

# LinCodeWeightInv: Library for Computing the Weight Distribution of Linear Codes Over Finite Fields

MARIA PASHINSKA-GADZHEVA, Institute of Mathematics and Informatics, Bulgarian Academy of Sciences, Bulgaria

ILIYA BOUYUKLIEV, Institute of Mathematics and Informatics, Bulgarian Academy of Sciences, Bulgaria

We present the Linear Codes Weight Invariant Library (LinCodeWeightInv) for optimized computing of the weight distribution and other weight characteristics of a random linear code (minimum distance, number of codewords with a given weight, etc.). The presented library is developed for linear codes over finite fields with up to 64 elements. We use two main methods for optimizations - efficient algorithms for generating the codewords and integration of extended vector registers with SSE4.1, AVX2 and AVX512 instructions sets for x86 architectures and NEON instructions set for ARM. The LinCodeWeightInv is compared to other software systems for linear codes over finite fields. Comparing our library to the Magma software we get between 1.3 and 4 times faster execution times for  $\mathbb{F}_2$  and  $\mathbb{F}_5$  and up to 49 depending on the field and the code length. Comparing the LinCodeWeightInv library to the open source software GAP we observe a reduction in computation time by factors between 5 and 7 for  $\mathbb{F}_2$ . For other finite fields, we observe more than 100 times faster execution times.

CCS Concepts: • **Mathematics of computing** → **Coding theory**.

Additional Key Words and Phrases: linear codes, weight spectrum, finite fields

## ACM Reference Format:

Maria Pashinska-Gadzheva and Iliya Bouyukliev. 2024. LinCodeWeightInv: Library for Computing the Weight Distribution of Linear Codes Over Finite Fields. 1, 1 (November 2024), 22 pages. <https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

## 1 INTRODUCTION

Computing the weight distribution of a linear code is an important problem in Coding Theory. The weight distribution takes a crucial place in the methods and algorithms for construction, classification and computation of various characteristics and parameters of linear codes. It is useful for analysing the performance of a code under maximum-likelihood decoding and various other decoding algorithms. The probability of undetected error of a linear binary code used to detect errors in a symmetric memoryless channel is expressed in terms of the code weight distribution [12]. Therefore, the computation of the weight distribution of a linear code over a finite field is part of any coding theory software.

It is known that the general problem of finding the weights of a linear code is NP-complete [4]. There are families of codes whose weight distribution is known, for example the families of Hamming, Golay and Reed-Solomon codes. Different approaches for computing the weight distribution of codes of special types have been used, for example for BCH codes [14], cyclic codes

---

Authors' addresses: Maria Pashinska-Gadzheva, [mariqpashinska@math.bas.bg](mailto:mariqpashinska@math.bas.bg), Institute of Mathematics and Informatics, Bulgarian Academy of Sciences, Veliko Tarnovo, Bulgaria; Iliya Bouyukliev, [iliyab@math.bas.bg](mailto:iliyab@math.bas.bg), Institute of Mathematics and Informatics, Bulgarian Academy of Sciences, Veliko Tarnovo, Bulgaria.

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM XXXX-XXXX/2024/11-ART

<https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

[3], polar codes [33], etc. In the general case, the natural way to find the weight distribution is to generate all nonproportional codewords and to compute their weights [7, 16]. Another method that can be applied to any linear code uses a characteristic vector of the code and fast discrete transforms [8]. It is effective for linear codes with small rate (large length and small dimension) and suitable for parallel implementations with GPU [28]. More information about algorithms for computing the weight distribution of linear codes is given in [8]. Some of them are implemented in the software systems related to Coding Theory, such as computer algebra system Magma [5], and the GAP package GUAVA [11]. However, these systems are not intended to be used specifically for classification of linear codes. The presented library is intended to be used for the development of optimized classification algorithms as well as a stand alone software.

In this work, we present a C/C++ library for computing various parameters of a linear code over a finite field with a given generator matrix related to the weight distribution of the code. It is based on the advanced processor instructions inherent in most modern processor architectures such as x86, ARM and others. They introduce a SIMD parallelization capabilities of the computations. This is also the main difference from the already existing software. Some compilers in simple cases can integrate extended processor SIMD instructions automatically, but we use these advanced capabilities in special low-level algorithms that have different logic for different fields and different organization of data into vectors that correspond to SIMD instructions.

The main features of the presented library are the following:

- The library can compute weight parameters such as minimum distance, number of codewords with a given weight, the weight distribution, as well as whether the minimum distance is less than a given positive integer  $w$ , for a linear code over a finite field with  $q \leq 64$  elements.
- The used algorithms and the organization of the data allow the generation of each new codeword to be done only by vector additions. Moreover, only the non-proportional codewords are generated.
- The code is highly optimized by utilizing extended vector registers. Modern CPUs have Single Instruction Multiple Data (SIMD) extended instructions for handling vectors of multiple data elements in parallel.
- A bitwise representation of the elements is used for fields with characteristic 2 and 3. Thus, bitwise operations are used in the addition of vectors. This is described in Section 4.
- Functions from the library can be used in any linear code analysis software without the need to know the implemented algorithms or internal functions.
- Experimental results compare the execution time of a corresponding functions in the software package Magma and the open source package GAP. The obtained speedup depends on the length of the code, as well as the finite field itself. Thus, we can observe more than 4 times faster execution times for prime fields and more than 30 times for composite fields comparing the presented library to Magma. Comparing the experimental execution times with GAP, we can observe more than 100 times faster computations in the non binary cases.

The library is developed to be portable on a variety of operating systems (Windows, MacOS, Ubuntu, Red Hat, etc.) and two target architectures (x86 architectures and ARM architectures). The presented examples are focused on the implementation on x86 architectures since they are among the most popular ones. More on the difference between instruction sets for x86 (SSE, AVX, etc) and the NEON instruction set for ARM can be found in [27].

The rest of the paper is structured as follows: some preliminaries are given in Section 2, basic algorithms for generating all nonproportional codewords of codes over different finite fields are presented in Section 3. We refer to those algorithm as high-level. Our optimizations are given in

Section 4, experimental results can be found in Section 5, and a conclusion remarks are given in Section 6.

## 2 PRELIMINARIES

In this section we present the basic definitions, some previous work on the problem and some CPU instructions needed for the realizations. The mathematical definitions and notations can be found in the fundamental books [21], [20].

### 2.1 Mathematical Background

Let  $\mathbb{F}_q^n$  be the  $n$ -dimensional vector space over the finite field  $\mathbb{F}_q$  with  $q$  elements, where  $q = p^m$  for a prime  $p$  and positive integer  $m$ . For each prime number  $p$ , the prime field  $\mathbb{F}_p$  of order  $p$  may be constructed as the integers modulo  $p$ ,  $\mathbb{Z}/p\mathbb{Z}$ . The elements of  $\mathbb{F}_p$  may be represented by the integers  $\{0, 1, \dots, p-1\}$  with addition and multiplication modulo  $p$ .

For any prime  $p$  and positive integer  $m$ , there is at least one irreducible polynomial  $g(x) \in \mathbb{F}_p[x]$  of degree  $m$ . This polynomial can be used as a generator polynomial of the composite field with  $q = p^m$  elements, so we may consider  $\mathbb{F}_q = \{r(x) \in \mathbb{F}_p[x], \deg r(x) < m\}$  with addition and multiplication modulo  $g(x)$  (this is the additive representation of the elements of the field). The elements of  $\mathbb{F}_q$  can be represented as vectors in  $\mathbb{F}_p^m$  using the correspondence  $r(x) = r_0 + r_1x + \dots + r_{m-1}x^{m-1} \mapsto (r_0, r_1, \dots, r_{m-1})$ . The nonzero elements of the field form the cyclic group  $\mathbb{F}_q^* = \{1, \alpha, \dots, \alpha^{q-2}\}$ ,  $\alpha^{q-1} = 1$ , generated by a primitive element  $\alpha \in \mathbb{F}_q$ .

A  $k$ -dimensional subspace  $C$  of  $\mathbb{F}_q^n$  is called a linear  $[n, k]$  code with length  $n$  and dimension  $k$ . A  $k \times n$  matrix  $G$  whose rows form a basis of  $C$  is called a generator matrix of the code. Linear codes can be defined, constructed and, in many cases, used in terms of generator matrices. A large number of problems on linear codes are defined as search problems on generator matrices. The vectors in  $C$  are called codewords.

The *Hamming weight*  $\text{wt}(v)$  of a vector  $v \in \mathbb{F}_q^n$  is the number of its nonzero coordinates. The Hamming distance between two vectors  $u = (u_1, \dots, u_n)$  and  $v = (v_1, \dots, v_n) \in \mathbb{F}_q^n$  is  $d(u, v) = \#\{i | u_i \neq v_i\}$ . Then, the minimum distance and the minimum weight of the linear code  $C$  are defined by

$$d(C) = \min\{d(a, b) | a, b \in C, a \neq b\}, \quad \text{wt}(C) = \min\{\text{wt}(c) | c \in C \setminus \{0\}\}.$$

The minimum distance of a linear code coincides with its minimum weight. A linear code with minimum distance  $d$  can detect up to  $d-1$  errors and can correct up to  $\lfloor (d-1)/2 \rfloor$  errors.

The sequence  $(A_0, A_1, \dots, A_n)$  is called the weight distribution of the code  $C$ , where  $A_i$  is the number of codewords with weight  $i$  in  $C$ ,  $i = 0, 1, \dots, n$ . The most used algorithm for computing the minimum distance of a linear code in practice is the Brouwer-Zimmermann algorithm [9, 29, 32]. For small rate and dimension, the minimum distance can be determined through the weight distribution of the code.

The problems of computing the weight distribution and minimum distance of a linear code is known to be NP-complete. In most cases, however, computing the weight distribution of a linear code is based on the generation of  $(q^k - 1)/(q - 1)$  nonproportional codewords for codes over a field with  $q$  elements. For prime fields, in the generation process, each new codeword can be obtained by adding one codeword. This can be realized either by using a Gray code [16] or an additional temporary matrix [7]. If the field is composite, data pre-preparation and more memory are required to achieve similar performance, as shown in Section 3.

## 2.2 Extended Vector Registers and Instructions

The computer word in modern computers consists of 64 bits. Therefore, the base instructions are for 64-bit data. But the availability of the extended 128, 256, and even 512-bit registers in contemporary CPUs makes it possible to expand the processor's instruction set. These advanced instructions use the concept of SIMD to perform a single operation on multiple scalar data at the same time. Typically, instructions that operate on extended registers treat the information in them as arrays (vectors) of integer or real numbers. These instructions can perform the operations independently on each element of the array simultaneously. In hardware, this can be implemented by having multiple ALUs (arithmetic logic units) running in parallel. This process is known as vectorization. As a result, although these instructions perform many more arithmetic operations than standard ones, they can be approximately as fast as standard instructions. The different architectures' manufacturers have developed their own SIMD types of registers. Therefore, there are different instruction sets for work with the extended vector registers. Our library works with the extended instructions sets SSE4.1, AVX2 and AVX512 developed by Intel [22] that targets x86 architectures and the NEON set for ARM [2]. In this section and the presented examples in Section 4 we focus on the SSE4.1 and AVX2 instruction sets and the presentation of the low-level algorithms with them since they are the most widely available instruction sets for both workstation and HPC systems. These instruction sets introduce new data types and special functions for manipulating this data. The data types show how to interpret the information in a register. For example, the data type `__m128i` shows that a 128-bit register contains packed integers. In addition, the names of the functions that operate on this data contain information about the length of the integers (8, 16, 32, or 64 bits) and whether these functions use signed or unsigned data. Thus, function `_mm_add_epi8` adds 16 eight-bit integers from two registers of type `__m128i`. These functions and data types are defined in the `<intrin.h>`, `<smmintrin.h>`, `<emmintrin.h>`, `<immintrin.h>` libraries for Windows and `<x86intrin.h>` library for Linux. The effect of their use is an acceleration (increase in speed) of the computations. In our case, for example, these extended instructions allow a 256-bit register to be treated as 32 eight-bit integers, each of which represents an element of a prime field.

In addition to typical operations with vectors such as addition and subtraction, these advanced instructions provide many supplementary possibilities that greatly facilitate or improve the efficiency in the program implementation of some algorithms, for example:

- Shifting or executing a certain permutation of the elements of the vector (eg. `_mm_srli_si128 (__m128i a, int imm8)`).
- Obtaining a vector whose elements are selected from two different vectors according to the corresponding values in a third (mask) vector (eg. `_mm_blendv_epi8 (__m128i a, __m128i b, __m128i mask)`).
- Comparing the corresponding elements in two different vectors according to "less than" relation and storing 0xFF if true and 0 otherwise in the resulting vector (eg. `_mm_cmplt_epi8 (__m128i a, __m128i b)`).

These instruction sets contain functions with different complexities that can be roughly categorized as heavy (multiplication, permutation, etc.) and light (bitwise operations, addition, etc.) instructions. Additional information on using the extended registers and instructions can be found in [13, 18, 26].

The AVX512 instruction set is divided into different categories where most CPUs do not have the full set of instructions. Most of the current CPUs with 512-bit registers have the foundation instruction set AVX512F. Even though AVX and AVX2 can be considered as extending the functionalities of SSE instruction sets over larger registers, some of the functions in AVX512 are different than the

respective functions for the smaller registers. Therefore, AVX512 implementation is commented on in Section 4 only if the main idea of the presented algorithms is completely different.

The NEON instruction set developed for ARM architectures also allows packed data to be saved in 128-bit register as a vector. Thus, it allows for SIMD type of parallelism. The difference between corresponding types of instructions from SSE and NEON sets is described in details in [27]. However, the basis of the low-level algorithms is relatively the same regardless of the instruction set.

Most modern CPUs also have a specialized instruction that counts the number of nonzero bits in a computer word. It is often referred to as a population count instruction or *popcnt*. It is essential for the algorithms described in Section 4. It has a higher complexity than most standard operations. If the available architecture does not have this hardware instruction, we use the algorithm described in [10]. A subcategory of AVX512 (*AVX512 VPOPCNTDQ*) contains a set of instructions that calculate the number of nonzero bits in the corresponding 64-bit computer words that are stored in the register. The defined function are for 128-, 256- and 512-bit registers and the result is stored in a register of the corresponding length. The result can be viewed as a vector  $x = (x_1, \dots, x_n)$ , where  $n$  is the number of 64-bit words that are stored in the register (for 512-bit register  $n = 512/64 = 8$ ) and  $x_i$  is the number of nonzero bits of the corresponding element  $i$  stored in the register. Therefore, we can calculate the number of nonzero bits in a full register by summing (with proper instruction) the coordinates in  $x$ . The NEON instruction set does not have a direct equivalent to either the *popcnt* instruction or the 128-bit instruction defined in AVX512. However, 64- and 128-bit registers can be represented as vectors of eight 1-byte elements and sixteen 1-byte elements, respectively. There are two instructions that count the number of nonzero bits in each byte of the described representation. The result similarly to the 512-bit instructions is saved in a register of the appropriate length and the total count of nonzero bits can be calculated by summing the elements of the resulted vector by a single function of the NEON instruction set. Therefore, we have an implementations to count the nonzero bits in registers for SSE, AVX, AVX512 and NEON instruction sets. In the rest of the paper we use the notation *popcnt* for a calculation of the number of nonzero bits in a 64-bit computer word for simplicity of the presentation of the algorithms.

### 3 HIGH-LEVEL ALGORITHMS

In this section we present the implemented algorithms for the generation of the codewords of a linear code  $C$  represented by its generator matrix  $G$ . There are different implementations for prime and composite fields with some modifications for special cases ( $\mathbb{F}_2$  and  $\mathbb{F}_3$ ). The low-level algorithms are represented as functions *add(v1,v2)* and *get\_weight(v)*. These functions return the result of the operation  $v1 + v2$  and the number of non-zero coordinates of  $v$ , respectively.

#### 3.1 Basic Algorithm for Generation of the Codewords

We need to calculate all linear combinations of the rows of the generator matrix  $G$  to obtain all codewords of the linear code  $C$ . The popular method is to use Gray code [23] for the generation, especially for binary codes. Our implementation is more flexible, thus facilitating computations over composite fields. Algorithm 2 shows a recursive approach for generation of all nonproportional codewords. This algorithm simulates nested loops as shown in [7]. We use a temporary  $(k + 1) \times n$  matrix  $T$  with rows  $T[0], T[1], \dots, T[k]$ , where  $T[0]$  is equal to the zero vector, and  $T[i]$  is equal to a linear combination of exactly  $i$  rows of  $G$  (with nonzero coefficients),  $i = 1, \dots, k$ . This allows us to generate the next linear combination of  $i + 1$  rows by adding exactly one row of  $G$  multiplied by a nonzero element of the field to the  $i$ -th row of  $T$ . We consider only those linear combinations of the rows of  $G$  in which the first nonzero coefficient is equal to 1. This allows us to generate only nonproportional codewords. The generation of the needed linear combinations can be implemented

by  $2k - 1$  nested loops that traverse the rows of the matrix  $G$  and show which element of the field to use as a coefficient in order to obtain the current codeword. For the current element of the field we use  $l(e) = e$  for the prime fields, and  $l(e) = \alpha^{q-e}$  for the composite fields, where  $\alpha$  is a primitive element. This is shown in Algorithm 1. More detailed description of this algorithm can be found in [7]. Obviously, this algorithm cannot be implemented in a program, it only describes the idea.

---

**Algorithm 1** Basic algorithm for generation of nonproportional codewords

---

```

1: global definition
2:   int  $G[k][n]$  - global two-dimensional array containing  $k \times n$  generator matrix
3:   int  $T[k+1][n]$  - global two-dimensional array containing  $(k+1) \times n$  temporary matrix
4: end global definition
5: function LINEAR_COMBINATIONS_SEQUENTIAL(int  $n$ , int  $k$ )
6:   for ( $i_1 = 1$ ;  $i_1 \leq k$ ;  $i_1++$ ) do  $T[1] = G[i_1]$ ;
7:     for ( $i_2 = i_1 + 1$ ;  $i_2 \leq k$ ;  $i_2++$ ) do
8:       for ( $e_2 = 1$ ;  $e_2 < q$ ;  $e_2++$ ) do  $T[2] = T[1] + l(e_2)G[i_2]$ ;
9:       ...
10:      for ( $i_k = i_{k-1} + 1$ ;  $i_k \leq k$ ;  $i_k++$ ) do
11:        for ( $e_k = 1$ ;  $e_k < q$ ;  $e_k++$ ) do  $T[i_k] = T[i_{k-1}] + l(e_k)G[i_k]$ ;
12:        end for
13:      end for
14:    end for
15:  end for
16: end for
17: end function

```

---

Algorithm 2 gives a recursive implementation of the presented method for generation of all nonproportional codewords as linear combinations of the rows of a given generator matrix  $G$ . The variable  $qf$  shows the number of the possible coefficients to the summands in the considered linear combinations. Note that the coefficient to the first summand is always 1. This guarantees the generation of nonproportional codewords (Row 10 in the algorithm). Rows 12 and 13 in the recursive algorithm correspond to a pair of nested loops iterating the values  $i_j$  and  $e_j$  in the sequential algorithm (e.g. rows 7 and 8). Since the code contains at most  $\theta = (q^k - 1)/(q - 1)$  nonproportional codewords, we have  $\theta$  vector additions,  $\theta$  scalar-vector multiplications and  $\theta$  computations of the weight of the resulting codewords.

Although there is multiplication by a scalar, the implemented optimizations presented in the following subsections reduce the computation time, as each new codeword is obtained by just adding a vector, taking into account also a suitable representation of the data (using precomputation in the case of composite fields).

The main idea of the following algorithms show which rows from matrices  $G$  and  $T$  to choose for the generation of the current codeword and in which row of  $T$  to put the result.

### 3.2 Prime fields

We can substitute the multiplication of a vector by a scalar over prime fields with a repeated addition of the same vector. As can be seen in rows 7 and 8 in Algorithm 3, to multiply a row of the generator matrix by a nonzero scalar  $\neq 1$  we add the current row  $G[i]$  to the current linear combination of  $r$  rows. Thus, we avoid the more expensive and complicated multiplication operation. This is in exchange of introducing an additional *if* statement. There are two cases where the multiplication

**Algorithm 2** Basic algorithm for computing the weight distribution of a  $q$ -ary linear code

---

```

1: global definition
2:   int  $G[k][n]$  global two-dimensional array containing  $k \times n$  generator matrix
3:   int  $T[k+1][n]$  - global two-dimensional array containing  $(k+1) \times n$  temporary matrix
4:   int  $A[n+1]$  - global array containing the weight distribution
5:   int  $n, k, q$  global parameters showing the parameters of the code
6:   int  $get\_weight(v)$  - global external function for calculation of the weight of vector  $v$ 
7: end global definition
8: function LINEAR_COMBINATIONS(int r, int h)
9:   int  $qf = q$ ;
10:  if  $h=1$  then  $qf = 2$ 
11:  end if
12:  for (int  $i = h$ ;  $i \leq k$ ;  $i++$ ) do
13:    for (int  $e = 1$ ;  $e \leq qf$ ;  $e++$ ) do
14:       $T[r] = T[r-1] + l(e)G[i]$ ;
15:      int  $w = get\_weight(T[r])$ ;
16:       $A[w]++$ ;
17:      if  $r < k$  then Linear_Combinations( $r+1, i+1$ );
18:    end if
19:  end for
20: end for
21: end function
22: function WEIGHT_DISTRIBUTION( $G_{curr}, n_{curr}, k_{curr}, q_{curr}$ )
23:    $G = G_{curr}$ ;  $n = n_{curr}$ ;  $k = k_{curr}$ ;  $q = q_{curr}$ ;
24:   for (int  $i = 0$ ;  $i \leq n$ ;  $i++$ ) do  $A[i] = 0$ ;
25: end for
26: /* all precomputations needed in the modified algorithm are executed here */
27:   Linear_Combinations(1,1);
28: end function

```

---

can be easily removed without implementing an *if* statement for linear codes over  $\mathbb{F}_2$  and  $\mathbb{F}_3$ . For the field with two elements the *if* statement is not needed since there are not nonzero elements  $\neq 1$ . Furthermore, row 3 from the standard Algorithm 3 can be omitted because there are no proportional codewords. This alteration can be seen in Algorithm 4. When calculations are executed for  $\mathbb{F}_3$  we can use an arithmetic calculation of the index of temporary matrix  $T$  to which we need to add the current row depending on the scalar. This is accomplished by replacing rows 7 and 8 in Algorithm 3 with the statement  $T[r] = add(T[r + l(e) - 2], G[i])$ .

### 3.3 Composite fields

The elements of the field  $\mathbb{F}_q$  where  $q = p^m$  for a prime  $p$  and an integer  $m > 1$  can be represented as polynomials whose coefficients are in  $\mathbb{F}_p$  (the additive representation of the elements). We also consider the representation of the elements as vectors and integers as shown below.

$$y = a_0 + a_1x + \dots + a_{m-1}x^{m-1} \mapsto v = (a_{m-1}, \dots, a_0), v \in \mathbb{F}_p^m. \quad (1)$$

The additive representation allows a simple implementation of the addition operation in the field. However, the multiplication of polynomials is computationally more expensive [17]. Therefore, for the implementation we use the set of matrices  $\{G, \alpha G, \dots, \alpha^{m-1}G\}$ , where  $\alpha$  is a primitive

---

**Algorithm 3** Standard algorithm for computing linear combinations over prime fields
 

---

```

1: function LINEAR_COMBINATIONS_PRIME(r, h)
2:   int qf = q;
3:   if h==1 then qf = 2
4:   end if
5:   for (i = h; i <= k; i++) do
6:     for (int e = 1; e < qf; e++) do
7:       if e==1 then T[r] = add(T[r - 1], G[i]);
8:       else T[r] = add(T[r], G[i]);
9:       end if
10:      int w = get_weight(T[r]);
11:      A[w]++;
12:      if r<k then Linear_Combinations(r + 1, i + 1);
13:      end if
14:    end for
15:  end for
16: end function

```

---



---

**Algorithm 4** Standard algorithm - modification for  $\mathbb{F}_2$ 


---

```

1: function LINEAR_COMBINATIONS_BINARY(r, h)
2:   for (i = h; i<=k; i++) do
3:     T[r] = add(T[r - 1], G[i]);
4:     int w = get_weight(T[r]);
5:     A[w]++;
6:     if r<k then Linear_Combinations(r + 1, i + 1);
7:     end if
8:   end for
9: end function

```

---

element of the field. We use a primitive generator polynomial and therefore we can take  $\alpha = x$ . The set is precomputed using lookup tables before the function *Linear\_combinations* in Algorithm 2. The transition sequence (TS) of  $q$ -ary Gray code is used to show which matrix of the set has to be chosen for the current linear combination. In Algorithm 5, the parameter  $e$  traverses the transition sequence and thus the elements of the field in Gray code order as can also be seen in Table 1, showing the generation of a linear combination over  $\mathbb{F}_8$ . As can be seen, the vectors in the first column are representations of the polynomials that are multiplied with  $G[i]$  in the linear combination. The set of matrices is used to simulate the multiplication of a field element with the current row of the generator matrix. For clarity, in Table 1 we note the rows  $r$ ,  $r - 1$  and  $i$  of matrices  $T$  and  $G$  respectively as  $T[r]$ ,  $T[r - 1]$  and  $G[i]$ .

The correctness of Algorithms 3, 4, and 5 follows from the correctness of the basic Algorithm 2 since they are based on it. As can be seen from the presented high-level algorithms, the most expensive parts are the  $\theta$  additions and  $\theta$  weight computations. Therefore, the main optimizations in the next section are focused on these two operations.



**Algorithm 5** Extended algorithm for computing linear combinations over composite fields

---

```

1: function LINEAR_COMBINATIONS_COMPOSITE(r, h)
2:   int qf = q;
3:   if h==1 then qf = 2
4:   end if
5:   for (i = h; i<=k; i++) do
6:     for (int e = 1; e < qf; e++) do
7:       if e==1 then T[r] = add(T[r - 1], G[i]);
8:       else
9:         int j = TS[e] - 1;
10:        T[r] = add(T[r], xjG[i]);    // i-th row of matrix xjG
11:      end if
12:      int w = get_weight(T[r]);
13:      A[w]++;
14:      if r<k then Linear_Combinations_composite(r + 1, i + 1);
15:    end if
16:  end for
17: end for
18: end function

```

---

Table 1. Example of generation next linear combination over  $\mathbb{F}_8$ 

Gray code	TS	e	Generation	Linear combination
000	0	0	-	-
001	1	1	$T[r] = T[r - 1] + G[i]$	$T[r - 1] + G[i]$
011	2	2	$T[r] + xG[i] = T[r - 1] + G[i] + xG[i]$	$T[r - 1] + (x + 1)G[i]$
010	1	3	$T[r] + G[i] = T[r - 1] + (x + 1)G[i] + G[i]$	$T[r - 1] + xG[i]$
110	3	4	$T[r] + x^2G[i] = T[r - 1] + xG[i] + x^2G[i]$	$T[r - 1] + (x^2 + x)G[i]$
111	1	5	$T[r] + G[i] = T[r - 1] + (x^2 + x)G[i] + G[i]$	$T[r - 1] + (x^2 + x + 1)G[i]$
101	2	6	$T[r] + xG[i] = T[r - 1] + (x^2 + x + 1)G[i] + xG[i]$	$T[r - 1] + (x^2 + 1)G[i]$
100	1	7	$T[r] + G[i] = T[r - 1] + (x^2 + 1)G[i] + G[i]$	$T[r - 1] + x^2G[i]$

#### 4 LOW-LEVEL ALGORITHMS

In this section we present the implementation of the low-level algorithms for vector addition and calculating the weight of a vector (functions *add* and *get\_weight*). They have different implementations depending on  $q$ , the number of the elements in the considered field. At the heart of these algorithms is the appropriate representation of the elements of the field in a register data type. Depending on the length  $n$  of the vector we can supplement the register with zeros if necessary or use multiple registers if the vector cannot be written in a single one. For each implementation we firstly discuss the representation of the elements and after that the connected algorithms. Before proceeding with the low-level algorithms, we present some notations we use (for both 128 and 256-bit registers). These operations are defined to perform over registers, but for simplicity in the context of the algorithms we use them for vectors.

- Bitwise instructions ( $\wedge$ ,  $\vee$ ,  $\oplus$ ,  $\gg$ ,  $\ll$ ) operate on two registers, where the corresponding bitwise operation (*AND*, *OR*, *XOR*, *SHIFT RIGHT*, *SHIFT LEFT*) is executed on all bits at the same time regardless of the data type that is stored into the register.

- Right shift function ( $\text{srl}(a, n)$ ) that shifts register  $a$  by  $n$  bytes, and positions that have been vacated by the shift operation are filled with zeros, regardless of the data type that is stored into the register.
- Simple add and subtract functions (+, -) that execute the corresponding operation on the 8-bit packed integer elements of two registers componentwise.
- Comparison functions ( $\text{cmplt}(a,b)$ ,  $\text{cmpgt}(a,b)$ ,  $\text{cmpeq}(a,b)$ ) that compare 8-bit packed integer elements of two registers using the corresponding operation (<, >, =) and store 0xFF if the comparison is true and 0 otherwise in the resulting vector.
- Operation for blending two registers ( $\text{blendv}(a, b, \text{mask})$ ) where the resulting register stores an element of  $a$  if the corresponding element of  $\text{mask}$  is negative and an element of  $b$  otherwise.
- The notation  $\_\text{m128i}$  is used to represent a register with length 128 bits containing integer data. The presented low-level algorithms can be executed on 128-, 256- and 512-bit register. For simplicity, the algorithms are presented for smaller code length (e.g.  $n < 16$ ) that can be saved in a single 128-bit register. In practice, the algorithms are implemented using multiple registers for larger lengths and with different register lengths.

*Remark:* The description of the algorithms uses the generally accepted notations  $\text{FF}$  to represent one byte equal to '11111111'.

Therefore, in the rest of the paper we use the notation  $\text{popcnt}$  for function calculating the number of nonzero bits in a computer word.

#### 4.1 The fields $\mathbb{F}_2$ and $\mathbb{F}_4$

The bit representation over  $\mathbb{F}_2$  is natural - each coordinate of a binary vector is stored in one bit of the computer word. Hence, for a codeword in a code with length  $n$  we need  $\lfloor ((n-1)/64) + 1 \rfloor$  computer words. For the register representations (128-bit registers)  $\lfloor ((n-1)/128) + 1 \rfloor$  registers are needed. We store each codeword in a register.

This implementation is not very effective for  $n \leq 64$  since half of the register will not be used. An additional modification of Algorithm 4 is implemented for these specific cases. To store the codewords of a code of length  $n \leq 64$ , we use its  $[n, k-1]$  subcode  $C'$  with a generator matrix  $G'$  obtained from  $G$  by removing the last row  $G[k]$ . We also need the coset  $G[k] + C' = \{G[k] + c \mid c \in C'\}$ . Thus, the original code  $C$  is the set  $C' \cup (G[k] + C')$ . In the implementation, we write the row  $G[k]$  in the second 64 bits of the zeroth row of the matrix  $T$  (the first 64 bits are zeros). Moreover, we store a row of the matrix  $G'$  in a register twice (one copy in the first 64 bits and one more in the second 64 bits). So with each addition of a row of matrix  $G'$  we obtain a codeword form  $C'$  and its coset at the same time. The addition of vectors is also natural - the bitwise operation  $\text{XOR}$  is used.

To calculate the weight of a vector, the  $\text{popcnt}$  function has to be executed on the two halves of a register that are saved in two computer words. If  $n > 64$ , we need to sum the weights of the two computer words. If  $n \leq 64$ , the  $\text{popcnt}$  function applied to these two computer words gives the weight of two vectors in  $C$ .

Any element of the composite field  $\mathbb{F}_4$  is stored in two bits containing the coefficients of the polynomials. Thus, we need to double the memory space to represent a vector over the field. For example, the vector  $y = (y_1, \dots, y_n) \in \mathbb{F}_4^n$  is represented as  $(v_0, v_1)$ , where  $y_i = a_i + b_i x \in \mathbb{F}_4$ ,  $i = 1, \dots, n$ , and  $v_0 = (a_1, a_2, \dots, a_n)$ ,  $v_1 = (b_1, b_2, \dots, b_n) \in \mathbb{F}_2^n$ . This representation allows easy implementation of the low-level functions. The addition of vectors is implemented only with an  $\text{XOR}$  operation on the registers. The calculation of the weight of a vector over  $\mathbb{F}_4$  is executed in two steps - an  $\text{OR}$  operation on the two halves of the register and execution of the  $\text{popcnt}$  function over the result. Table 2 shows the representation of a vector in 128-bit register and the implementation of the  $\text{get\_weight}$  function for the three main cases. The first column shows the field, the second

Table 2. Example of bitwise representation in a register and *get\_weight* function ( $n \leq 128$ )

Field	128-bit register	<i>get_weight</i> (reg)
$\mathbb{F}_2$ $n < 64$	reg = $(v_1, v_2)$ $v_1 \in C', v_2 \in G[k] + C'$	$w_1 = \text{popcnt}(v_1)$ $w_2 = \text{popcnt}(v_2)$
$\mathbb{F}_2$ $n \geq 64$	reg = $(v)$ $v = (v_1, v_2), v \in \mathbb{F}_2^n$	$\text{popcnt}(v_1) + \text{popcnt}(v_2)$
$\mathbb{F}_4$	reg = $(v_0, v_1)$ $v \in \mathbb{F}_4^n, v_0, v_1 \in \mathbb{F}_2^n$	$\text{popcnt}(v_0   v_1)$

column gives the vector representation in a register and the third - the implementation of the *get\_weight* function.

#### 4.2 The fields $\mathbb{F}_3, \mathbb{F}_9$ and $\mathbb{F}_{27}$

The main optimizations for the fields with characteristic 3 are based on a mapping  $\Pi : \mathbb{F}_3 \rightarrow \mathbb{F}_2^2$  defined by

$$\Pi(0) = (1, 1)$$

$$\Pi(1) = (1, 0)$$

$$\Pi(2) = (0, 1)$$

This allows us to use bitwise operations and representations of the elements. Let us first consider the field with three elements. There are other representations for  $\mathbb{F}_3$  as seen in [6], [24]. The mapping  $\Pi$  is introduced in [10] and gives easy implementation of basic operation over elements of the field. We extend this representation to the vectors over  $\mathbb{F}_3$  in the following way: we introduce the mapping  $\pi : \mathbb{F}_3^n \rightarrow \mathbb{F}_2^{2n}$  as

$$\pi(v) = (\alpha_1, \alpha_2, \dots, \alpha_n, \beta_1, \beta_2, \dots, \beta_n),$$

where  $v = (v_1, v_2, \dots, v_n) \in \mathbb{F}_3^n$ ,  $\Pi(v_i) = (\alpha_i, \beta_i)$ . We consider also the vector

$$\overline{\pi(v)} = (\beta_1, \beta_2, \dots, \beta_n, \alpha_1, \alpha_2, \dots, \alpha_n).$$

Using  $\pi$  and  $\overline{\pi(v)}$ , we can implement Algorithm 6 for addition of vectors over  $\mathbb{F}_3$ .

---

#### Algorithm 6 Algorithm for vector addition over the field $\mathbb{F}_3$

---

```

1: function ADDF3(a, b)
2:    $t, u, r \in \mathbb{F}_2^{2n};$ 
3:    $t = \pi(a) \oplus \pi(b);$ 
4:    $u = t \oplus \overline{\pi(b)};$ 
5:    $r = \bar{t} \vee u;$ 
6: return r
7: end function
```

---

This representation also allows to use *popcnt* function in the implementation of *get\_weight*. As can be seen in the mapping  $\Pi$  the zero element of the field is represented by (1, 1). Therefore, the weight of a vector  $a$  can be calculated by applying XOR operation between the  $\alpha$  and  $\beta$  parts of  $\pi(v)$ . Algorithm 7 shows the calculation of the weight of a vector over  $\mathbb{F}_3$ .

The correctness of Algorithm 6 follows from the chosen element representation described in [10]. Representing the vectors in  $\mathbb{F}_2^{2n}$  also allows for the computations to be executed in 5 steps. In our specific case, the vectors are rows of the generator matrix  $G$  and temporary matrix  $T$ . Since  $G$  is

---

**Algorithm 7** Algorithm for calculating the weight of a vector over  $\mathbb{F}_3$ 


---

```

1: function GET_WEIGHTF3( $\pi(v)$ )
2:    $a, b \in \mathbb{F}_2^n$ ;
3:    $a = (\alpha_1, \dots, \alpha_n)$ ;
4:    $b = (\beta_1, \dots, \beta_n)$ ;
5:    $t = a \oplus b$ ;
6:    $\text{int } r = \text{popcnt}(t)$ ;
7:   return  $r$ 
8: end function

```

---

saved in global 2-dimensional array, we can precompute  $\overline{\pi(G[i])}$  for every row  $G[i]$  of  $G$ . This will result in execution of vector addition in only 4 steps at the expense of greater memory complexity. The elements of the composite fields  $\mathbb{F}_9$  and  $\mathbb{F}_{27}$  are represented by vectors in  $\mathbb{F}_3^2$  and  $\mathbb{F}_3^3$  respectively. Thus, the computations can be executed by the presented algorithms for  $\mathbb{F}_3$ . Vector addition is performed by the same algorithm for larger vector length. The calculation of the weight executed two or three *OR* operations (for  $\mathbb{F}_9$  and  $\mathbb{F}_{27}$  respectively) given the right data representation.

#### 4.3 The fields $\mathbb{F}_p$ and $\mathbb{F}_{p^m}$

As previously said, the elements of the prime field with  $p$  elements can be represented by the integers in the interval  $[0, p - 1]$ . Therefore, in the cases when  $p \neq 2, 3$  we use 8-bit packed integers. The vector addition over  $\mathbb{F}_p$  is typically represented by a modulo operation that takes more processing time. Thus, our implementation aims to execute vector addition with a set of different instruction. The first method, shown in Algorithm 8, executes three instructions - addition of two vectors, subtraction of the vector  $P = (p, p, \dots, p)$  with length  $n$  from the result of the previous operation and blending the two obtained vectors. The last operation is implemented by the special instruction *blendv*( $a, b, m$ ) of the extended instruction set. This function writes a coordinate from either  $a$  or  $b$  based on the corresponding coordinate in the mask vector  $m$ . Algorithm 9 is more general and it uses multiple comparison and bitwise operations. It is appropriate to be used in cases where the *blendv* function is not implemented in the architecture (e.g. NEON instruction set).

The elements of the composite fields  $\mathbb{F}_{p^2}$  for  $p = 5$  and  $7$  can be represented as  $a + bx$ , where  $a, b \in \mathbb{F}_p$ . Then the vector  $v = (v_1, \dots, v_n) \in \mathbb{F}_{p^2}^n$  with  $v_i = a_i + b_i x$ ,  $a_i, b_i \in \mathbb{F}_p$ ,  $i = 1, \dots, n$ , can be mapped to the vector  $v' = (A, B) \in \mathbb{F}_p^{2n}$  where  $A = (a_1, \dots, a_n) \in \mathbb{F}_p^n$  and  $B = (b_1, \dots, b_n) \in \mathbb{F}_p^n$ . So, the implementation of the addition of vectors over the composite field is based on vector addition over a prime field.

---

**Algorithm 8** Algorithm for vector addition over  $\mathbb{F}_p$  with *blendv*


---

```

1: function ADD_FP_BLENDV( $u, v \in \mathbb{F}_p^n$ )
2:    $\_\_m128i$   $r\_add, r\_sub, r\_blendv, P = (p, p, \dots, p)$ ;
3:    $r\_add = u + v$ ;
4:    $r\_sub = r\_add - P$ ;
5:    $r\_blendv = \text{blendv}(r\_add, r\_sub, r\_sub)$ ;    //  $r\_blendv_i = \begin{cases} r\_add_i & \text{if } r\_sub_i < 0 \\ r\_sub_i & \text{if } r\_sub_i \geq 0 \end{cases}$ 
6:   return  $r\_blendv$ ;
7: end function

```

---

**Algorithm 9** Algorithm for vector addition over  $\mathbb{F}_p$ 


---

```

1: function ADD_FP( $u, v \in \mathbb{F}_p^n$ )
2:   Register variables:  $r\_add, r\_sub, m\_cmplt, m\_cmpgeq, P = (p, p, \dots, p)$ ;
3:    $r\_add = u + v$ ;
4:    $r\_sub = r\_add - P$ ;           //  $r\_sub_i = r\_add_i - p$ 
5:    $m\_cmpgeq\_ = cmpeq(r\_add, P)$ ;           //  $m\_cmpgeq_i = \begin{cases} FF & \text{if } r\_add_i \geq p \\ 0 & \text{otherwise} \end{cases}$ 
6:    $m\_cmplt\_ = cmplt(r\_sub, 0)$ ;           //  $m\_cmplt_i = \begin{cases} FF & \text{if } r\_sub_i < 0 \\ 0 & \text{otherwise} \end{cases}$ 
7:    $r\_add = bitwise\_and(r\_add, m\_cmplt)$ ;           //  $(r\_add \wedge m\_cmplt)$ 
8:    $r\_sub = bitwise\_and(r\_sub, m\_cmpgeq)$ ;           //  $(r\_sub \wedge m\_cmpgeq)$ 
9:   return  $bitwise\_or(r\_add, r\_sub)$            //  $r\_add \vee r\_sub$ ;
10: end function

```

---

There are two main approaches for the calculation of the weight of a vector. For simplicity, in the presented algorithms we assume that  $n \leq 16$  since a single 128-bit register ( $n \leq 32$  for 256-bit register,  $n \leq 64$  for 512-bit register) contains up to 16 elements of the desired type. If the length of the code is less than 16, we take  $n = 16$  and extend all codewords with  $16 - n$  zeros. If  $n > 16$  ( $n > 32, n > 64$ ) we simply execute the algorithms over multiple registers. The first algorithm marks the positions of the register, containing nonzero elements and afterwards calculates the weight of the given vector as shown in Algorithm 10.

The second approach uses the number of zero elements  $w_0$  to calculate the weight. It constructs a computer word that has  $w_0$  nonzero bits. This is accomplished using a comparison with zero register, a mask  $h$  and bitwise operations. Algorithm 11 shows the implementation of the second version in the case with 128-bit registers and lengths of the code  $\leq 16$ . The implementation for 256-bit registers extends  $h$  to the vector (8, 8, 8, 8, 8, 8, 8, 8, 4, 4, 4, 4, 4, 4, 4, 4, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1) and executes two more operations - a right shift of 128 bits and another OR operation. The main advantage of Algorithm 11 is the minimization of the use of *popcnt* instruction since it needs to extract the 64-bit parts of the register into a computer words, which is a relatively slow process. The 512-bit implementation does not need a mask register  $h$ , since the comparison functions save the result in a single computer word. Therefore, we only need a single *popcnt* instruction to calculate  $w_0$ .

**Algorithm 10** Algorithm for calculating the weight of a vector over  $\mathbb{F}_p$  (version 1)

---

```

1: function GET_WEIGHTFP_VERSION1( $v \in \mathbb{F}_p^n$ )
2:    $\_\_m128i$   $r = (0, 0, \dots, 0)$ 
3:    $r = cmpgt(v, 0)$ ;           //  $r_i = \begin{cases} FF & \text{if } v_i > 0 \\ 0 & \text{if } v_i \leq 0 \end{cases}$ 
4:   return  $(popcnt(r_1, \dots, r_{n/2}) + popcnt(r_{(n/2)+1}, \dots, r_n)) >> 3$ ;           //  $(>> 3)$  - fast division by 8
5: end function

```

---

The finite fields  $\mathbb{F}_{2^m}^n$  with  $m > 2$  are also represented using 8-bit integers. In this case the addition is executed by a single XOR operation. Due to the representation of the elements the *get\_weight* function can be implemented by both Algorithms 10 and 11.

**Algorithm 11** Algorithm for calculating the weight of a vector over  $\mathbb{F}_p$  (version 2)

---

```

1: function GET_WEIGHTFP_VERSION2( $v \in \mathbb{F}_p^n$ )
2:    $\_\_m128i$   $r1, r2, r, m\_cmpl$ ,    $h = (2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1)$ ;
3:    $m\_cmpeq = cmpeq(v, 0)$ ;           $// m\_cmpeq_i = \begin{cases} FF & \text{if } v_i = 0 \\ 0 & \text{if } v_i \neq 0 \end{cases}$ 
4:    $r1 = m\_cmpeq \wedge h$ ;
5:    $r2 = srli(r1, 8)$ ;                 $// \text{right shift of the bits in } r1 \text{ with 8 bytes}$ 
6:    $r = r1 \vee r2$ ;
7:    $w_0 = popcnt(r_{(n/2)+1}, \dots, r_n)$ ;
8: return  $16 - w_0$ ;
9: end function

```

---

**5 EXPERIMENTAL RESULTS**

In this section, we present some experimental results on the efficiency of the implemented algorithms. The main goals of the presented experimental results are the following:

- comparison between the execution times of the function for computing the weight distribution of linear codes with the presented library for different instruction sets with x86 architecture;
- comparison between the execution times of the presented library with different architectures (x86 and ARM);
- comparison between the execution times of the function for computing the weight distribution of linear codes with the presented library and with other software packages such as Magma and GAP.
- evaluation of optimal instruction set to be used for given parameters.

The experimental results for x86 architectures are executed on Windows operating system with Intel Core I5 1035G1 CPU @1.00 GHz and gcc 8.1 compiler. The computations using Guava package for GAP have been executed using the same platform. The experimental results with NEON instruction set and ARM architecture have been executed on Apple M1 chipset @3.2 GHz and clang compiler version 15.0.0. The experimental results with Magma software package have been executed on the online calculator running on a virtual machine with processor Intel Xeon Processor E3-1220 @3.10GHz. The library has also been tested (compilation, correctness, installation) on the platforms presented in Table 3.

Table 3. Tested platforms

CPU	OS	Instruction set	Compiler	Build system or IDE
Intel Xeon Gold 5118 @2.36GHz	Red Hat Enterprise 7.9	AVX512 F/BW	gcc 9.2	Makefile
Intel Core i5-13600K @3.5GHz	Windows 11 Pro	AVX2	gcc 10.3 msvc 19.37.32724	CodeBlocks 20.03 VS Community 17
Intel Core i5-1035G @1.0GHz	Windows 10 Pro	AVX512 F/BW/ VPOPCNTDQ	gcc 8.1 msvc 19.29.30146	CodeBlocs 20.03 VS Community 16
Intel Xeon E5-2640 @2.5GHz	Ubuntu 18.04	SSE4.2	gcc 7.5	Makefile CodeBlocks 20.03
AMD Ryzen 5 7600 @3.8GHz	Windows 11 Home	AVX512 F/BW/ VPOPCNTDQ	gcc 8.1 msvc 19.39.33523	CodeBlocks 20.03 VS Community 17
Apple M1 @3.2 GHz	macOS Sonoma 14.1.2	NEON	clang 15.0.0	Xcode 15.0.2

Table 4. Execution times (sec.) for different software packages

n	k	q	Magma	GAP	SSE4.1	AVX2	AVX512	Neon	Scalar
60	29	2	1.780	6.832	1.431	1.252	1.314	1.125	60.936
500	29	2	12.910	24.067	5.920	7.311	3.145	3.832	430.856
60	16	4	14.190	825.245	5.998	7.389	6.892	5.381	173.034
500	16	4	56.450	6999.176	31.981	28.995	10.968	12.184	1141.309
60	19	3	4.880	311.815	2.871	3.224	2.357	2.376	67.744
500	19	3	25.600	2589.720	9.285	10.364	5.774	4.799	461.789
60	9	9	3.260	68.989	0.271	0.236	0.209	0.240	5.156
500	9	9	24.830	492.667	1.094	0.977	0.544	0.560	38.973
60	6	27	2.000	66.120	0.093	0.093	0.106	0.076	1.710
500	6	27	14.700	501.733	0.389	0.339	0.296	0.222	11.736
60	13	5	4.540	242.880	4.356	3.378	2.888	2.925	35.781
500	13	5	20.830	1989.983	20.568	14.979	7.367	10.612	241.332
60	6	25	1.320	41.891	0.166	0.141	0.093	0.106	1.105
500	6	25	10.060	321.037	0.921	0.541	0.331	0.593	8.093

To analyse the efficiency of the vectorization, we use the concept of *vectorization factor* [1]. The vectorization is defined as the maximum number of coordinates of a vector that can be stored in a computer word or an extended register. In general, the vectorization factor depends on the representations, described in Section 4, and the length of the used register. The experimental results are presented in Tables 4 and 5. They show average execution times for functions that calculate the weight distribution for a given code. Examples of codes over different finite fields have been selected, which include all the different options for representing the elements of the respective field. In Table 4, the first three columns give the parameters  $n$ ,  $k$ , and  $q$ . The next two columns present the execution times for Magma and GAP software packages, and the following three columns show the execution times for SSE4.1, AVX2 and AVX512 implementation, respectively. The last two columns give the average execution times for implementation with NEON instruction set and the scalar implementation on x86 architecture using Algorithm 2, respectively. Table 4 is used for the analysis of the efficiency with different architecture and comparison to different software packages. The presented code lengths and dimensions are chosen due to some limitation of Magma and GAP - the Magma online calculator limits computation time and the GAP computations take extensive amount of time for larger parameters.

Introducing more experiments allows for better analysis of the effectiveness of different instruction sets. Table 5 gives additional experimental results for x86 architectures with more values of  $n$ , considering codes with fixed dimension  $k$  for each used field.

The values of  $n$ ,  $k$  and  $q$  are given in the first three columns, respectively. In the following three columns, we give the execution times for the SSE4.1, AVX2 and AVX512 instruction sets, respectively. Table 5 is used for more in depth analysis for different code lengths and dimensions on x86 architectures. Figures 1 and 3 visualize the experimental results for fixed finite fields and different lengths and dimensions. They present the execution times in seconds for the finite fields  $\mathbb{F}_2$ ,  $\mathbb{F}_3$ ,  $\mathbb{F}_5$ ,  $\mathbb{F}_{27}$  using AVX512 instruction set (fig. 1) and NEON instruction set (fig. 3). The execution times are shown for larger set of lengths and for different dimensions.

Table 5. Execution times (sec.) on x86 architecture

n	q	k	SSE4.1	AVX2	AVX512
100	2	30	5,87	4,79	3,80
500	2	30	13,69	15,53	6,65
1000	2	30	21,86	26,90	8,89
2000	2	30	39,23	50,59	12,46
100	4	15	1,88	2,10	1,77
500	4	15	8,41	8,74	3,25
1000	4	15	14,23	15,08	4,03
2000	4	15	28,60	24,46	5,41
100	3	20	20,64	13,76	8,21
500	3	20	33,44	37,58	19,07
1000	3	20	52,65	62,39	23,85
2000	3	20	83,31	105,62	30,31
100	9	10	4,93	2,75	2,53
500	9	10	9,78	8,81	4,86
1000	9	10	15,37	15,52	6,02
2000	9	10	27,57	26,22	8,77
100	27	7	4,85	3,80	5,91
500	27	7	12,38	8,77	8,83
1000	27	7	18,79	15,26	11,61
2000	27	7	34,78	28,70	16,92
100	5	12	1,20	1,03	0,75
500	5	12	4,70	3,02	1,81
1000	5	12	8,68	5,96	3,12
2000	5	12	16,17	11,27	5,96
100	25	7	6,73	4,29	3,53
500	25	7	27,17	13,07	9,34
1000	25	7	48,94	25,32	17,51
2000	25	7	92,16	48,71	33,27

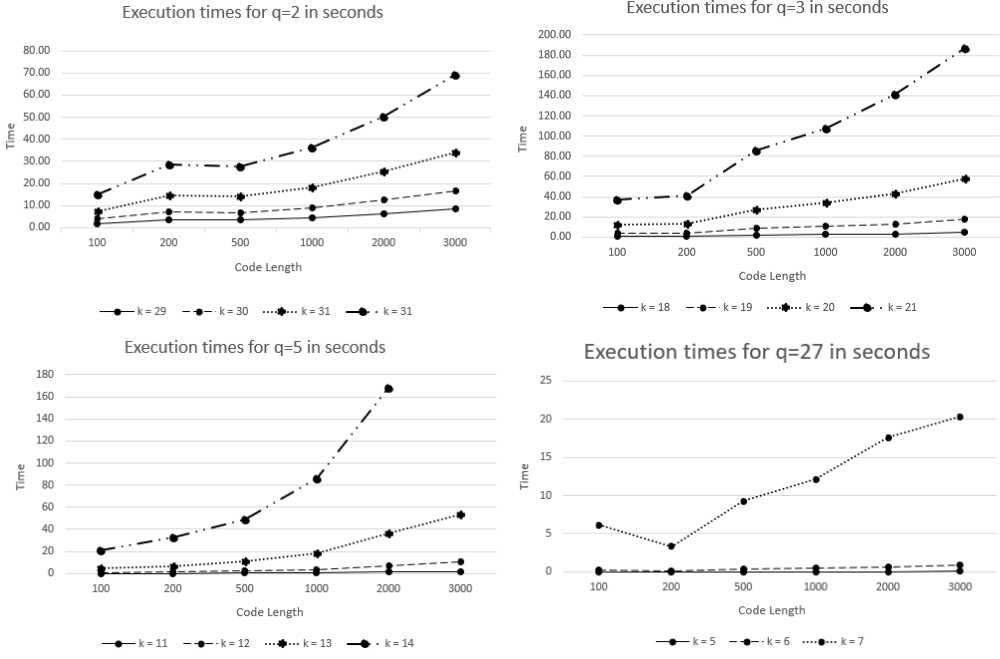
### 5.1 Analysis of experimental results with high vectorization factor

Let us first consider the cases with high vectorization factor (and bitwise representation of the elements of the considered field) and the presented experimental results in Table 5. Such are the finite fields  $\mathbb{F}_2, \mathbb{F}_3, \mathbb{F}_4, \mathbb{F}_9, \mathbb{F}_{27}$ . For example, in the case of  $\mathbb{F}_2$  and a 128-bit register we have vectorization factor of 128, while for  $q \neq 2$  in the presented finite fields and 128 bit register we have vectorization factor between 21 and 64. However, in cases where  $q \neq 2$ , multiple instructions are used to perform the operations of vector addition and vector weight computation, as seen in Section 4. The scalar implementation (using directly Algorithm 2 with lookup tables) is included for completeness and testing for correctness. The high vectorization factor explains the achieved speedup.

In the cases with prime fields ( $\mathbb{F}_2, \mathbb{F}_3$ ), the implementation with 256-bit registers is between 10% and 20% slower for  $n > 100$ . Code profiling with Visual Studio Profiling Tools shows that more than 75% of the computational time is spent on calculating the weight of the vectors. Due to the representation of the elements of the field and the available instructions, multiple *popcnt*



Fig. 1. Graphical representation of execution times with AVX512

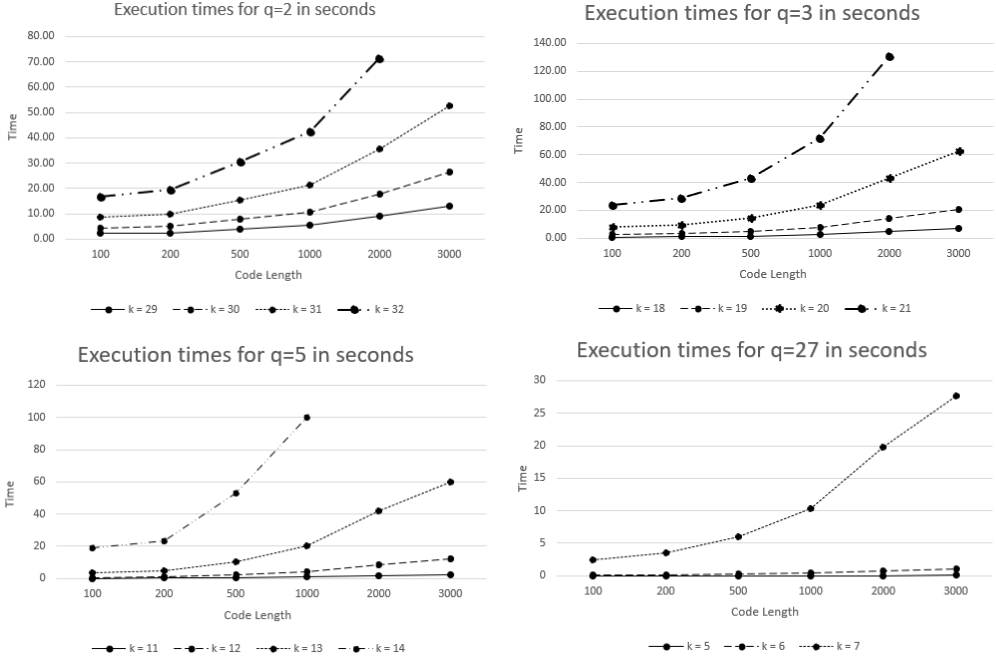


functions are executed in those cases. Data also is transferred between large and small (64 bits) registers, which is a slow process. Further inspection of the generated assembly files shows that in the functions to calculate the weight of a vector, the data is transferred from 256-bit registers into 128-bit ones and afterwards to a 64-bit register. This also increases the overhead. In the case of  $n = 100$ , the computations are executed using a single register and using cosets for  $q = 2$ , thus AVX2 performs better with decrease of execution between 18% and 33% compared to SSE4.1. In the cases with composite fields, fewer *popcnt* functions are executed, thus decreasing the time spent for calculation of the weight of the vector to approximately 60%. However, this function is still heavy and the other limitations still stand. Therefore, the execution time with AVX2 is between 10% and 30% faster compared to SSE4.1.

Let us analyse the experimental results with AVX512, and compare them with the results with SSE4.1 and AVX2. For smaller lengths, where the computations don't use the full register ( $q = 2, 4$  and  $n = 100$ ), the speedup is less than 2 times. An exception is the case when  $q = 27$  where multiple heavy functions are executed (register permutation and a *popcnt* function), resulting in slower execution times than the SSE4.1 version. When the full register (or even more than half of the register) is used, as seen for  $n > 500$  in both Table 5 and Figure 1, we observe faster execution times by factor of up to 5 compared to the SSE4.1 version. The better performance for larger lengths is even more pronounced with the increase of  $k$ .

The experimental results with ARM architecture and NEON registers show execution times close to those with AVX512 on x86 architecture. We observe difference in execution times by no more than 30. Figure 3 shows that the execution time increases for all presented finite fields as  $n$  increases, unlike the AVX512 implementation.

Fig. 3. Graphical representation of execution times with NEON



Let us compare the execution times of the presented library with the implementation using AVX512 and NEON instruction sets to other widely used software packages that have a function for computing the weight distribution of a linear code. The Guava package for GAP is an open source software and is also included in the mathematical software SAGE. According to its documentation, optimization for the target functionality is only implemented for the field  $\mathbb{F}_2$ . This is also apparent from the presented experimental results in Table 4. In this case, we get faster execution time by factors between 5 and 7 for x86 architectures. The achieved speedup is increased with higher vectorization factor (512-bit registers) and larger lengths. Comparing to implementation on ARM architectures, we observe approximately 6 times faster computations. For codes over fields with  $q > 2$  elements, we observe more than 100 times faster execution times both x86 and ARM architectures.

Let us consider a comparison with the Magma software. Magma uses bitwise representation of the elements of the fields for  $q \leq 5$  [31]. Comparing the execution times for the implementations using AVX512, we observe faster execution times with approximately 35% for codes over  $\mathbb{F}_2$  and 50% for  $\mathbb{F}_5$  of length 60. However, for lengths  $> 500$  a speedup between 2 and 4 times can be seen for  $\mathbb{F}_5$  and  $\mathbb{F}_2$ , respectively. For the other finite fields considered here ( $\mathbb{F}_3, \mathbb{F}_4, \mathbb{F}_9, \mathbb{F}_{27}$ ), we obtain faster execution times by factors between 2 and 49, depending on the field (prime or composite) and the length of the code.

## 5.2 Analysis of experimental results with lower vectorization factor

In the case of an odd prime  $p > 3$ , the finite fields  $\mathbb{F}_p$  and  $\mathbb{F}_{p^m}$  have lower vectorization factor (byte representation of the elements of the field) in the library. For example, one element of the prime field  $\mathbb{F}_5$  is written in one byte and therefore the vectorization factor for 128-bit registers is

16. The chosen standard data type for the representation allows for the use of standard operations such as addition and subtraction without overflow. Comparing SSE4.1 with AVX2 based on the experimental results in Table 5, we observe faster execution times by factor of approximately 1.4 for prime fields and 1.8 for composite fields. Comparing to the AVX512 version based on the experimental results in Table 5, we observe speedup of up to 2.8 for both prime and composite fields. As can be seen from Figure 1, in the case  $q = 5$  the execution time increases even for smaller lengths, since a full register is used.

Comparing the AVX512 version to the NEON version of the library, we observe similar behaviour in the execution times for different  $k$  and  $n$ . The experimental results in Table 4 show that for smaller lengths, the execution times are comparable, while for larger lengths the implementation with AVX512 is with approximately 40% faster for prime fields and with up to 80% faster for composite fields.

Let us compare the presented library using AVX512 instruction set to Magma. In the case of composite fields, we observe faster execution times between 14 and 30 times. Comparing to the Guava package for GAP we observe more than 80 times faster execution times depending on the field and the code length.

### 5.3 Some remarks on the optimization techniques

Here we describe the main optimization techniques that are used. We consider two types of techniques based on their impact on the execution time. Some are only applicable for the implementations in composite fields (see Section 3.3). First, we consider those optimizations that decrease the execution time more than two times. The high level optimization consists of two main concepts - (1) generating only nonproportional codewords, and (2) using only vector addition operation in the main computations. The algorithm, presented in [16], generates all  $q^k$  codewords of a linear code over  $\mathbb{F}_q$ . In our library, multiplication with scalar is only used in the precomputational part of the algorithms for composite fields and therefore, its use is limited (total number of multiplications of a vector by scalar is  $mk$ ). Considering the low level algorithms, the main optimization technique is the vectorization, characterized by the vectorization factor. The bitwise representation is fundamental for this optimization technique. For the fields  $\mathbb{F}_2$  and  $\mathbb{F}_4$  the bitwise implementation is natural. For the other bitwise implementations (in the fields  $\mathbb{F}_{3^m}$ ) the representation of the data in the memory is essential. The use of extended vector registers allows the addition of two vectors to be performed in only 5 operations. For  $\mathbb{F}_{p^m}$ ,  $p > 3$ , and  $\mathbb{F}_{2^m}$ ,  $m > 2$ , the addition is executed without the use of the modulo operation. Multiple operations are used to substitute it, with the heavies one being the *blendv* (see Section 4).

The rest of the optimizations result in speedup of less than two times. Such techniques are the modifications of the high level algorithms for composite fields. For the prime fields  $\mathbb{F}_p$ , where  $p > 3$ , we present two algorithms for computation of the weight of a vector using *popcnt* instruction. Algorithm 11 reduces the number of used *popcnt* instruction. This optimization is only applicable for instruction sets SSE4.1, AVX2 and NEON.

In order to choose an appropriate set of instructions for x86 depending on the parameters of the studied codes, what matters is how large the portion of the extended register actually used is. This is different from the vectorization factor, as it is possible for some values of  $n$  to use only part of the register. Experimental results show that it matters whether at least half of the register is used to store vector coordinates. We consider the following cases:

- In the cases with bitwise representation and high vectorization factor ( $\mathbb{F}_2, \mathbb{F}_4, \mathbb{F}_3, \mathbb{F}_9, \mathbb{F}_{27}$ ), the 256-bit register implementation performs comparable to the 128-bit version. Therefore, the SSE4.1 implementation should be preferred for smaller lengths.

- For  $\mathbb{F}_2$ , a coordinate of a vector is stored in a single bit, while in the case of  $\mathbb{F}_4$  it is stored in two bits. Thus, for  $n \leq 64$ , a 128-bit register is appropriate, since less than half of the 512-bit register is used.
- For finite fields  $\mathbb{F}_{3^m}$ ,  $m \geq 1$ , an element of the field is stored in  $2m$  bits. Thus, for  $n \leq 64$ , 128-bit registers are more suitable to be used. An exception is the case of codes over  $\mathbb{F}_{27}$  where 128-bit register is more appropriate for  $n \leq 128$ .

## 6 CONCLUSION

In this paper, we presented a library developed for computing some weight characteristics of linear codes over finite fields with  $q$  elements, where  $q \leq 64$ . An extensive description of the compilation process, the available end user functions and a simple console application example with a testing functionalities are given in a user's manual. The presented algorithms in Section 3 give the following optimizations: to compute the weight distribution, we generate only the nonproportional codewords of a linear code; to generate the codewords, we use only vector addition. The optimizations related to the vectorization, are more clearly visible in the presented low-level algorithms. The presented experimental results in Section 5 show that even smaller registers result in a significant speedup compared to a non vectorized implementation of Algorithm 2. The effectiveness of the vectorization is best seen in the implementation with AVX512 and NEON instruction sets. As can be seen from Section 5, the developed library is faster in all presented cases than the best known open source library in the same direction - GAP.

The library is intended to be used in addition to other software for classification of linear codes over different finite fields and this is its main advantage over software packages such as Magma and GAP. Furthermore, it can be integrated in other parallel software developed with MPI or OpenMP and with different computer architectures for high-performance computing as Avitohol [25]. On the other hand, the design of the high-level algorithms can be modified in order to compute the number of linear combinations of up to fixed number of vectors. This is most expensive part in algorithms for calculation of the minimum distance such as the Brower-Zimmerman algorithm (See for example [9, 15] and parallel implementations for the binary case [19] and [29]). The main functionality of the library, however, is focused on the calculation of the weight distribution of linear codes and not its minimal distance. Thus, the current implementation of the function for calculation of minimal distance is not optimized and its efficiency may not be comparable to specialized algorithms and techniques in the general case. The library works for both x86 architectures with all currently available register lengths and ARM architectures having the NEON instruction set. It is important to note that there is evidence that the use of larger registers in multithreaded programs most likely will result in a lowering of the working clock frequency [30].

There are several ways in which the presented library can be extended. One approach is to integrate methods for computing the minimum distance of a linear code using modifications based on the Brower-Zimmerman algorithm. It can also be expanded with functions for calculating other characteristics of linear codes.

## ACKNOWLEDGMENTS

We are greatly indebted to the anonymous referees for their valuable and helpful suggestions and comments.

The research of the first author is partially supported by the Bulgarian National Science Fund under Contract No KP-06-H62/2/13.12.2022. The work of the second author was partially supported by the Bulgarian Ministry of Education and Science, grant no. D01-325/01.12.2023 for NCHDC, a part of the Bulgarian National Roadmap on RIs.

We acknowledge the provided access to the e-infrastructure of the Centre for Advanced Computing

and Data Processing, with the financial support by the Grant No BG05M2OP001-1.001-0003, financed by the Science and Education for Smart Growth Operational Program (2014-2020) and co-financed by the European Union through the European structural and Investment funds.

## REFERENCES

- [1] AMIRI, H., AND SHAHBAHRAMI, A. Simd programming using intel vector extensions. *Journal of Parallel and Distributed Computing* 135 (2020), 83–100.
- [2] ARM. Arm architecture instruction sets guide, 2023.
- [3] BARG, A., AND DUMER, I. On computing the weight spectrum of cyclic codes. *IEEE Transactions on Information Theory* 38, 4 (1992), 1382–1386.
- [4] BERLEKAMP, E., McELIECE, R., AND VAN TILBORG, H. On the inherent intractability of certain coding problems (corresp.). *IEEE Transactions on Information Theory* 24, 3 (1978), 384–386.
- [5] BOSMA, W., CANNON, J., AND PLAYOUST, C. The magma algebra system i: The user language. *Journal of Symbolic Computation* 24, 3-4 (1997), 235–265.
- [6] BOUYUKLIEV, I., AND BAKOEV, V. Efficient computing of some vector operations over  $gf(3)$  and  $gf(4)$ . *Serdica Journal of Computing* 2, 2 (2008), 137–144.
- [7] BOUYUKLIEV, I., AND BAKOEV, V. A method for efficiently computing the number of codewords of fixed weights in linear codes. *Discrete applied mathematics* 156, 15 (2008), 2986–3004.
- [8] BOUYUKLIEV, I., BOUYUKLIEVA, S., MARUTA, T., AND PIPERKOV, P. Characteristic vector and weight distribution of a linear code. *Cryptography and Communications* 13 (2021), 263–282.
- [9] BOUYUKLIEVA, S., AND BOUYUKLIEV, I. An extension of the brouwer–zimmermann algorithm for calculating the minimum weight of a linear code. *Mathematics* 9, 19 (2021).
- [10] COOLSAET, K. Fast vector arithmetic over  $f_3$ . *Bulletin of the Belgian Mathematical Society-Simon Stevin* 20, 2 (2013), 329–344.
- [11] CRAWWINCKEL, J., ROIJACKERS, E., BAART, R., MINKES, E., RUSCIO, L., MILLER, R., BOOTHBY, T., TJHAI, C., AND JOYNER, D. Gap package guava.
- [12] DODUNEKOVA, R., AND DODUNEKOV, S. Sufficient conditions for good and proper error-detecting codes. *IEEE Transactions on Information Theory* 43, 6 (1997), 2023–2026.
- [13] FOG, A. Vcl-c++ vector class library manual. *Copenhagen, Denmark: Technical University of Denmark* (2020).
- [14] FUJIWARA, T., AND KUSAKA, T. The weight distributions of the  $(256, k)$  extended binary primitive bch codes with  $k \leq 71$  and  $k \geq 187$ . *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences E104.A*, 9 (2021), 1321–1328.
- [15] GRASSL, M. Searching for linear codes with large minimum distance. In *Discovering Mathematics with Magma: Reducing the Abstract to the Concrete*, Springer, pp. 287–313.
- [16] GULLIVER, T. A., BHARGAVA, V. K., AND STEIN, J. M. Q-ary gray codes and weight distributions. *Applied mathematics and computation* 103, 1 (1999), 97–109.
- [17] HARVEY, D., HOEVEN, J. V. D., AND LECERF, G. Faster polynomial multiplication over finite fields. *J. ACM* 63, 6 (jan 2017).
- [18] HASSAN, S. A., HEMEIDA, A., AND MAHMOUD, M. M. Performance evaluation of matrix-matrix multiplications using intel’s advanced vector extensions (avx). *Microprocessors and Microsystems* 47 (2016), 369–374.
- [19] HERNANDO, F., IGUAL, F. D., AND QUINTANA-ORTÍ, G. Algorithm 994: Fast implementations of the brouwer-zimmermann algorithm for the computation of the minimum distance of a random linear code. *ACM Trans. Math. Softw.* 45, 2 (jun 2019).
- [20] HUFFMAN, W. C., KIM, J.-L., AND SOLÉ, P., Eds. *Concise Encyclopedia of Coding Theory*. Chapman and Hall/CRC, 2021.
- [21] HUFFMAN, W. C., AND PLESS, V. *Fundamentals of Error-Correcting Codes*. Cambridge University Press, Cambridge, UK, 2003.
- [22] INTEL. Intel® 64 and ia-32 architectures software developer’s manual, 2023.
- [23] JOICHI, J. T., WHITE, D. E., AND WILLIAMSON, S. G. Combinatorial gray codes. *SIAM Journal on Computing* 9, 1 (1980), 130–141.
- [24] KAWAHARA, Y., AOKI, K., AND TAKAGI, T. Faster implementation of  $\eta$ pairing over  $gf(3m)$  using minimum number of logical instructions for  $gf(3)$ -addition. In *Pairing-Based Cryptography – Pairing 2008* (Berlin, Heidelberg, 2008), S. D. Galbraith and K. G. Paterson, Eds., Springer Berlin Heidelberg, pp. 282–296.
- [25] KOLEVA-EFREMOVA, V. *Testing Performance and Scalability of the Pure MPI Model Versus Hybrid MPI-2/OpenMP Model on the Heterogeneous Supercomputer Avitohol*. Springer International Publishing, Cham, 2019, pp. 93–105.
- [26] KUSSWURM, D. *Modern X86 Assembly Language Programming: Covers X86 64-bit, AVX, AVX2, and AVX-512*. Apress, 2018.

- [27] MITRA, G., JOHNSTON, B., RENDELL, A. P., MCCREATH, E., AND ZHOU, J. Use of simd vector operations to accelerate application code performance on low-powered arm and intel platforms. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum* (2013), pp. 1107–1116.
- [28] PASHINSKA, M., AND BOUYUKLIEV, I. A parallel algorithm for computing the weight spectrum of binary linear codes. In *2020 Algebraic and Combinatorial Coding Theory (ACCT)* (2020), pp. 1–5.
- [29] QUINTANA-ORTÍ, G., HERNANDO, F., AND IGUAL, F. D. Algorithm 1033: Parallel implementations for computing the minimum distance of a random linear code on distributed-memory architectures. *ACM Trans. Math. Softw.* 49, 1 (mar 2023).
- [30] SCHÖNE, R., ILSCHÉ, T., BIELERT, M., GOCHT, A., AND HACKENBERG, D. Energy efficiency features of the intel skylake-sp processor and their impact on performance. In *2019 International Conference on High Performance Computing & Simulation (HPCS)* (2019), pp. 399–406.
- [31] WHITE, G. Enumeration-based algorithms in linear coding theory, 2006.
- [32] WHITE, G., AND GRASSL, M. A new minimum weight algorithm for additive codes. In *2006 IEEE International Symposium on Information Theory* (2006), pp. 1119–1123.
- [33] YAO, H., FAZELI, A., AND VARDY, A. A deterministic algorithm for computing the weight distribution of polar codes. In *2021 IEEE International Symposium on Information Theory (ISIT)* (2021), IEEE Press, p. 1218–1223.