithm 1 first appeared in [11], and is the main reference from which this work is based on. In Section 4 we analyze Algorithm 2, our updated algorithm from this work. We have a new implementation of Algorithm 2, so we leave a discussion of specific implementation details to Section 5.2.

3.1 Efficient iteration through $\mathbb{F}_{2^n}$

This section is devoted to justifying how explaining how lines 7 and 8 of Algorithm 1 admit efficient iteration through putative normal bases in $\mathbb{F}_{2^n}$.

Algorithm 1 Exhaustive search algorithm from [11]

1: **Input:** $n \in \mathbb{Z}_{>0}$
2: **Returns:** $\alpha \in \mathbb{F}_{2^n}$, a normal element with minimum complexity
3:
4: min_complexity $\leftarrow \infty$; min_element $\leftarrow 0$; $\alpha \leftarrow 0$; $\mathcal{B} = \{0, \ldots, 0\}$
5: Precompute $\{e_i^{2^j}\}, 0 \leq i, j \leq n-1$
6: **for** $idx \leftarrow 0$ **to** $2^n - 1$ **do**
7: $\quad \gamma \leftarrow \Gamma(idx)$ ▷ See Section 3.1
8: $\quad \mathcal{B} \leftarrow \{\alpha + e_\gamma, \alpha^2 + e_\gamma^2, \ldots, \alpha^{2^{n-1}} + e_\gamma^{2^{n-1}}\}$
9: $\quad$ **if** $\alpha \neq \min_<(\mathcal{B})$ **then** ▷ See Section 3.2
10: $\quad\quad$ **continue**
11: $\quad$ **if** $\gcd(\sum_{i=0}^{n-1} \alpha^{2^i} x^{n-1-i}, x^n - 1) \neq 1$ **then** ▷ See Theorem 1
12: $\quad\quad$ **continue**
13: $\quad$ cplex $\leftarrow \mathcal{C}_\mathcal{B}$ ▷ See Algorithm 3
14: $\quad$ **if** cplex $<$ min_complexity **then**
15: $\quad\quad$ min_complexity $\leftarrow$ cplex; min_element $\leftarrow \alpha$
16: **return** min_complexity, min_element

For any element $\alpha \in \mathbb{F}_{2^n}$, calculating $\mathcal{B} = \{\alpha, \alpha^2, \ldots, \alpha^{2^{n-1}}\}$ generically requires $n-1$ multiplications in $\mathbb{F}_{2^n}$ using repeated squaring. Suppose $\mathcal{B}$ is known, and consider computing the putative basis $\mathcal{B}' = \{\alpha + \beta, \ldots, \alpha^{2^{n-1}} + \beta^{2^{n-1}}\}$. We show how to efficiently iterate through all of $\mathbb{F}_{2^n}$ after the precomputation of a small set of elements in $\mathbb{F}_{2^n}$.

Definition 3. *A* Gray code *is an ordering* $\{g_0, g_1, \ldots, g_{2^n-1}\}$ *of* $\mathbb{F}_2^n$ *such that* $H(g_i, g_{i+1}) = 1$ *for* $i = 0, 1, \ldots, 2^n - 2$*, where* $H(a, b)$ *is the Hamming distance of* $a, b \in \mathbb{F}_2^n$*.*

We use a Gray code for efficient iteration through $\mathbb{F}_2^n$ as follows. Any element $\alpha \in \mathbb{F}_{2^n}$ is isomorphic to its binary vector of coefficients in a fixed basis; say $\alpha \cong g_i$ for some g_i in a Gray code. The *successor* of α is the element $\beta \cong g_{i+1}$. For ease of notation we identify $\alpha, \beta \in \mathbb{F}_{2^n}$ with their coefficient vectors. In a Gray code, $\beta = \alpha + e_\gamma$ for some standard basis vector e_γ, $0 \leq \gamma \leq n-1$. Then $\beta^{2^j} = \alpha^{2^j} + e_\gamma^{2^j}$, by the linearity of Frobenius. Hence, with a precomputation of the vectors $e_i^{2^j}$ for $0 \leq i, j \leq n-1$, any β^{2^j} can be computed by a binary vector addition.

Given a Gray code $\{g_0, \ldots, g_{2^n-1}\}$ define $\Gamma\colon [0, 2^n - 1] \to [0, n-1]$, where $\Gamma(i) = \gamma$ whenever $g_{i+1} = g_i + e_\gamma$. The function Γ is used to compute a Gray code successor in line 7-8 in Algorithm 1; a specific instantiation of Γ is given in Proposition 5.

3.2 Reducing search space using a canonical element check

In this section, we present Proposition 2 to justify line 9, which reduces the size of the search space in Algorithm 1 by a factor of n.

Proposition 2. *Let* $\mathcal{B}_\alpha = \{\alpha, \alpha^q, \ldots, \alpha^{q^{n-1}}\}$ *be a putative normal basis. For any* $\beta \in \mathcal{B}_\alpha$*, as an unordered multiset* $\mathcal{B}_\beta = \{\beta, \beta^q, \ldots, \beta^{q^{n-1}}\} = \mathcal{B}_\alpha$ *and hence* $\mathcal{B}_\beta$ *is a basis if and only if* $\mathcal{B}_\alpha$ *is. If both* $\mathcal{B}_\alpha$ *and* $\mathcal{B}_\beta$ *are bases, then* $\mathcal{C}_{\mathcal{B}_\beta} = \mathcal{C}_{\mathcal{B}_\alpha}$*.*

By Proposition 2, we need to check normality and compute complexity for only a canonical basis representative among its conjugates. The elements of $\mathbb{F}_{2^n}$ (and hence, the elements of $\mathcal{B}_\alpha$) can be represented as n-long binary vectors under some implicit basis, so for any $\alpha, \beta \in \mathbb{F}_{2^n}$ we define the ordering $\leq$ by $\alpha < \beta$ if α has lexicographically smaller binary vector representation than β. Equality is well-defined under this ordering: α and β have equal order if and only if $\alpha = \beta$. Equality occurs in a putative basis $\mathcal{B}$ if and only if $\beta = \alpha^{2^i}$ for some i, $1 \leq i \leq n-1$,

proving that $\mathcal{B}$ is not a basis. We denote by $\min_<(\mathcal{B})$ the unique minimal element of $\mathcal{B}$ under $\leq$, if it exists.

For implementation notes on computing lexicographically small elements, see Section 5.2.

3.3 Checking normality

Line 11 of Algorithm 1 is used to filter non-normal elements of $\mathbb{F}_{2^n}$, based on Theorem 1.

Theorem 1. [13, Theorem 5.2.11] *Let $\alpha \in \mathbb{F}_{2^n}$ and let $g_\alpha(x) = \alpha x^{n-1} + \alpha^2 x^{n-2} + \cdots + \alpha^{2^{n-2}} x + \alpha^{2^{n-1}}$. Then α is normal over $\mathbb{F}_2$ if and only if $\gcd(g_\alpha, x^n - 1) = 1$.*

We combine all the ingredients in this section to justify the correctness of Algorithm 1.

Theorem 2. *Algorithm 1 is correct and computes the complexity of every normal basis of $\mathbb{F}_{2^n}$ over $\mathbb{F}_2$.*

Proof. Let $\mathcal{B} = \{\alpha, \alpha^2, \ldots, \alpha^{2^{n-1}}\} \subset \mathbb{F}_{2^n}$ be a normal basis of $\mathbb{F}_{2^n}$ over $\mathbb{F}_2$. If $\mathcal{B}$ is a basis, then $\mathcal{B}$ has a unique minimal element under $\leq$, so without loss of generality suppose $\alpha = \min_<(\mathcal{B})$.

By Proposition 5, α will be met for some index idx; hence, $\mathcal{B}$ will be computed in line 8 at index idx. By supposition, α passes line 9. Finally, $\gcd(\sum_{i=0}^{n-1} \alpha^{2^i} x^{n-1-i}, x^n - 1) = 1$ by Theorem 1, hence the complexity of $\mathcal{B}$ is computed in line 13. ■

4 An improved algorithm for exhaustive search for normal bases

In this section, we present a number of algorithmic improvements to the basic algorithm (Algorithm 1) from [11]. Our improved algorithm is given in Algorithm 2. We maintain efficient iteration through $\mathbb{F}_{2^n}$ as in Algorithm 1 and provide an additional cut down of the search space (see Section 4.1). In Section 4.2, we show how to improve the run-time by deferring the normality check due to Theorem 1.

Algorithm 2 Updated exhaustive search for low complexity normal bases

1: **Input:** $n \in \mathbb{Z}_{>0}$
2: **Returns:** $\alpha \in \mathbb{F}_{2^n}$, a normal element with minimum complexity
3:
4: min_complexity $\leftarrow \infty$; min_element $\leftarrow 0$; $\alpha \leftarrow 0$; $\mathcal{B} = \{0, \ldots, 0\}$; trace $\leftarrow 0$
5: Precompute $\{e_i^{2^j}\}, 0 \leq i, j \leq n-1$ and $\mathrm{Tr}(e_i), 0 \leq i \leq n-1$
6: **for** $idx \leftarrow 0$ **to** $2^n - 1$ **do**
7: $\quad\gamma \leftarrow \Gamma(idx)$ ▷ See Section 3.1
8: $\quad\mathcal{B} \leftarrow \{\alpha + e_\gamma, \alpha^2 + e_\gamma^2, \ldots, \alpha^{2^{n-1}} + e_\gamma^{2^{n-1}}\}$
9: $\quad$trace $\leftarrow$ trace $\oplus$ $\mathrm{Tr}(e_\gamma)$
10: $\quad$**if** trace $= 0$ **then**
11: $\quad\quad$**continue**
12: $\quad$**if** $\alpha \neq \min_<(\mathcal{B})$ **then** ▷ See Section 3.2
13: $\quad\quad$**continue**
14: $\quad$cplex $\leftarrow \mathcal{C}_\mathcal{B}$ ▷ See Algorithm 3
15: $\quad$**if** cplex $<$ min_complexity **then**
16: $\quad\quad$min_complexity $\leftarrow$ cplex; min_element $\leftarrow \alpha$
17: **return** min_complexity, min_element

4.1 Trace precomputation

We use another pre-computation and cut-down due to the following observation, based on the fact that $\mathcal{B}$ is a normal basis if and only if its elements are linearly independent.

Definition 4. *Let $\mathbb{F}_{2^n}$ be the degree n extension of $\mathbb{F}_2$. Denote the* trace *of an element $\alpha \in \mathbb{F}_{2^n}$ by $\mathrm{Tr}(\alpha) = \sum_{i=0}^{n-1} \alpha^{2^i}$.*

Recall from Theorem 1 that an element α is normal if and only if $g_\alpha(x) = \sum_{i=0}^{n-1} \alpha^{2^i} x^{n-i}$ is coprime with $x^n - 1$. Since 1 is a root of $x^n - 1$ for all n, if 1 is a root of g_α, then α is not normal. We summarize the contrapositive in the next proposition.

Proposition 3. *If α is normal, then $\mathrm{Tr}(\alpha) \neq 0$.*

The trace is an additive map $\mathbb{F}_{2^n} \to \mathbb{F}_2$, and by linearity it is easy to see that $\mathrm{Tr}(\alpha^2) = \mathrm{Tr}(\alpha)$. Thus, the value of $\mathrm{Tr}(\alpha)$ is invariant across the set α^{2^j}. Hence, with an additional precomputation of $\mathrm{Tr}(e_i)$ for $0 \leq i < n$, when computing the successor $\alpha' \leftarrow \alpha + e_j$ for some j, we also compute $\mathrm{Tr}(\alpha') = \mathrm{Tr}(\alpha) + \mathrm{Tr}(e_j)$, with $\mathrm{Tr}(\alpha)$ and $\mathrm{Tr}(e_j)$ known. If $\mathrm{Tr}(\alpha + e_j) = 0$, then α' is not normal and we continue the outer loop. This can be seen in lines 9-11 of Algorithm 2.

Proposition 4 shows that with this small trace precomputation, requiring only a storage of n bits, we receive an immediate cut-down in the search space by a factor of 2.

Proposition 4. [10, Theorem 2.25] *Let $\alpha \in \mathbb{F}_{2^n}$. We have $\mathrm{Tr}(\alpha) = 0$ if and only if $\beta^2 + \beta = \alpha$ for some $\beta \in \mathbb{F}_{2^n}$. Hence, exactly 2^{n-1} elements of $\mathbb{F}_{2^n}$ have trace 0.*

4.2 Removing explicit normality checking

A key difference between Algorithms 1 and 2 is the *removal* of line 11 from Algorithm 1, which ensures that only bases pass to the complexity calculation step given by Algorithm 3. In this section we show that, with a small change to Algorithm 3, the removal of this line clears a performance bottleneck from Algorithm 1.

We highlight the simple but important observation that $P_\alpha = \left(\alpha\ \alpha^2 \cdots\ \alpha^{2^{n-1}}\right)$ has full rank if and only if $\mathcal{B} = \{\alpha, \alpha^2, \ldots, \alpha^{2^{n-1}}\}$ is a normal basis of $\mathbb{F}_{2^n}$ over $\mathbb{F}_2$.

Algorithm 3 is used to calculate the complexity from a putative normal element. The matrix P_α^{-1} is a "change of basis" matrix from the implicit underlying basis to $\mathcal{B}$. If $\mathcal{B}$ is not invertible (hence if the element α is not normal), then P_α is non-invertible. If P_α^{-1} is computed by Gaussian reduction, as soon as a non-pivot row is discovered, an inversion routine can return $0 = \det(P_\alpha)$. Hence, checking normality and inversion can be performed simultaneously.

It is not obvious that deferring normality checking to the Gaussian reduction step is a net benefit. The remainder of this section is devoted to showing that this removal indeed is a benefit.

Theorem 3. [5] *The proportion $\psi(n)$ of normal bases of $\mathbb{F}_{2^n}$ over $\mathbb{F}_2$ satisfies*

$$\psi(n) = \frac{\Phi(x^n - 1)}{n2^n} \geq \frac{1}{ne^{0.83}(1 + \log_2(n))},$$

where Φ is Euler's phi function for polynomials over $\mathbb{F}_2$.

Since trace-0 elements are not normal, exactly $2n\psi(n)$ trace-1 elements of $\mathbb{F}_{2^n}$ are normal.

Let $G(n)$ be the cost of a gcd in $\mathbb{F}_{2^n}$, $M(n)$ be the cost of a field multiplication in $\mathbb{F}_{2^n}$ and $I(n)$ be the cost of an $n \times n$ matrix inversion over $\mathbb{F}_2$. The cost of first checking normality

on $2^{n-1}/n$ elements and then calculating complexity on a lower-bound of the remaining $2\psi(n)$ elements is

$$\frac{2^{n-1}}{n}\left(G(n) + 2\psi(n)\left((n-1)M(n) + I(n)\right)\right). \tag{1}$$

On the other hand, using Gaussian reduction as a normality check takes time

$$\frac{2^{n-1}}{n}\left((n-1)M(n) + I(n)\right). \tag{2}$$

Clearly, if $G(n) > (1 - 2\psi(n))((n-1)M(n) + I(n))$, then Expression (1) is more expensive than Expression (2). By Table 1, $G(n) > nM(n)\log(n)$ for $n > 2$, so it is easy to verify that Expression (1) always dominates.

4.3 Computational complexity of Algorithm 2

Algorithm 2 largely resembles Algorithm 1, with the removal of the gcd-based normality check and the addition of a trace precomputation.

We focus on the main loop of Algorithm 2 and follow the notation from Table 1. Lines 7-11 are called 2^n times: lines 7 and 9 are constant time operations, so we ignore them, and line 8 has cost $nA(n)$. Indeed, the calculation of the Gray code index in line 7 is constant in an amortized sense, since $\sum_{i=1}^{n} i/2^i \leq 2$. Exactly 2^{n-1} elements pass to line 12 with cost $L(n)$, where $L(n)$ is the cost of checking canonicity (e.g., lex-first) on input of length n. Exactly $2^{n-1}/n$ elements enter Algorithm 3 in line 14. Line 7 of Algorithm 3 has cost $I(n)$ for each input; see Table 1. Exactly $\Phi(x^n - 1)/n$ elements pass beyond line 7 of Algorithm 3. The cost of the remaining lines is $(n-1)M(n) + \mathfrak{M}(n)$.

Theorem 4. *The cost $C(n)$ of the main loop of Algorithm 2, ignoring lower order terms, satisfies*

$$\frac{47}{6}n^2 > \frac{C(n)}{2^n} > \frac{11}{6}n^2 + 2.616\frac{n^2}{\log_2(n)}.$$

Proof. Let $C(n)$ be the cost of lines 6-16 of Algorithm 2 for $2 < n < 63$. Then we have

$$C(n) = 2^n nA(n) + 2^{n-1}L(n) + \frac{2^{n-1}}{n}I(n) + \frac{\Phi(x^n-1)}{n}\left((n-1)M(n) + \mathfrak{M}(n)\right).$$

A lex-first calculation can be implemented in n^2 bit comparisons, so we use $L(n) = n^2$. We recall the costs from Table 1 using schoolbook methods: $A(n) = n$, $M(n) = 4n^2 - 5n + 1$ $I(n) = (2/3)(n^3 + 3n^2 - 4n)$ and $\mathfrak{M}(n) = 2n^3 - n^2$. By Theorem 3, $\frac{\Phi(x^n-1)}{n} \geq \frac{2^n}{ne^{0.83(1+\log_2(n))}}$ and we have the trivial upper bound $\Phi(x^n - 1) < 2^n$. Therefore,

$$\frac{47}{6}n^2 > \frac{C(n)}{2^n} > \frac{11}{6}n^2 + 2.616\frac{n^2}{\log_2(n)}.$$ ■

5 Implementation details

In this section, we discuss some details of our implementation of Algorithm 2. We discuss our development methodology in Section 5.1 and give a demonstration of the importance of practical and efficient implementation in Section 5.2.

n	10	11	12	13	14	15	16	17	18	19
Theorem 1 (Hybrid)	210ms	323ms	604ms	1.4s	2.77s	5.72s	14.3s	30.1s	65s	157s
Implicit (Algorithm 2)	115ms	166ms	388ms	773ms	1.44s	2.84s	7.77s	12.8s	26.4s	55.9s
`C++` (Algorithm 2)	3ms	3ms	4ms	8ms	13ms	29ms	52ms	113ms	188ms	415ms

Table 2. Runtime for complete exhaustive search of $\mathbb{F}_{2^n}$ over $\mathbb{F}_2$ with different normality checks: the first using Theorem 1, and the second the implicit normality check as performed in Algorithm 2. Runtime for single-threaded `C++` implementation of Algorithm 2 is also given.

5.1 Development methodology

In this section, we discuss our methodology for determining which routines are most promising for implementation at scale. Our methodology is to use a platform that is agile and simple to test before spending a much larger effort to code an efficient `C++` implementation.

For our experimental development, we use the `Sage` computer algebra system [15] running in JuPyter Notebooks. `Sage` is open-source and is naturally linked to many other mathematical libraries (like `NTL`). `Sage` is built on Python, so it is interpreted, object-oriented, not strongly typed, and memory is internally managed.

JuPyter notebooks provide a user interface in a web browser linked to various kernels in the back-end: in our case a `Sage` interpreter. Code is written in "cells", which is particularly convenient for testing experimental changes. Effectively, for our purpose, JuPyter notebooks provides an easy-to-use, graphical, interactive interface to `Sage`. The trade-off of this ease of development (say, over direct implementation in `C++`) is in overall performance and lack of control over internals that our highly-efficient implementations use.

In our `Sage` notebook, we have a `Normal_Search` object, which accepts functions as arguments: in particular, specific implementations of canonicity checking, normality checking and complexity computation (Algorithm 3). We find the most efficient implementation in each function class by profiling the main routine of each instance of `Normal_Search` instantiated with each set of distinct implementations in each function class.

We use simple timing and line-profiling inherited in the `Sage` notebook to compare implementations of Algorithms 1 and 2. In particular, we compare the use of a normality check in Theorem 1 against the implicit normality check in Algorithm 3. In Table 2, we present comparative timings for a `Sage` implementation of Algorithm 1 *with added trace precomputation* and Algorithm 2 (the only difference between these algorithms is therefore the presence or absence of normality checking due to Theorem 1). We observe that Theorem 1 gives slower runtimes; this led us to do the careful analysis in Section 4.2. The third row of Table 2 also provides the runtime of our high performance `C++` implementation; see Section 5.2 for details.

We also used this comparative analysis to determine a more efficient implementation of canonicity checking between the one that appeared in [11] and this work. In Table 3, we give comparative timings for two different canonicity checks. We discuss this implementation in more detail in Section 5.2.

Our Sage JuPyter notebook appears under the GitHub project [18].

5.2 High performance implementation of Algorithm 2

After determining candidates for efficient implementations in `Sage`, we code these in a `C++` program using `NTL` for finite field arithmetic. We first implemented a single-threaded version and profiled the code until the dominant subroutines are algorithmically necessary `NTL` internal calls; see Figure 1, for example. We added two levels of parallelism: multi-node parallelism and multi-threading for intra-node parallelism. We give more details on our profiling in this section

and on our parallel implementation in Section 5.3. As we see in Table 2, the absolute runtimes in our `C++` implementation are 100-fold faster than in `Sage`, but importantly we notice that the relative timings of the two implementations are proportional.

Our reasoning for using `NTL` for arithmetic in our high-performance implementation included:

Familiarity: The authors have significant prior experience using `NTL`.

Comparability: Coding in NTL allowed us to compare our current code with the previous version from [11].

Performance: `NTL` is a mature library focusing on performance and contains a particularly fast implementation of $\mathbb{F}_{2^n}$ arithmetic.

Thread-safety: `NTL` has been thread-safe since version 7.0 (2014), though this feature has matured greatly as of version 9.8.0 (2016). For ease of intra-node parallelism, we make use of the `BasicThreadPool` library in `NTL`.

Exposure of pointers: `NTL` exposes pointers to arrays and to the bit representatives of elements of $\mathbb{F}_{2^n}$, which is crucial for our implementation.

We do not make any attempt to write our own vectorized instructions, and we rely on compiler optimizations to unroll loops and assign `AVX` instructions, if available. There may be scope for this sort of assembly-level optimization, but this is outside of the authors' area of expertise.

While we present our algorithms for clarity, their implementations may differ slightly from the presentation. Consider Algorithm 3, which accepts a putative normal basis and either returns its complexity or ∞ if it is not a basis. Here, we step through this algorithm in detail to show the differences that may occur between the implementation and the algorithmic description.

Algorithm 3 Check normality and calculate the complexity of a putative normal basis.

1: **Input**: A putative normal basis $\mathcal{B} = \{\alpha, \alpha^2, \ldots, \alpha^{2^{n-1}}\}$
2: **Returns:** The complexity $\mathcal{C}_\mathcal{B}$ of the normal basis $\mathcal{B}$ if $\mathcal{B}$ is a basis, otherwise ∞
3:
4: $P_\alpha \leftarrow \left(\alpha\ \alpha^2\ \cdots\ \alpha^{2^{n-1}}\right)$
5: **if** $\det(P_\alpha) = 0$ **then** ▷ New for Algorithm 2
6: **return** ∞
7: $P'_\alpha \leftarrow P_\alpha^{-1}$
8: $B_\alpha \leftarrow \left(\alpha\alpha\ \alpha\alpha^2\ \cdots\ \alpha\alpha^{2^{n-1}}\right)$
9: $T_\mathcal{B} = (t_{ij})_{i,j=0}^{n-1} \leftarrow B_\alpha P'_\alpha$ ▷ Multiplication table of $\mathcal{B}$
10: **return** $\sum_{ij} t_{ij}$ ▷ Complexity of $\mathcal{B}$, $\mathcal{C}_\mathcal{B}$

1. Each element α^{2^i} is represented as an n-vector over $\mathbb{F}_2$ using an implicit basis of $\mathbb{F}_{2^n}$ over $\mathbb{F}_2$.
2. The matrices P_α and B_α are initialized only once per thread, to save overhead.
3. The rows of the matrix $P_\alpha = \left(\alpha\ \alpha^2\ \cdots\ \alpha^{2^{n-1}}\right)$ are pointers assigned to the $\mathbb{F}_2$-vector representation of α^{2^i}.
4. Since $\alpha\alpha^{2^i}$ with $i = 0$ is simply α^2, hence already stored, we save a multiplication in line 8.
5. In `NTL`, the matrix inverse function returns the determinant of a matrix, and if the matrix is invertible procedurally stores the inverse in a passed argument. Hence, lines 5-7 are computed using a single function call.
6. It is critical to performance to use the procedure versions of the matrix inverse in line 7 and matrix multiplication in line 9 to avoid pass-by-values. In practice, we store $P'_\alpha = P_\alpha^{-1}$ in the existing P_α matrix and the product $B_\alpha P'_\alpha$ in the B_α matrix.

7. `NTL` does not have a matrix weight function, so we compute the complexity of $\mathcal{B}$ from $T_{\mathcal{B}}$ row-by-row in line 10. We improved in performance over the `NTL` weight function by dereferencing pointers to underlying `NTL` objects and calling the `gnu` built-in `popcount` intrinsic.

These low-level programming details are incredibly important to the efficiency of the *implementation* of the algorithm. We believe our presentation accurately represents the underlying algorithms of our search in a clear fashion for the reader.

Efficient Gray code successor To realize an efficient iteration through $\mathbb{F}_{2^n}$, we use an efficient implementation of a Gray code successor algorithm, such as [9, Algorithm 2.3].

Proposition 5. *Denote by $H(g)$ the Hamming weight of $g \in \mathbb{F}_2^n$. Let $g_0 = (0, 0, \ldots, 0) \in \mathbb{F}_2^n$ and let $g_i = (g_{i0}, \ldots, g_{i(n-1)})$ for $i = 0, \ldots, 2^n - 1$. Define $S\colon \mathbb{Z}_{>0} \to \mathbb{Z}_{\geq 0}$, where $S(k)$ is the number of trailing 0s in the binary expansion of k. Let $g_{i+1} = g_i + e_{S(i+1)}$ for $i \geq 0$. Then $G = \{g_0, g_1, \ldots, g_{2^n-1}\}$ is a Gray code of length n.*

In some compilers, the function S from Proposition 5 is given by a built-in intrinsic (for example, in our code we use the `gnu C++` compiler's `__builtin_ctzl` intrinsic).

The importance of profiling: efficient canonical element checking This section highlights the interplay between theory and practice: namely, the interplay between fast algorithms and computational complexity and the implementations of those algorithms. In particular, in Section 4.3 we describe the computational complexity of our new exhaustive search Algorithm 2 in comparison to Algorithm 1 from [11], and in this section we discuss evaluating the actual running time of the various components.

In our `C++` development, we use the `gperftools` (formerly Google Performance Tools) packages for CPU profiling; detailed documentation is available [7]. The profiler aggregates data about the "call tree" of a program. Specifically it samples a running program at regular intervals, and records the function that is running at interrupt time along with all the functions and its call stack. By analyzing the call tree, we can determine in a very practical sense the CPU-time bottlenecks of the program. An example of a current call tree (given by `kcachegrind` run on the output of `gperftools`) appears in Figure 1.

From the profiling output, we analyze bottlenecks in our implementation; for example, inefficiencies due to some objects being passed by value rather than by reference. The most surprising bottleneck came in our canonical element routine, where we found that approximately one-third of the total runtime was spent in this routine. While this routine is run $2^n/n$ times for each degree n, improving the runtime of this relatively simple subroutine was *as important* than algorithmic improvements to the overall performance of the program.

In [11], we implemented a naive bit-by-bit lexicographical checker. After profiling, we experimented with various canonical element checkers, the most performant of which uses an underlying integer representative of an element of $\mathbb{F}_{2^n}$. Explicitly, if $\alpha = \sum_{j=0}^{n-1} b_j \zeta_j$ is the expansion of α in an implicit basis $\{\zeta_0, \ldots, \zeta_{n-1}\}$, then its integer representation is $\sum_{j=0}^{n-1} b_j 2^j$. We notice on average an approximate 25% improvement on overall runtime due to this new implementation in both `Sage` development and `C++` production versions. We give a comparison of the runtimes of Algorithm 2 (in `Sage`) using these two routines in Table 3.

As indicated in Figure 1, after our optimizations the dominant functions in the call graph are `NTL` internal functions, indicating that we are reaching the limit of the efficiency of implementing the current algorithm.

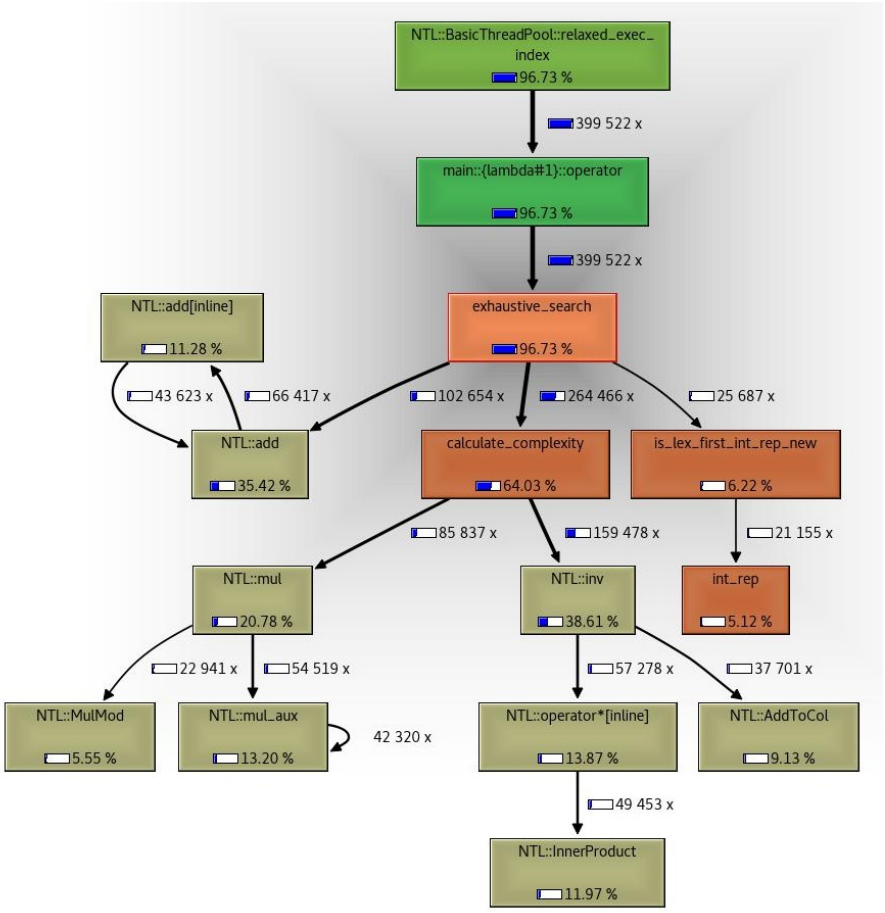

Fig. 1. Call tree for single-node `C++/NTL` implementation of Algorithm 2 for $\mathbb{F}_{2^{32}}$ over $\mathbb{F}_2$.

5.3 Parallelism of Algorithm 2

Algorithm 2 is an example of an *embarrassingly parallel* application; that is, the search function requires no communication between workers, so with an appropriate implementation we can avoid overhead in parallelism.

Due to the embarrassingly parallel nature of the search, we do not attempt any thread boosting of the underlying arithmetic. For example, in the Gray code successor, we could update $(\alpha+e_j)^{2^i} = \alpha^{2^i}+e_j^{2^i}$ for multiple i in parallel. However, this would reduce the number of available threads and would require a barrier after this computation to ensure data coherence. Hence, we only implement parallelism by partitioning the search space according to Gray code rank.

Our test system was a single 2-core Core i7-4500 CPU with 1.80 GHz clock speed run within a Linux virtual machine. We ran the code on a variety of available systems; in all cases the systems were Intel Core x86_64 architectures with `AVX` instructions. The code was compiled with the `GNU C++` compiler version 7.x and `NTL` version 10.5.0 compiled with thread safety (`NTL` version 11.0.0 was released after the pre-proceedings version of this paper).

The combination of iteration through elements in Gray code order with a canonicity check forces an imbalance in the computation, since the relatively expensive Algorithm 3 computation is performed for elements with extreme integer representation. To mitigate this imbalance, we partition the search space into a large number of "tasks". Each task represents a contiguous

n	10	11	12	13	14	15	16	17	18	19
Integer rep.	111ms	168ms	354ms	748ms	1.45s	3.07s	6.32s	12.9s	26.1s	55.3s
Bit-by-bit	130ms	250ms	456ms	951ms	1.87s	3.89s	7.71s	16.7s	32.2s	72s

Table 3. Runtime for `Sage` implementation of the exhaustive search of $\mathbb{F}_{2^n}$ over $\mathbb{F}_2$ with two lex-first checks: integer representative and bit-by-bit comparison.

region of Gray code indices; in practice, the number of tasks is set to be much larger than the number of available compute nodes, and compute nodes are assigned to tasks asynchronously.

When a compute node is assigned a task, it initializes a pool of n_c threads, where n_c is the number of cores on the node. The task is partitioned into n_c equally-sized sub-tasks executed synchronously in parallel one per thread. While asynchronous execution at the thread level may be preferable, in practice we assign a sufficient number of tasks that any overhead is acceptable.

We have implemented parallelism in a variety of ways. For intra-node parallelism, we originally had a direct `pthreads` implementation, but for simplicity we eventually removed this in favour of using `NTL`'s built in intrinsics. For inter-node parallelism, an earlier version of the code used MPI to distribute tasks to nodes. This is an acceptable strategy if a single executable is required, but specifying asynchronous computation, dealing with load-balancing and checkpointing prove to be complicated. Our preferred program accepts as input the task index as a command-line argument and is run independently one-per-node. This has the benefit of being able to rely on `gnu` compilers and not rely on platform-specific (and often less performant) compilers. In practice we assign a number of tasks so that all but the most expensive tasks finish within 5 (or so) minutes. This paradigm requires some additional job-scheduling by the user, but this can be a feature when using highly-shared computing platforms.

We observe that this computation is an excellent candidate for GPU processing, since minimal data communication is required between the CPU and GPU. However, we do not have a pool of GPUs readily available for use, nor do we have the expertise to attempt a port of `NTL`, or other underlying field arithmetic library, to CUDA/GPUs. We leave this for future work.

6 Results and future work

We conclude this paper by presenting the main result of the search, an update to the minimum complexity of a normal basis of $\mathbb{F}_{2^n}$ over $\mathbb{F}_2$, $n = 40, \ldots, 46$. We also present a number of avenues for future work. Our reference code can be accessed through GitHub [18]. See the documentation on that page for instructions on building and running the code.

By defining the macro `ALL_BASES` at compile-time, the code will print a $\mathbb{F}_2$-vector representing every normal basis, along with its complexity. Similarly, by defining `ALL_TABLES`, the code will print the multiplication tables of all bases. For a sense of scale, for $n \leq 29$, the `ALL_BASES` output totaled just over 1GB, and for $n \leq 25$ the `ALL_TABLES` output totaled 1.5GB. Some compressed output is available in the GitHub repository [18].

In most cases, the basis with the minimum complexity in $\mathbb{F}_{2^n}$ over $\mathbb{F}_2$ is unique. Two exceptions occur when $n = 18$ and $n = 19$. The $n = 18$ case is very special: it satisfies the conditions for both types of so-called "optimal normal bases" (bases with minimum possible complexity). Whether the $n = 19$ case is causal or a combinatorial surprise is unknown to us.

6.1 Running time of Algorithm 2

Table 4 provides the runtime for our high performance implementation of Algorithm 2 for $30 \leq n \leq 39$.

n	30	31	32	33	34	35	36	37	38	39
CPU time ($\log_2$(sec))	9.35	10.58	11.60	12.54	13.75	14.59	15.60	16.87	17.84	18.84

Table 4. CPU time for high performance implementation of Algorithm 2 on Intel Broadwell at 2.3GHz.

Our largest computation was for $n = 46$, which totaled $20,801$ CPU-hours on Intel Broadwell CPUs at 2.3 GHz. In core-wall-time, this totaled $24,816$ hours, indicating a near 20% overhead. This was a noticeable result of the synchronous execution of threads within the most compute-intensive tasks. For our eventual computation of $47 \leq n \leq 49$, we can mitigate this either by implementing asynchronous thread execution within a task, or reducing the size of the most compute-intensive tasks to mitigate any overhead.

By Theorem 4, Algorithm 2 scales exponentially in n, plus a quadratic factor. This can be seen for example, with some variability, in Tables 2 and 3. Based on the running time for $n = 46$ and assuming stability and homogeneity on production systems, we estimate approximately $287,000$ CPU hours, or approximately 33 CPU years, to complete the searches for $47 \leq n \leq 49$. This is well within the resource allocation budgets of, for example, annual Compute Canada awards; see [2], for more information.

6.2 Changes to table of minimal complexity normal bases

In [11], we presented tables of the minimum known complexity of a normal basis of any extension $\mathbb{F}_{2^n}$ over $\mathbb{F}_2$. These tables were repeated in [13, Section 2.2]. Our results here update the exhaustive tables in [11, 13]. Also in [11, 13], there appears a table of the minimum known complexity of $\mathbb{F}_{2^n}$ over $\mathbb{F}_2$ for $n \geq 40$ using all known methods and constructions. All non-explicit constructions take known normal bases from either subfields or extensions and use them to build normal bases in extensions and subfields, respectively.

In Table 5 we present the results of the search for $40 \leq n \leq 46$. Each number i in the set in the second column represents the term x^i appearing in the minimal polynomial of the basis. A `Sage` code snippet to check the minimal polynomial for $n = 45$ follows in Figure 2. For $40 \leq n \leq 46$, we find no elements having smaller complexity than those that appear in [13,

```
print minpoly45
F = GF(2)
Fx = PolynomialRing(GF(2), 'x')
modulus = 1 + Fx.gen() + Fx.gen()^3 + Fx.gen()^4 + Fx.gen()^(45) #The NTL default modulus for n=45
K = GF(2**n, name='x', modulus=modulus) #GF(2^n) = GF(2)[x]/(modulus)
congs = minpoly45.roots(K, multiplicities=False) #The roots of minpoly45 are a set of conjugates
B = matrix([ (congs[i]*congs[0])._vector_() for i in range(n) ]) #Construct the matrix B_{cong[0]}
P = matrix( [ congs[i]._vector_() for i in range(n) ]) #Construct the matrix P_{\cong[0]}
Bpinv = B*P.inverse() #Construct BP^{-1}

print len(Bpinv.nonzero_positions()) #The complexity of the basis is the number of nonzero positions in the matrix
```

```
x^45 + x^44 + x^42 + x^41 + x^40 + x^39 + x^38 + x^37 + x^33 + x^30 + x^23 + x^20 + x^18 + x^16 + x^14 + x^13 + x^1
2 + x^11 + x^9 + x^6 + x^2 + x + 1
153
```

Fig. 2. Code snippet to check the complexity of a normal basis from its minimal polynomial.

Table 2.2.10]. These complexities are confirmed as the minimum complexities of any normal basis of $\mathbb{F}_{2^n}$ over $\mathbb{F}_2$, but no additional updates to [13, Table 2.2.10] result from bootstrapping methods.

Our eventual goal is to complete the search for $47 \leq n \leq 49$. The most likely candidate to find a previously unknown minimum complexity basis is in the $n = 48$ case, since the best known

n	Nonzero terms of minimal polynomial	Complexity
40	$\{40, 39, 37, 34, 31, 26, 24, 23, 21, 19, 18, 16, 9, 5, 0\}$	189
41	$\{41, 40, 38, 37, 36, 33, 32, 22, 21, 20, 17, 16, 9, 8, 6, 5, 4, 1, 0\}$	81
42	$\{42, 41, 40, 38, 36, 35, 31, 30, 26, 23, 22, 20, 19, 18, 15, 12, 3, 2, 0\}$	135
43	$\{43, 42, 40, 37, 35, 33, 31, 30, 29, 28, 27, 25, 24, 22, 20, 18, 14, 12, 11, 9, 8, 7, 5, 3, 0\}$	165
44	$\{44, 43, 42, 40, 35, 33, 30, 28, 27, 26, 25, 24, 23, 21, 19, 18, 17, 12, 11, 10, 9, 5, 3, 2, 0\}$	147
45	$\{45, 44, 42, 41, 40, 39, 38, 37, 33, 30, 23, 20, 18, 16, 14, 13, 12, 11, 9, 6, 2, 1, 0\}$	153
46	$\{46, 45, 44, 42, 40, 39, 36, 32, 29, 28, 26, 23, 20, 15, 13, 10, 7, 4, 0\}$	135

Table 5. Minimum complexity normal basis generator of $\mathbb{F}_{2^n}$ over $\mathbb{F}_2$.

complexity here is 425: easily the largest minimum complexity for any $n < 57$. We do not expect to extend the table beyond $n = 50$, since there are optimal normal bases for $n = 50, 51, 52, 53$, and for $n = 54$ there is a normal basis with complexity $4n - 7$ by construction, so we expect that this basis is minimal. Searching $n \geq 55$ is well out of reach for reasonable computational resources.

6.3 Choice of basis representation

All `NTL` field arithmetic is represented using a polynomial (or power) basis. Suppose instead that we choose a normal basis representation. For any $\alpha \in \mathbb{F}_{2^n}$, computing the set $\mathcal{B} = \{\alpha, \alpha^2, \ldots, \alpha^{2^{n-1}}\}$ under normal basis representation can be done by a series of cyclic bit shifts, so we no longer need to iterate through $\mathbb{F}_{2^n}$ in Gray code order.

In normal basis representation, we also keep an efficient cut-down from the trace computation. Recall that $\mathrm{Tr}(\alpha) = \alpha + \alpha^2 + \cdots + \alpha^{2^{n-1}}$, where α^{2^i} is the i-fold shift of the underlying bit vector of α. Hence $\mathrm{Tr}(\alpha) = \sum_{j=0}^{n-1} \alpha_j \pmod 2$, where α_j is the jth bit of α in normal basis representation. The weight of a vector can be taken by modern computer architectures in a single `popcount` operation, hence the trace computation can be considered essentially free.

Regardless of basis, calculating normality (either by gcd as in Theorem 1 or by matrix inversion) and complexity requires computing field multiplications. Moreover, the cost of multiplication depends precisely on the complexity of the underlying basis. *A priori*, we expect a random basis of $\mathbb{F}_{2^n}$ over $\mathbb{F}_2$ to have complexity about $n^2/2$, see [11], hence this multiplication will be expensive *until after we find a low-complexity normal element.*

Other (non-normal) bases may be appropriate as well, but any other choice of basis would require hand-rolling a new finite field arithmetic library. Hence, we leave this for future work.

6.4 A best-case scenario

Any algorithm that goes exhaustively through every normal element computing their bases must have cost at least

$$(n-1)M(n)\frac{\Phi(x^n-1)}{n},$$

where $\Phi(x^n - 1)/n$ is the number of normal basis of $\mathbb{F}_{2^n}$ over $\mathbb{F}_2$. This expression assumes that we compute the representation of $B_i = \alpha\alpha^{2^i}$ for $i = 1, \ldots, n-1$, while looping only over normal elements and also having B_i represented immediately in the normal basis. It is unclear that the latter part is even possible. However, it is possible in spirit to loop over only normal elements using the *primary decomposition*, seen in [3], for example. We state the main result as follows.

Proposition 6. *Let $q = p^e$ for some prime p and positive integer e and let $\gcd(n, p) = 1$ so that $x^n - 1 = f_1 f_2 \cdots f_k$, where the f_i are distinct irreducible factors over $\mathbb{F}_q$. If $f_i(x) = \sum_{j=0}^{n} f_{ij} x^j$,*

let $F_i(x) = \sum_{j=0}^{n} f_{ij} x^{q^j}$. *Each* F_i *is a* linearized polynomial, *hence a linear operator over* $\mathbb{F}_q$, *and its roots form a vector subspace of* $\mathbb{F}_{q^n}$ *over* $\mathbb{F}_q$ *that is stable under* q*th powers. Let* $V_i = \ker(F_i)$, *then* $\alpha = \sum_{1 \le i \le k} \alpha_i$ *with* $\alpha_i \in V_i$ *is a normal element of* $\mathbb{F}_{q^n}$ *over* $\mathbb{F}_q$ *if and only if* $\alpha_i \neq 0$ *for all* $1 \le i \le k$.

Proposition 6 can be modified to remove the restriction on the field degree, though the choices of α_i become more complicated. In order to iterate through only normal elements, we can iterate through the nonzero elements of each of the V_i vector subspaces of $\mathbb{F}_{q^n}$. It is unclear how to do this efficiently, so we leave this for future work.

6.5 The case $n = 64$

We are still unaware of any low complexity normal bases of $\mathbb{F}_{2^n}$ over $\mathbb{F}_2$ when n is a power of 2, except for those obtained by exhaustive search ($n = 2^\ell, \ell \le 5$). When n is a power of 2, we realize a dramatic speedup due to the following specific version of Proposition 3.

Proposition 7. [13, Corollary 5.2.9] *An element* $\alpha \in \mathbb{F}_{2^{64}}$ *is normal if and only if* $\mathrm{Tr}(\alpha) \neq 0$.

By Proposition 7, precisely 2^{63} elements of $\mathbb{F}_{2^{64}}$ are normal, so with a trace pre-computation all elements passing into Algorithm 3 are normal. Hence, with the canonicity check, we compute the complexity of exactly 2^{57} elements. Each computation requires 63 multiplications (each estimated at 64^2 bit operations) and a 64×64 matrix inversion over $\mathbb{F}_2$ (estimated at 64^3 bit operations), so in total we estimate at least 2^{76} bit operations to calculate all of their complexities. This seems out of reach for reasonable modern computational power. This cost may be lowered by a few bits using an early-abort strategy, but any such exhaustive search is likely to be untenable for the foreseeable future. Of course, any algorithm that inspects every element of $\mathbb{F}_{2^{64}}$, as ours does, has 2^{64} as a lower bound on the running time.

Acknowledgement We would like to thank the three reviewers for their helpful suggestions, which greatly improved the presentation of this paper.

References

1. F. Arnault, E. J. Pickett, S. Vinatier, Construction of self-dual normal bases and their complexity, Finite Fields and Their Applications, **18** (2012), 458-472.
2. Compute Canada, 2018 Final Allocations for Publication, available: `https://www.computecanada.ca/research-portal/accessing-resources/resource-allocation-competitions/rac-2018-results/`, last date accessed May 17, 2018.
3. I. F. Blake, S. Gao and R. C. Mullin, Specific irreducible polynomials with linearly independent roots over finite fields, Linear Algebra and Its Applications, **253** (1997), 227-249.
4. S. Gao and H. W. Lenstra, Optimal normal bases, Designs, Codes and Cryptography, **2** (1992), 315–323.
5. S. Gao and D. Panario, Density of normal elements, Finite Fields and Their Applications, **3** (1997), 141–150.
6. J. von zur Gathen and J. Gerhard, Modern Computer Algebra, Cambridge University Press, Cambridge UK, 2013.
7. gperftools, Authors unlisted © Google Inc. (2005), documentation available online `www.github.io/gperftools`, last date accessed: March 27, 2018.
8. D. Jungnickel, Finite Fields: Structure and Arithmetics, Bibliographisches Institut, Mannheim GE, 1993.
9. D. L. Kreher and D. R. Stinson, Combinatorial Algorithms: Generation, Enumeration and Search, CRC Press, Boca Raton, FL, 1999.
10. R. Lidl and H. Niederreiter, Finite Fields, Cambridge University Press, Oxford UK, 1997.
11. A. Masuda, L. Moura, D. Panario and D. Thomson, Low complexity normal elements over finite fields of characteristic two, IEEE Transactions on Computers, **57** (2008), 990–1001.
12. R. C. Mullin, I. M. Onyszchuk, S. A. Vanstone and R. M. Wilson, Optimal normal bases in $GF(p^n)$, Discrete Applied Mathematics, **22** (1988/9), 149–161.
13. G. L. Mullen and D. Panario, Handbook of Finite Fields, CRC Press, Boca Raton, 2013.

14. National Institute for Standards in Technology, Digital Signature Standard FIPS PUB 186-4, 2013.
15. SageMath, the Sage Mathematics Software System (Version 7.1.0), The Sage Developers, 2018, `http://www.sagemath.org`.
16. V. Shoup, NTL: Number Theory Library, documentation available online `www.shoup.net/ntl`, last date accessed: March 3, 2018.
17. D. Silva and F. Kschischang, Fast encoding and decoding of Gabidulin codes, 2009 IEEE International Symposium on Information Theory, 2009, 2858–2862.
18. D. Thomson, Exhaustive search for normal basis, Version 1.0.0, `www.github.com/dgthoms/exh_normal`, last updated: May 26, 2018.