# QDistRnd

## Calculate the distance of a q-ary quantum stabilizer code

0.3.0

30 July 2021

**Leonid P. Pryadko**

**Vadim A. Shabashov**

**Leonid P. Pryadko**
Email: leonid.pryadko@gmail.com
Homepage: http://faculty.ucr.edu/~leonid
Address: Leonid Pryadko
Department of Physics & Astronomy
University of California
Riverside, CA 92521
USA

**Vadim A. Shabashov**
Email: vadim.art.shabashov@gmail.com
Homepage: https://sites.google.com/view/vadim-shabashov/
Address: Vadim Shabashov
The Department of Physics & Engineering
ITMO University
St. Petersburg, 197101
Russia

# Copyright



# Acknowledgements

We appreciate very much all past and future comments, suggestions and contributions to this package and its documentation provided by GAP users and developers.

# Contents

# Chapter 1

# Introduction

The GAP package QDistRnd gives a reference implementation of a probabilistic algorithm for finding the distance of a $q$-ary quantum low-density parity-check code linear over a finite field $F = GF(q)$. While there is no guarantee of the performance of the algorithm (the existing bounds in the case of quantum LDPC codes are weak, see 3.2.2), an empirical convergence criterion is given to estimate the probability that a minimum weight codeword has been found. Versions for CSS and regular stabilizer codes are given, see Section 4.1

In addition, a format for storing matrices associated with $q$-ary quantum codes is introduced and implemented, see Chapter 5 and Sec. 4.2. The format is based on the well establised MaTrix market eXchange (MTX) Coordinate format developed at NIST, and is designed for full backward compatibility with this format. Thus, the files are readable by any software package which supports MTX.

The routines in the package are derived from the code originally written by one of the authors (LPP). A related Covering Set algorithm has a provable performance for generic (non-LDPC) quantum codes based on random matrices [DKP17]. Implemented version is a variant of the random *information set* (IS) algorithm based on random column permutations and Gauss' elimination [Leo88] [Kru89] [CG90].

The GAP computer algebra system was chosen because of its excellent support for linear algebra over finite fields. Here we give a reference implementation of the algorithm, with a focus on matrix formats and generality, as opposed to performance. Nevertheless, the routines are sufficiently fast when dealing with codes of practically important block lengths $n \lesssim 10^3$.

# Chapter 2

# Examples

A few simple examples illustrating the use of the package. For more information see Chapter 4

## 2.1 The 5-qubit code

Generate the matrix of the 5-qubit code over GF(3) with the stabilizer group generated by cyclic shifts of the operator $X_0 Z_1 \bar{Z}_2 \bar{X}_3$ which corresponds to the polynomial $h(x) = 1 + x^3 - x^5 - x^6$ (a factor $X_i^a$ corresponds to a monomial $ax^{2i}$, and a factor $Z_i^b$ to a monomial $bx^{2i+1}$), calculate the distance, and save into the file.

Example

```
gap> q:=3;; F:=GF(q);;
gap> x:=Indeterminate(F,"x");; poly:=One(F)*(1+x^3-x^5-x^6);;
gap> n:=5;;
gap> mat:=QDR_DoCirc(poly,n-1,2*n,F);; #construct circulant matrix with 4 rows
gap> Display(mat);
 1 . . 1 . 2 2 . . .
 . . 1 . . 1 . 2 2 .
 2 . . . 1 . . 1 . 2
 . 2 2 . . . 1 . . 1
gap> d:=DistRandStab(mat,100,1,0 : field:=F,maxav:=20/n);
3
gap> WriteMTXE("tmp/n5_q3_complex.mtx",3,mat,
>          "% The 5-qubit code [[5,1,3]]_3",
>          "% Generated from h(x)=1+x^3-x^5-x^6",
>          "% Example from the QDistRnd GAP package"    : field:=F);
File tmp/n5_q3_complex.mtx was created
```

Here is the contents of the resulting file which also illustrates the `complex` data format. Here a pair $(a_{i,j}, b_{i,j})$ in row $i$ and column $j$ is written as 4 integers, "$i$ $j$ $a_{i,j}$ $b_{i,j}$", e.g., "1 2 0 1" for the second entry in the 1st row, so that the matrix in the file has $n$ columns.

Example

```
%%MatrixMarket matrix coordinate complex general
% Field: GF(3)
% The 5-qubit code [[5,1,3]]_3
% Generated from h(x)=1+x^3-x^5-x^6
% Example from the QDistRnd GAP package
% Values Z(3) are given
```

```
4 5 20
1 1 1 0
1 2 0 1
1 3 0 2
1 4 2 0
2 2 1 0
2 3 0 1
2 4 0 2
2 5 2 0
3 1 2 0
3 3 1 0
3 4 0 1
3 5 0 2
4 1 0 2
4 2 2 0
4 4 1 0
4 5 0 1
```

And now let us read the matrix back the file. In the simplest case, all optional parameters are read from the file. Output is a list: `[field,mode,matrix,(list of comments)]`. Notice that a `mode=2` or `mode=3` matrix is always converted to `mode=1`, i.e., with $2n$ intercalated columns $(a_1,b_1,a_2,b_2,\ldots)$.

Example

```
gap> lis:=ReadMTXE("tmp/n5_q3_complex.mtx");;
gap> lis[1]; # the field
GF(3)
gap> lis[2]; # converted to ‘mode=1‘
1
gap> Display(lis[3]);
 1 . . 1 . 2 2 . . .
 . . 1 . . 1 . 2 2 .
 2 . . . 1 . . 1 . 2
 . 2 2 . . . 1 . . 1
```

The remaining portion is the list of comments. Notice that the 1st and the last coment lines have been added automatically.

Example

```
gap> lis[4];
[ "% Field: GF(3)", "% The 5-qubit code [[5,1,3]]_3",
  "% Generated from h(x)=1+x^3-x^5-x^6",
  "% Example from the QDistRnd GAP package", "% Values Z(3) are given" ]
```

## 2.2 Hyperbolic codes from a file

Here we read a pair of matrices from two different files which correspond to a hyperbolic code $[[80,18,5]]$ with row weight $w=5$ and the asymptotic rate $1/5$. Notice that `pair=0` is used for both files (regular matrices).

Example

```
gap> lisX:=ReadMTXE("matrices/QX80.mtx",0);;
gap> GX:=lisX[3];;
gap> lisZ:=ReadMTXE("matrices/QZ80.mtx",0);;
```

```
gap> GZ:=lisZ[3];;
gap> DistRandCSS(GX,GZ,100,1,2:field:=GF(2));
5
```

Here are the matrices for a much bigger hyperbolic code $[[900,182,8]]$ from the same family. Note that the distance here scales only logarithmically with the code length (this code takes about 15 seconds on a typical notebook and will not actually be executed).

Example

```
gap> lisX:=ReadMTXE("matrices/QX900.mtx",0);;
gap> GX:=lisX[3];;
gap> lisZ:=ReadMTXE("matrices/QZ900.mtx",0);;
gap> GZ:=lisZ[3];;
gap> DistRandCSS(GX,GZ,1000,1,0:field:=GF(2));
8
```

## 2.3 Randomly generated cyclic codes

As a final and hopefully somewhat useful example, the file "lib/cyclic.g" contains a piece of code searching for random one-generator cyclic codes of length $n = 15$ over the field $GF(5)$, and generator weight `wei=4`. Note how the `mindist` parameter and the option `maxav` are used to speed up the calculation.

# Chapter 3

# Description of the algorithm

## 3.1 Elementary version

### 3.1.1 What it does?

In the simplest possible terms, we are given a pair of matrices $P$ and $Q$ with orthogonal rows, $PQ^T = 0$. The matrices have entries in a finite field $F = GF(q)$, where $q$ is a power of a prime. The goal is to find the smallest weight of a vector $c$ over the same field $F$, such that $c$ be orthogonal with the rows of $P$, $Pc^T = 0$, and linearly independent from the rows of $Q$.

### 3.1.2 The algorithm

We first construct a generator matrix $G$ whose rows form a basis of the $F$-linear space of all vectors orthogonal to the rows of $P$. At each step, a random permutation $S$ is generated and applied to the columns of $G$. Then, Gauss' elimination with back substitution renders the resulting matrix to the reduced row echelon form, after which the inverse permutation $S^{-1}$ is applied to the columns. Rows of the resulting matrix $G_S$ that are linearly independent from the rows of $Q$ are considered as candidates for the minimum weight vectors. Thus, after $N$ steps, we are getting an upper bound on the distance which is improving with increasing $N$.

### 3.1.3 Intuition

The intuition is that each row of $G_S$ is guaranteed to contain at least $\texttt{rank}(G_S) - 1$ zeros. Thus, we are sampling mostly lower-weight vectors from the linear space orthogonal to the rows of $P$.

### 3.1.4 CSS version of the algorithm

The described version of the algorithm is implemented in the function `DistRandCSS` 4.1.2. It applies to the case of Calderbank-Shor-Steane (CSS) codes, where the matrices $P = H_X$ and $Q = H_Z$ are called the CSS generator matrices, and the computed minimum weight is the distance $d_Z$ of the code. The number of columns $n$ is the block length of the code, and it encodes $k$ qudits, where $k = n - \texttt{rank}(H_X) - \texttt{rank}(H_Z)$. To completely characterize the code, we also need the distance $d_X$ which can be obtained by calling the same function with the two matrices interchanged. The conventional code distance $d$ is the minimum of $d_X$ and $d_Z$. Parameters of such a $q$-ary CSS code are commonly denoted as $[[n,k,(d_X,d_Z)]]_q$, or simply $[[n,k,d]]_q$ as for a general $q$-ary stabilizer code.

### 3.1.5 Generic version of the algorithm

CSS codes are a subclass of general $F$-linear stabilizer codes which are specified by a single stabilizer generator matrix $H = (A|B)$ written in terms of two blocks of $n$ columns each. The orthogonality condition is given in a symplectic form,

$$AB^T - BA^T = 0,$$

or, equivalently, as orthogonality between the rows of $H$ and the symplectic-dual matrix $\tilde{H} = (B|-A)$. Non-trivial vectors in the code must be orthogonal to the rows of $P = \tilde{H}$ and linearly independent from the rows of $Q = H$. The difference with the CSS version of the algorithm is that we must minimize the *symplectic* weight of $c = (a|b)$, given by the number of positions $i$, $1 \le i \le n$, such that either $a_i$ or $b_i$ (or both) be non-zero.

The parameters of such a code are denoted as $[[n,k,d]]_q$, where $k = n - \mathtt{rank}H$ is the number of encoded qudits, and $d$ is the minimal symplectic weight of a non-trivial vector in the code. It is easy to check that a CSS code can also be represented in terms of a single stabilizer generator matrix. Namely, for a CSS code with generators $H_X$ and $H_Z$, the stabilizer generator matrix has a block-diagonal form, $H = \mathtt{diag}(H_X, H_Z)$.

A version of the algorithm for general $F$-linear stabilizer codes is implemented in the function `DistRandStab` 4.1.3.

*Important Notice*: In general, here one could use most general permutations of $2n$ columns, or restricted permutations of $n$ two-column blocks preserving the pair structure of the matrix. While the latter method would be much faster, there is no guarantee that every vector would be found. As a result, we decided to use general permutations of $2n$ columns.

## 3.2 Some more details

### 3.2.1 Quantum stabilizer codes

Representation of quantum codes in terms of linear spaces is just a convenient map. In the case $q = 2$ (qubits), the details can be found, e.g., in the book of Nielsen and Chuang, [NC00]. Further details on the theory of stabilizer quantum error correcting codes based on qubits can be found in the Caltech Ph.D. thesis of Daniel Gottesman [Got97] and in the definitive 1997 paper by Calderbank, Rains, Shor, and Sloane [CRSS98]. Theory of stabilizer quantum codes based on qudits ($q$-state quantum systems) was developed by Ashikhmin and Knill [AK01] (prime fields with $q$ prime) and by Ketkar, Klappenecker, Kumar, & Sarvepalli [KKKS06] (extension fields with $q$ a non-trivial power of a prime).

In the binary case (more generally, when $q$ is a prime), $F$-linear codes coincide with *additive* codes. The *linear* codes [e.g., over $GF(4)$ in the binary case [CRSS98]] is a different construction which assumes an additional symmetry. A brief summary of $F$-linear quantum codes [where $F = GF(q)$ with $q = p^m$, $m > 1$ a non-trivial power of a prime] can be found in the introduction of Ref. [ZP20]. The construction is equivalent to a more physical approach in terms of a lifted Pauli group suggested by Gottesman [Got14].

### 3.2.2 The algorithm

Case of *classical linear codes*

The algorithm 3.1.2 is closely related to the algorithm for finding minimum-weight codewords in a classical linear code as presented by Leon [Leo88], and a related family of *information set* (IS) decoding algorithms [Kru89] [CG90].

Consider a classical linear $q$-ary code $[n,k,d]_q$ encoding $k$ symbols into $n$, specified by a generator matrix $G$ of rank $k$. Using Gauss' algorithm and column permutations, the generator matrix can be rendered into a *systematic form*, $G = (I|A)$, where the two blocks are $I$, the size-$k$ identity matrix, and a $k$ by $n-k$ matrix $A$. In such a representation, the first $k$ positions are called the information set of the code (since the corresponding symbols are transmitted directly) and the remaining $n-k$ symbols provide the redundancy. Any $k$ linearly-independent columns of $G$ can be chosen as the information set, which defines the systematic form of $G$ up to a permutation of the rows of $A$.

The IS algorithm and the original performance bounds [Leo88] [Kru89] [CG90] are based on the observation that for a long random code a set of $k+\Delta$ randomly selected columns, with $\Delta$ of order one, are likely to contain an information set. ISs are (approximately) in one-to-one correspondence with the column permutations, and a random IS can thus be generated as a set of *pivot* columns in the Gauss' algorithm after a random column permutation. Thus, if there is a codeword $c$ of weight $d$, the probability to find it among the rows of reduced-row-echelon form $G_S$ after a column permutation $S$ can be estimated as that for a randomly selected set of $k$ columns to hit exactly one non-zero position in $c$.

The statistics of ISs is more complicated in other ensembles of random codes, e.g., in linear *low-density parity-check* (LDPC) codes where the check matrix $H$ (of rank $n-k$ and with rows orthogonal to those of $G$) is additionally required to be sparse. Nevertheless, a provable bound can be obtained for a related *covering set* (CS) algorithm where a randomly selected set of $s \geq k-1$ positions of a putative codeword are set to be zero, and the remaining positions are constructed with the help of linear algebra. In this case, the optimal choice [DKP17] is to take $s \approx n(1-\theta)$, where $\theta$ is the erasure threshold of the family of the codes under consideration. Since $\theta \geq R$ (here $R = k/n$ is the code rate), here more zeros must be selected, and the complexity would grow (assuming the distance $d$ remains the same, which is usually *not* the case for LDPC codes).

Note however that rows of $G_P$ other than the last are not expected to contain as many zeros (e.g., the first row is only guaranteed to have $k-1$ zeros), so it is *possible* that the performance of the IS algorithm on LDPC codes is actually closer to that on random codes as estimated by Leon [Leo88].

Case of *quantum CSS codes*

In the case of a random CSS code (with matrices $P$ and $Q$ selected randomly, with the only requirement being the orthogonality between the rows of $P$ and $Q$), the performance of the algorithm 3.1.2 can be estimated as that of the CS algorithm, in terms of the erasure threshold of a linear code with the parity matrix $P$, see [DKP17].

Unfortunately, such an estimate fails dramatically in the case of *quantum LDPC codes*, where rows of $P$ and $Q$ have weights bounded by some constant $w$. This is a reasonable requirement since the corresponding quantum operators (supported on $w$ qudits) have to actually be measured frequently as a part of the operation of the code, and it is reasonable to expect that the measurement accuracy goes down (exponentially) quickly as $w$ is increased. Then, the linear code orthogonal to the rows of $P$ has the distance $\leq w$ (the minimal weight of the rows of $Q$), and the corresponding erasure threshold is exactly zero. In other words, there is a finite probability that a randomly selected $w$ symbols contain a vector orthogonal to the rows of $P$ (and such a vector would likely have nothing to do with non-trivial *quantum* codewords which must be linearly independent from the rows of $Q$).

On the other hand, for every permutation $S$ in the algorithm 3.1.2, the matrix $G_S$ contains exactly $k = n - \mathtt{rank}(P) - \mathtt{rank}(Q)$ rows orthogonal to rows of $P$ and linearly independent from rows of $Q$

(with columns properly permuted). These vectors contain at least $s$ zeros, where $[1-\theta_*(P,Q)]n \le s \le n-\mathtt{rank}(Q)$, where $\theta_*(P,Q)$ is the erasure threshold for $Z$-like codewords in the quantum CSS code with $H_X = P$ and $H_Z = Q$.

*What is it that we do not understand?*

What missing is an understanding of the statistics of the ISs of interest, namely, the ISs that overlap with a minimum-weight codeword in one (or a few) positions.

Second, we know that a given column permutation $S$ leads to the unique information set, and that every information set can be obtained by a suitably chosen column permutation. However, there is no guarantee that the resulting information sets have equal probabilities. In fact, it is easy to construct small matrices where different information sets are obtained from different numbers of column permutations (and thus have different probabilities). It is not clear whether some of the ISs may have vanishingly small probabilities in the limit of large codes; in such a case the algorithm would fail.

## 3.3 Empirical estimate of the success probability

The probability to find a codeword after $N$ rounds of the algorithm can be estimated empirically, by counting the number of times each codeword of the minimum weight was discovered. We *expect* the probability $P(c)$ to discover a given codeword $c$ to depend only on its (symplectic) weight $\mathtt{wgt}(c)$, with the probability a monotonously decreasing function of the weight. If, after $N$ steps, codewords $c_1, c_2, \ldots, c_m$ of the same (minimal) weight $w$ are discovered $n_1, n_2, \ldots, n_m$ times, respectively, we can estimate the corresponding Poisson parameter as

$$\lambda_w = \frac{1}{Nm}\sum_{i=1}^{m} n_i.$$

Then, the probability that a codeword $c_0$ of the true minimal weight $d < w$ be *not* discovered after $N$ steps can be upper bounded as (the inequalities here become equalities in the limit of small $\lambda_w$)

$$P_{\text{fail}} < (1-\lambda_w)^N < e^{-N\lambda_w} = \exp\left(-m^{-1}\sum_{i=1}^{m} n_i\right) \equiv \exp(-\langle n\rangle).$$

Thus, the probability to fail is decreasing as an exponent of the parameter $\langle n\rangle$, the *average number of times a minimum-weight codeword has been found.*

The hypothesis about all $P(c_i)$ being equal to $\lambda_w$ is testable, e.g., if one considers the distribution of the ratios $x_i = n_i/N$, where $N = \sum_{i=1}^{m} n_i$ is the total number of codewords found. These quantities sum up to one and are distributed according to multinomial distribution[Ste53]. Further, under our assumption of all $P(c_i)$ being equal, we also expect the outcome probabilities in the multinomial distribution to be all equal, $\pi_i = 1/m$, $1 \le i \le m$.

This hypothesis can be tested using Pearson's $\chi^2$ test. Namely, in the limit where the total number of observations $N$ diverges, the quantity

$$X^2 = \sum_{i=1}^{m}\frac{(n_i - N\pi_i)^2}{N\pi_i} = N^{-1}\sum_{i=1}^{m}\frac{n_i^2}{\pi_i} - N \overset{\pi_i=1/m}{\to} \frac{m}{N}\sum_{i=1}^{m} n_i^2 - N,$$

is expected to be distributed according to the $\chi^2_{m-1}$ distribution with $m-1$ parameters, see [CL54] [Cra99].

In practice, we can approximate with the $\chi^2_{m-1}$ distribution as long as the total $N$ be large compared to the number $m$ of the codewords found (i.e., the average $\langle n \rangle$ must be large, which is the same condition as needed for confidence in the result.)

With `debug[4]` set (binary value 8) in `DistRandCSS` and `DistRandStab` 4.1, whenever more than one minimum-weight vector is found, the quantity $X^2$ is computed and output along with the average number of times $\langle n \rangle$ a minimum-weight codeword has been found. However, no attempt is made to analyze the corresponding value or calculate the likelihood of the null hypothesis that the codewords be equiprobable.

# Chapter 4

# All Functions

## 4.1 Functions for computing the distance

### 4.1.1 Examples

Here are a few simple examples illustrating the use of distance functions. In all examples, we use `DistRandCSS` and `DistRandStab` with `debug=2` to ensure that row orthogonality in the input matrices is verified.

Example

```
gap> F:=GF(5);;
gap> Hx:=One(F)*[[1,-1,0,0 ],[0,0,1,-1]];;
gap> Hz:=One(F)*[[1, 1,1,1]];;
gap> DistRandCSS(Hz,Hx,100,0,2 : field:=F);
2
```

Now, if we set the minimum distance `mindist` parameter too large, the function terminates immediately after a codeword with such a weight is found; in such a case the result is returned with the negative sign.

Example

```
gap> DistRandCSS(Hz,Hx,100,2,2 : field:=F);
-2
```

The function `DistRandStab` takes only one matrix. This example uses the same CSS code but written into a single matrix. Notice how the values from the previous example are intercalated with zeros.

Example

```
gap> F:=GF(5);;
gap> H:=One(F)*[[1,0, -1,0,  0,0,  0,0 ], # original Hx in odd positions
>              [0,0,  0,0,  1,0, -1,0 ],
>              [0,1,  0,1,  0,1,  0,1 ]];; # original Hz in even positions
gap> DistRandStab(H,100,0,2 : field:=F);
2
```

### 4.1.2 DistRandCSS

▷ `DistRandCSS(`*HX, HZ, num, mindist[, debug]: field := GF(2), maxav := fail*`)`

(function)

**Returns:** An upper bound on the CSS distance $d_Z$

Computes an upper bound on the distance $d_Z$ of the $q$-ary code with stabilizer generator matrices $H_X, H_Z$ whose rows are assumed to be orthogonal (*orthogonality is not verified*). See Section 3.1 for the description of the algorithm.

Details of the input parameters

- `HX`, `HZ`: the input matrices with elements in the Galois `field` $F$
- `num`: number of information sets to construct (should be large)
- `mindist` - the algorithm stops when distance equal or below `mindist` is found and returns the result with negative sign. Set `mindist` to 0 if you want the actual distance.
- `debug`: optional integer argument containing debug bitmap (default: `0`)
  - 1 (0s bit set) : print 1st of the vectors found
  - 2 (1st bit set) : check orthogonality of matrices and of the final vector
  - 4 (2nd bit set) : show occasional progress update
  - 8 (3rd bit set) : maintain cw count and estimate the success probability
- `field` (Options stack): Galois field, default: $GF(2)$.
- `maxav` (Options stack): if set, terminate when $\langle n \rangle$>`maxav`, see Section 3.1.4. Not set by default.

### 4.1.3 DistRandStab

▷ `DistRandStab(H, num, mindist[, debug]: field := GF(2), maxav := fail)` (function)

**Returns:** An upper bound on the code distance $d$

Computes an upper bound on the distance $d$ of the $F$-linear stabilizer code with generator matrix $H$ whose rows are assumed to be symplectic-orthogonal, see Section 3.1.5 (*orthogonality is not verified*).

Details of the input parameters:

- `H`: the input matrix with elements in the Galois `field` $F$ with $2n$ columns $(a_1, b_1, a_2, b_2, \ldots, a_n, b_n)$.

The remaining options are identical to those in the function `DistRandCSS` 4.1.2.

- `num`: number of information sets to construct (should be large)
- `mindist` - the algorithm stops when distance equal or smaller than `mindist` is found - set it to 0 if you want the actual distance
- `debug`: optional integer argument containing debug bitmap (default: `0`)
  - 1 (0s bit set) : print 1st of the vectors found
  - 2 (1st bit set) : check orthogonality of matrices and of the final vector
  - 4 (2nd bit set) : show occasional progress update
  - 8 (3rd bit set) : maintain cw count and estimate the success probability
- `field` (Options stack): Galois field, default: $GF(2)$.
- `maxav` (Options stack): if set, terminate when $\langle n \rangle$>`maxav`, see Section 3.1.4. Not set by default.

## 4.2 Input/Output Functions

### 4.2.1 ReadMTXE

▷ `ReadMTXE(FilePath[, pair]: field := GF(2))` (function)

**Returns:** a list [`field`, `pair`, `Matrix`, `array_of_comment_strings`]

Read matrix from an MTX file, an extended version of Matrix Market eXchange coordinate format supporting finite Galois fields and two-block matrices $(A|B)$ with columns $A = (a_1, a_2, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_n)$, see Chapter 5.

- `FilePath` name of existing file storing the matrix
- `pair` (optional argument): specifies column ordering; must correlate with the variable `type` in the file
  - `pair=0` for regular single-block matrices (e.g., CSS) `type=integer` (if `pair` not specified, `pair=0` is set by default for `integer`)
  - `pair=1` intercalated columns with `type=integer` $(a_1, b_1, a_2, b_2, \ldots)$
  - `pair=2` grouped columns with `type=integer` $(a_1, a_2, \ldots, a_n\, b_1, b_2, \ldots, b_n)$
  - `pair=3` this is the only option for `type=complex` with elements specified as "complex" pairs
- `field` (Options stack): Galois field, default: $GF(2)$. *Must* match that given in the file (if any).

*Notice*: with `pair=1` and `pair=2`, the number of columns specified in the file must be even, twice the block length of the code

1st line of file must read:

Code

```
%%MatrixMarket matrix coordinate ‘type‘ general
```

with `type` being either `integer` or `complex`

2nd line (optional) may contain:

Code

```
% Field: ‘valid_field_name_in_Gap‘
```

Any additional entries in the second line are silently ignored. By default, $GF(2)$ is assumed; the default can be overriden by the optional `field` argument. If the field is specified both in the file and by the optional argument, the corresponding values must match.

See Chapter 5 for the details of how the elements of the group are represented depending on whether the field is a prime field ($q$ a prime) or an extension field with $q = p^m$, $p$ prime, and $m > 1$.

### 4.2.2 WriteMTXE

▷ `WriteMTXE(StrPath, pair, matrix[, comment[, comment]]: field := GF(2))` (function)

**Returns:** no output

Export a `matrix` in Extended MatrixMarket format, with options specified by the `pair` argument.

- `StrPath` - name of the file to be created;

- `pair`: parameter to control the format details, must match the `type` of the matrix.
  - `pair=0` for regular matrices (e.g., CSS) with `type=integer`
  - `pair=1` for intercalated columns $(a_1, b_1, a_2, b_2, \ldots)$ with `type=integer`
  - `pair=2` for grouped columns with `type=integer` *(this is not supported!)*
  - `pair=3` for columns specified in pairs with `type=complex`
- Columns of the input `matrix` must be intercalated unless `pair=0`
- optional `comment`: one or more strings (or a single list of strings) to be output after the MTX header line.

The second line specifying the field will be generated automatically only if the GAP Option `field` is present. As an option, the line can also be entered explicitly as the first line of the comments, e.g., `"% Field:  GF(256)"`

See Chapter 5 for the details of how the elements of the group are represented depending on whether the field is a prime field ($q$ a prime) or an extension field with $q = p^m$, $m > 1$.

## 4.3 Helper Functions

### 4.3.1 Examples

Example

```
gap> QDR_AverageCalc([2,3,4,5]);
3.5
```

Example

```
gap> F:=GF(3);;
gap> x:=Indeterminate(F,"x");; poly:=One(F)*(1-x);;
gap> n:=5;;
gap> mat:=QDR_DoCirc(poly,n,2*n,F);; # make a circulant matrix with 5 rows
gap> Display(mat);
 1 2 . . . . . . . .
 . . 1 2 . . . . . .
 . . . . 1 2 . . . .
 . . . . . . 1 2 . .
 . . . . . . . . 1 2
```

### 4.3.2 QDR_AverageCalc

▷ `QDR_AverageCalc(vector)` (function)

Calculate the average of the components of a `vector` containing numbers

### 4.3.3 QDR_SymplVecWeight

▷ `QDR_SymplVecWeight(vector, field)` (function)

**Returns:** symplectic weight of a vector

Calculate the symplectic weight of a `vector` with an even number of entries from the field `field`. The elements of the pairs are intercalated: $(a_1, b_1, a_2, b_2, \ldots)$.

*Note: the parity of vector `length` and the format are not verified!!!*

### 4.3.4 QDR_WeightMat

▷ `QDR_WeightMat(matrix)` (function)

count the total number of non-zero entries in a matrix.

### 4.3.5 QDR_DoProbOut

▷ `QDR_DoProbOut(vector, n, num)` (function)

aux function to print out the relevant probabilities given the list `vector` of multiplicities of the codewords found. Additional parameters are `n`, the code length, and `num`, the number of repetitions; these are ignored in the present version of the program. See 3.3 for the importance of these parameters.

### 4.3.6 QDR_MakeH

▷ `QDR_MakeH(matrix, field)` (function)

**Returns:** `H` (the check matrix constructed)

Given a two-block `matrix` with intercalated columns $(a_1, b_1, a_2, b_2, \ldots)$, calculate the corresponding check matrix `H` with columns $(-b_1, a_1, -b_2, a_2, \ldots)$.

The parity of the number of columns is verified.

### 4.3.7 QDR_DoCirc

▷ `QDR_DoCirc(poly, m, n, field)` (function)

**Returns:** `m` by `2*n` circulant matrix constructed from the polynomial coefficients

Given the polynomial `poly` $a_0 + b_0x + a_1x^2 + b_1x^3 + \ldots$ with coefficients from the field `F`, constructs the corresponding `m` by `2n` double circulant matrix obtained by `m` repeated cyclic shifts of the coefficients' vector by $s = 2$ positions at a time.

# Chapter 5

# Extended MTX (MTXE) File Format

## 5.1 General information

The code supports reading matrices from an MTX file using `ReadMTXE` and writing new MTX file using `WriteMTXE` functions. Below a description of the format is given.

### 5.1.1 Representation of field elements via integers

Every finite field is isomorphic to a Galois field $F = GF(q)$, where $q$ is a power of a prime, $q = p^m$.

- When $q = p$ is a prime, $F$ is a prime field, isomorphic to the ring $\mathrm{Z}(q)$ of integers modulo $q$. In such a case, elements of the field are stored directly as integers from $\mathrm{Z}(p)$, ranging from 0 to $p-1$.

- When $q = p^m$ with $m > 1$, $F$ is an extension field. Elements of such a field could in principle be represented as polynomials modulo an irreducible polynomial with coefficients in the prime field $GF(p)$. In the actual file format, we chose to represent non-zero elements as integers, specifying the powers of a primitive element, while the zero field element is represented as $-1$. Notice that, depending on the choice of the primitive element, the elements of the field may be permuted. However, any such permutation is a field isomorphism preserving the multiplication table. In particular, orthogonality between the rows of the matrices is necessarily preserved.

- On input, other integer values are allowed; they are taken modulo $p$ for a prime field, and as the powers of the primitive element for an extension field. This is often convenient, e.g., with $\pm 1$ matrices which obey the orthogonality condition already over integers, and thus retain orthogonality over $\mathrm{Z}(q)$ with any $q$ (what is more relevant here, the same matrix would work with any prime field).

### 5.1.2 Matrix storage format

The generator matrix storage format dependes on 2 parameters: `pair` and `type`.

- With `pair=0` the matrix elements are stored in the usual order.. This is the defailt storage format for stabilizer generator matrices of CSS codes. In this case `type=integer`, since matrix elements are stored as integers.

- With `pair=1` the block matrix $(A,B)$ is stored with intercalated columns $(a_1,b_1,\ldots,a_n,b_n)$. In this case `type=integer`.
- With `pair=2` the block matrix $(A,B)$ is stored with separated columns $(a_1,\ldots,a_n,b_1,\ldots,b_n)$. In this case `type=integer`.
- With `pair=3` the block matrix $(A,B)$ is stored as a complex matrix $A+iB$, with columns $(a_1+ib_1,\ldots,a_n+ib_n)$. In this case `type=complex`, since matrix elements are represented as complex integers.

By default, `pair=0` corresponds to `type=integer` and `pair=3` corresponds to `type=complex`.

For efficiency reasons, the function `DistRandStab` 4.1 assumes the generator matrix with intercalated columns.

### 5.1.3 Explicit format of each line

The first line must have the following form:

Code
```
%%MatrixMarket matrix coordinate ‘type‘ general
```

with `type` either `integer` or `complex`.

The second line is optional and specifies the field. The field may be left undefined; by default, it is $GF(2)$, or it can be specified by hand when reading the matrices.

Code
```
% Field: GF(q)
```

Next the comment section is specified, with each line starting with the `%` symbol:

Code
```
% Example of the comment line
```

After the comment section there is the line with properties of the matrix:

Code
```
rows     columns     (number of non-zero elements)
```

Then all non-zero elements are listed according to the `type`:

- `type=integer`:

Code
```
i     j     element[i,j]
```

- `type=complex`:

Code
```
i     j     a[i,j]     b[i,j]
```

Notice that column and row numbers start with 1.

## 5.2 Example MTXE files

In this section we give two sample MTXE files storing the stabilizer generator matrix of 5-qubit codes.

First, matrix (with one redundant linearly-dependent row) stored with `type=integer` and `pair=1` (intercalated columns $[a_1, b_1, a_2, b_2, \ldots]$) is presented. Notice that the number of columns is twice the actual length of the code. Even though the field is specified explicitly, this matrix would work with any prime field.

Code

```
%%MatrixMarket matrix coordinate integer general
% Field: GF(7)
% 5-qubit code generator matrix / normal storage with intercalated cols
5 10 20
1 1 1
1 4 1
1 6 -1
1 7 -1
2 3 1
2 6 1
2 8 -1
2 9 -1
3 1 -1
3 5 1
3 8 1
3 10 -1
4 2 -1
4 3 -1
4 7 1
4 10 1
5 2 1
5 4 -1
5 5 -1
5 9 1
```

This same matrix is stored in the file `matrices/n5k1A.mtx`. This is how the matrix can be read and distance calculated:

Example

```
gap>     lis:=ReadMTXE("matrices/n5k1A.mtx" );;
gap>     Print("field ",lis[1],"\n");
field GF(7)
gap>     dist:=DistRandStab(lis[3],100,0 : field:=lis[1]);
3
```

The same matrix can also be stored with `type=complex` and `pair=3` (complex pairs $[a_1 + ib_1, a_2 + ib_2, \ldots]$). In this format, the number of columns equals the code length.

Code

```
%%MatrixMarket matrix coordinate complex general
% works with any prime field
% 5-qubit code generator matrix / normal storage with intercalated cols
% [[5,1,3]]_p
4 5 16
1 1 1 0
```

```
1 2 0 1
1 3 0 -1
1 4 -1 0
2 2 1 0
2 3 0 1
2 4 0 -1
2 5 -1 0
3 1 -1 0
3 3 1 0
3 4 0 1
3 5 0 -1
4 1 0 -1
4 2 -1 0
4 4 1 0
4 5 0 1
```

The matrix above is written in the file `matrices/n5k1.mtx`. To calculate the distance, we need to specify the field [unless we want to use the default binary field].

Example

```
gap>     lis:=ReadMTXE("matrices/n5k1.mtx" );;
gap>     Print("field ",lis[1],"\n");
field GF(2)
gap>     dist:=DistRandStab(lis[3],100,0,2 : field:=lis[1]);
3
gap>     q:=17;;
gap>     lis:=ReadMTXE("matrices/n5k1.mtx" : field:= GF(q));;
gap>     Print("field ",lis[1],"\n");
field GF(17)
gap>     dist:=DistRandStab(lis[3],100,0,2 : field:=lis[1]);
3
```

# References


[AK01] A. Ashikhmin and E. Knill. Nonbinary quantum stabilizer codes. *IEEE Trans. Info. Th.*, 47(7):3065–3072, Nov 2001. 9

[CG90] J. T. Coffey and R. M. Goodman. The complexity of information set decoding. *IEEE Trans. Info. Theory*, 36(5):1031 –1037, Sep 1990. 4, 10

[CL54] H. Chernoff and E. L. Lehmann. The use of maximum likelihood estimates in $\chi^2$ tests for goodness of fit. *The Annals of Mathematical Statistics*, 25(3):579 – 586, 1954. 11

[Cra99] Harald Cramér. *Mathematical Methods of Statistics (PMS-9)*. Princeton University Press, 1999. 11

[CRSS98] A. R. Calderbank, E. M. Rains, P. M. Shor, and N. J. A. Sloane. Quantum error correction via codes over GF(4). *IEEE Trans. Info. Theory*, 44:1369–1387, 1998. 9

[DKP17] I. Dumer, A. A. Kovalev, and L. P. Pryadko. Distance verification for classical and quantum LDPC codes. *IEEE Trans. Inf. Th.*, 63(7):4675–4686, July 2017. 4, 10

[Got97] Daniel Gottesman. *Stabilizer Codes and Quantum Error Correction*. PhD thesis, Caltech, 1997. 9

[Got14] Daniel Gottesman. Stabilizer codes with prime power qudits. Invited talk at QEC 2014 (ETH Zurich), 2014. 9

[KKKS06] A. Ketkar, A. Klappenecker, S. Kumar, and P. K. Sarvepalli. Nonbinary stabilizer codes over finite fields. *IEEE Trans. Info. Th.*, 52(11):4892–4914, Nov 2006. 9

[Kru89] E. A. Kruk. Decoding complexity bound for linear block codes. *Probl. Peredachi Inf.*, 25(3):103–107, 1989. (In Russian). 4, 10

[Leo88] J. S. Leon. A probabilistic algorithm for computing minimum weights of large error-correcting codes. *IEEE Trans. Info. Theory*, 34(5):1354 –1359, Sep 1988. 4, 10

[NC00] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Infomation*. Cambridge Unive. Press, Cambridge, MA, 2000. 9

[Ste53] R. G. D. Steel. Relation between Poisson and multinomial distributions. Biometrics Unit Technical Reports BU-39-M, Cornell University, 1953. 11

[ZP20] Weilei Zeng and Leonid P. Pryadko. Minimal distances for certain quantum product codes and tensor products of chain complexes. *Phys. Rev. A*, 102:062402, 2020. 9

# Index