Java – Finding Optimal Quaternary Hermitian Linear Complementary Dual Codes

Maysara Al Jumaily

April 21st, 2022

Contents

1 Main Overview

This Java program is a part of the M.Sc. in Computer Science thesis submitted by Maysara Al Jumaily (aljumaily.maysara@gmail.com) created to find optimal Hermitian Linear Complementary Dual (LCD) for Entanglement-Assisted Quantum Error Correction (EAQEC) codes (EAQECC).

- A `.pdf` version of this page is found here:
- The thesis is attached on GitHub and can be accessed from here
- The errata sheet of the thesis manuscript is attached on GitHub and can be accessed from here
- This is the initial release of the first version
  - In the meantime, there is no plan to release a second version
  - Please report (via email) any *major* bugs encountered in the program
    - A major bug is something that will give incorrect result (*i.e.*, incorrect generator matrix, incorrect Hamming distance, etc.)
  - Minor bugs can be reported on GitHub
  - There is no usage of GUI nor entering input via command-line. Please refer to the section on how to run the program
- The program uses Java 8 (for the code) and Java 16 for the code (use Java 16 to have everything work as expected)
- The source code is found here
- The documentation of this program is done *very* professionally and can be accessed through https://aljumaily.github.io/MScThesis/
- The IntelliJ IDEA 2021.2.4 (Community Edition) IDE was used to complete the code
- A `.zip` file of the IntelliJ project can be accessed from here
  - It will help with rendering LaTeX in the JavaDocs and add syntax highlighting to the code snippets

2 Program Implementation

2.1 Quaternary Field

The program focused on Hermitian LCD Quaternary codes. The quaternary digits used are: 0, 1, ω and $\overline{\omega}$, provided that $\overline{\omega} = \omega + 1 = \omega^2$ and $\omega^3 = 1$. Here are the addition, multiplication, division and conjugation tables in base 4 (note that adding in base 4 is the same as subtracting in base 4):

$+$	0	1	ω	$\overline{\omega}$
0	0	1	ω	$\overline{\omega}$
1	1	0	$\overline{\omega}$	ω
ω	ω	$\overline{\omega}$	0	1
$\overline{\omega}$	$\overline{\omega}$	ω	1	0

$\times$	0	1	ω	$\overline{\omega}$
0	0	0	0	0
1	0	1	ω	$\overline{\omega}$
ω	0	ω	$\overline{\omega}$	1
$\overline{\omega}$	0	$\overline{\omega}$	1	ω

$\div$	0	1	ω	$\overline{\omega}$
0	–	0	0	0
1	–	1	$\overline{\omega}$	ω
ω	–	ω	1	$\overline{\omega}$
$\overline{\omega}$	–	$\overline{\omega}$	ω	1

x	$x^\dagger$
0	0
1	1
ω	$\overline{\omega}$
$\overline{\omega}$	ω

x	Decimal	Binary
0	0	00
1	1	01
ω	2	10
$\overline{\omega}$	3	11

Table 1: The addition, multiplication, division conjugation and binary representation of quaternary elements from left to right.

2.2 Java Implementation

- A `long`, which is 64-bit two's complement integer, is used to represent a vector/codeword in the program
- A matrix is a 1-D array of `long`s
- All operations on vectors and matrices are done through binary manipulation
- The default constructors of classes in `hlcd.parameters` are hardcoded with initial values in the case where the parameters are not explicitly specified

The two crucial parameters are found in `hlcd.parameters.CodeParameters`. They are `appendIdentity` and `restrictCodewordGeneration`. Both should always be `true`. Appending the identity will cut down the search space as any generator matrix can be written in standard form $\left[\ I \mid P\ \right]$, where I is the identity

matrix and P is some other matrix. Placing a restriction on the codeword generation will further cut down the search space. It will hard code the top row to the appropriate number of 1's. Also, it will generate a subvector (defined next) where the first nonzero element from the left side is 1. In the program, a *subvector* or *subcodeword* refers to the right-side of the vector *without* appending the portion found in the identity matrix. The class `hlcd.operations.VectorGenerator` goes into this further in the documentation.

2.3 The mechanics of the program

The classes `hlcd.operations.VectorGenerator` and `hlcd.linearCode.Code` are the crucial components of the program.

The `VectorGenerator` class contains the parameters `appendIdentity` and `restrictCodewordGeneration`. In the case where

- `appendIdentity` is **`false`** and `restrictCodewordGeneration` is **`false`**, the vector generation will start at vector `0` and increment by `1` until a vector that satisfies the minimum weight of the code is found. The method `getNextFullVector()` is used to return the next valid vector that satisfies the minimum weight. This will continue until all $4^n - 1$ vectors have been examined, *i.e.*, all vectors in the search space. Then, the program will backtrack or conclude there doesn't exist a code with the specified `n` , `k` and `d` values when the top row examined all the vectors in search space
  - It is the slowest of all options and its usage is discouraged
- `appendIdentity` is **`false`** and `restrictCodewordGeneration` is **`true`**
  - The program will automatically change `appendIdentity` to **`true`**
- `appendIdentity` is **`true`** and `restrictCodewordGeneration` is **`false`**, the vector generation will examine $4^{n-k} - 1$ vectors
- `appendIdentity` is **`true`** and `restrictCodewordGeneration` is **`true`**
  - The starting vector will consist of `d-1` ones in the subvector of the P matrix and continues until it examines $4^{n-k-1} - 1$ vectors
    - At the worst case, the far-left cell in the P matrix will be hardcoded to 1, which will leave us with $n - k - 1$ cells in P to increment
  - This is the default case use in `hlcd.parameters.CodeParameters`

Furthermore, in the case where `restrictCodewordGeneration` is **`false`**, then the vector generator will traverse through the vectors as expected. However, if it is set to **`true`** (and provided that `appendIdentity` is **`true`** implicitly or explicitly), then the top row of the generator matrix will be hardcoded to $d - 1$ 1's in the P submatrix and the I submatrix will contain a single 1 making the entire row to have a weight of d. For the subsequent rows, the subcodeword in the P matrix will have the first nonzero element from the left side as 1 (*i.e.*, not 2 nor 3). This will cut down the space search even further.

The `Code` class will use `VectorGenerator` and recursively populate the generator matrix using the `backtrack(...)` method. Assume both `appendIdentity` and `restrictCodewordGeneration` are **`true`**:

- The very top row will be hardcoded
- A call to the `backtrack` method is performed to populate the next row
  - The vector generator will start at the current subvector and checks if the same value can be used while appending the identity subvector for the next row
    - For example, if the top row is: $\begin{bmatrix}1 & 0 & 0 & 0 & 1 & 1 & 1\end{bmatrix}$, then the next vector that will be examined is $\begin{bmatrix}0 & 1 & 0 & 0 & 1 & 1 & 1\end{bmatrix}$, *not* $\begin{bmatrix}0 & 1 & 0 & 0 & 1 & 1 & 2\end{bmatrix}$
    - In case $\begin{bmatrix}0 & 1 & 0 & 0 & 1 & 1 & 1\end{bmatrix}$ doesn't work, then it will be incremented to the following: $\begin{bmatrix}0 & 1 & 0 & 0 & 1 & 1 & 2\end{bmatrix}$
    - This is different from the original construction Masaaki Harada used where for $d \geq 3$, the codewords in matrix P cannot be equal
      - ⋆ There wasn't a proof to prove they have to be strictly less than
- Assume the vector generator returns a vector that satisfies the minimum weight
  - It will be checked with the other four codewords in the code (which are the zero vector, the hardcoded vector and the hardcoded vector multiplied by ω as well as $\overline{\omega}$)

- In the case where the vector is linearly orthogonal, then it will be identified as a codeword and will be placed in the generator matrix
  - Assume this is the case, then two rows are populated in the generator matrix and 16 codewords are found
  - A recursive call is performed to populate the next row
    - Assume all the possible vectors examined do not yield a valid solution
      - Backtrack to the last populated row
      - Find the next row that is larger than the current one
      - Repeat the process
- Continue until all the rows in the generator matrix are populated
  - This will also establish an ordering in the submatrix P where the subvector of the top row is the "smallest" value and the subvector of the bottom row is the "largest"
- In case there doesn't exist a code, then the generator matrix will either be a null matrix or populated with multiple duplicated rows with all-zero vector in the bottom row
- A validator would have to be used to ensure some simple properties of the generator matrix and code are satisfied such as all the codewords are unique (discussed later)

2.4 Speedups

The most frequent operations are finding the minimum weight of vectors and performing quaternary multiplication of vectors. Finding the weight of a quaternary vector is different from calculating the number of ones in a `long`. There needs to be a modification to the traditional approach used. The code snippet is copied from John Bollinger's answer and Java's official `Long` class documents.

```
private byte getWeightEngine(long v){
    v = (v & 0x5555_5555_5555_5555L) | ((v >>> 1) & 0x5555_5555_5555_5555L);
    v -= (v >>> 1) & 0x5555_5555_5555_5555L;
    v = (v & 0x3333_3333_3333_3333L) + ((v >>> 2) & 0x3333_3333_3333_3333L);
    v = (v + (v >>> 4)) & 0x0F0F_0F0F_0F0F_0F0FL;
    v += v >>> 8;
    v += v >>> 16;
    v += v >>> 32;
    return (byte) (v & 0x7f);
}
```

In the program, it is used in `hlcd.operations.HammingWeight`.

Furthermore, the multiplication of element-by-element quaternary vectors uses binary manipulation rather than a for-loop, which is at least 100 times faster. The code snippet is copied from Mark Dickinson's answer.

```
public long multiply(long v1, long v2) {
    long a = (v1 >>> 1) & 0x5555_5555_5555_5555L;
    long b = (v2 >>> 1) & 0x5555_5555_5555_5555L;
    return (((v1 & b) ^ (v2 & a)) << 1) ^ (a & b) ^ (v1 & v2);
}
```

In the program, it is used in `hlcd.operations.GF4Operations`.

To find the multiple of a vector:

- Use `0xAAAA_AAAA_AAAA_AAAAL` to multiply a vector by ω
- Use `0xFFFF_FFFF_FFFF_FFFFL` to multiply a vector by $\overline{\omega}$
- The methods `multiplyByScalarTwo(long v)` and `multiplyByScalarThree(long v)` take care of this in the `hlcd.operations.GF4Operations` class

2.5 Storing Codewords

A special array is used to store the codewords. Since the maximum array index java support is `Integer.MAX_VALUE - 8 = 2147483639`, it is not enough to store more codewords than this. Instead, the class `hlcd.operations.LongArray` is created which is a 2-D array, an array of arrays. It will create the appropriate number of 1-D arrays to satisfy the number of codewords to store. By default, all the cells will be initialized to the value `0`.

3 Features

The program checks for invalid input and write warnings when it can manage to continue running the program but change specific components. For example, if a specific column in the matrix needs to be replaced by another column of larger dimension, then a warning is displayed and the first appropriate rows are used.

There exists the ability to write the generator matrix and weight enumerator of the code to a LaTeX file. Also, the generator matrix (along with its transpose, Hermitian transpose and $G \cdot \overline{G}^T$) to a Matlab file to check the matrix properties as well as the code object (however, the program doesn't have a full support to *open* a `.bin` file).

There are multiple tests implemented to ensure the validity of the program:

- Checks there doesn't exist codewords whose value is 0 other than the all-zero vector
  - Java will initialize all the codewords to `0`
- Ensures all the codewords satisfy the minimum distance `d`
- Scans all the codeword to ensure they are unique and there is no duplicates
- Checking the determinant of $G \cdot \overline{G}^T$ is non-zero, which is a necessary condition for a code to be considered as a Hermitian LCD
- Uses the generator matrix and replicate the linear combination and ensures the codewords of the code matches the replicated set
- Test the Hermitian LCD property which exhaustively loop through all $base^n$ vectors in the space
  - Highly not recommended being executed because it will take *a lot* of time
- Allows for lots of options to be passed:
  - Ability to *significantly* cut down on the search space
  - Choosing the files to be exported
  - Printing on console the specified options
  - Capability to choose the validator tests that should be executed
- The program is *somewhat* designed to also include other bases
- The program implemented the ability to:
  - Multiply matrices
  - Find the determinant of a matrix using Bareiss Algorithm (a fraction-free algorithm)
    - As it is known that a matrix is invertible if its determinant is nonzero
  - Find Transpose and Hermitian form of a matrix
  - *Significantly* reducing the search space by having the vectors generated follow a similar construction to Masaaki Harada's approach
  - Find the weight enumerator of a code
  - Generate LaTeX outputs
  - Validate the code to ensure no bugs have occurred. This includes:
    - All codewords are unique (*i.e.*, there are no duplicates of codewords)
    - Ensure all the codewords have valid weights (*i.e.*, the minimum distance of the code is satisfied)

- Checking the determinant of $G \cdot \overline{G}^T$ is non-zero (a necessary condition for a Hermitian LCD code)
- Ensure there are no zero vectors other than the all-zero codeword (this is used because Java will initialize all the codewords to `0`)
- The ability to replicate the codewords found by using the generator matrix and find all linear combinations again
  - ⋆ The default set of codewords and the duplicated one *must* match
- A way to ensure the Hermitian LCD property is satisfied
  - ⋆ This is **not** recommended executing at all because it will go through all the 4^n possible vectors in the space

4 Execution of the Program

- First, ensure to hardcode the appropriate paths found in the `hlcd.Paths` class
  - There are two variables to handle: `DEFAULT_PATH` and `PARAMETERS_PATH`
  - `DEFAULT_PATH` will specify where the output file will be written in
    - It can be empty and the directory where the project is being executed will be used as the default location
  - `PARAMETERS_PATH` specifies the list of parameters that needs to be executed (a `.txt` file) along with the directory location, filename and `.txt` extension
    - The program will crash and an error message will be displayed if the file isn't found
- Executing the program can be done through the `hlcd.run` package
- To run a *single* parameter using the *simple* version of the program, run `SimpleExecutor` and ensure to specify the appropriate `n`, `k` and `d` values found in the constructor
- To run a *single* parameter using the *complex* version of the program, run `ComplexExecutor` and ensure to specify the appropriate `n`, `k` and `d` values found in the constructor along with all the other parameters of the program
- Both `SimpleExecutor` and `ComplexExecutor` *do not* use `PARAMETERS_PATH` as a single parameter is being tested
- To run *multiple* parameters, run `ListExecutor` and ensure the parameter list file and `PARAMETERS_PATH` are setup correctly (an example of the parameter list can be found here)

4.1 Setting up the IntelliJ project:

- As mentioned earlier, the `.zip` file of the project can be found here
- Once downloaded, open the `.zip` file and extract the single folder found onto the Desktop
- Open IntelliJ
  - To open a project, use the shortcut CTRL + SHIFT + O or manually go to File → Open → navigate to Desktop and IntelliJ should automatically identify it as a project
  - Select it with a single left-click and press the OK button
  - In case a popup window asks to trust the project, trust the project

4.2 Generating JavaDocs with LaTeX and Syntax Highlighting

This explains how to generate JavaDoc using IntelliJ.

- Surround LaTeX input using `\( ... \)`, *not* using dollar signs
  - For example, writing will yield `\(e^{i \pi} = -1\)` renders $e^{i\pi} = -1$
- To generate the default JavaDoc (use the shortcut CTRL + G) or navigate to Tools → Generate JavaDoc
- To generate customized Javadocs, download the `stylesheet.css` file from here then create a file named `options` without an extension (can find the file used from here). Write the following in `options` and make sure each option is a single line without a linebreak (note that the file paths are absolute using

Windows but refer to the official documents for other operating systems and pay close attention to \\ and // versus \ and /):

- There is a bug in IntelliJ regarding using relative path, instead, just use absolute path

```
-use
-splitindex
--main-stylesheet "C:\\hlcd-project\\src\\javadocs-tools\\stylesheet.css"
-doctitle "<h1>Hermitian Linear Complementary Dual Codes</h1>"
-windowtitle 'HLCD Javadoc'
-bottom '<div style="text-align:center"><img alt="Creative Commons License"
    style="border-width:0"
    src="https://i.creativecommons.org/l/by-nc-sa/4.0/88x31.png"/><br/>This work is
    licensed under a <a target="_blank"
    href="http://creativecommons.org/licenses/by-nc-sa/4.0/">Creative Commons
    Attribution-NonCommercial-ShareAlike 4.0 International License</a>.<br/>first
    name last name &copy; year<br/></div>'
-overview "C:\\hlcd-project\\src\\javadocs-tools\\overview.html"
--allow-script-in-comments -header "<script id="MathJax-script" async
    src="https://cdn.jsdelivr.net/npm/mathjax@3/es5/tex-mml-chtml.js">MathJax =
    {tex: { inlineMath: [['\\\\(', '\\\\)']], displayMath: [['\\\\[', '\\\\]'],
    ['\\begin{equation}', '\\end{equation}']], digits:
    /^(?:[0-9]+(?:\\{,\\}[0-9]{3})*(?:\\.[0-9]*)?|\\.[0-9]+)/,}};</script>"
```

- To generate the actual files of JavaDoc (use the shortcut CTRL + G) or navigate to Tools → Generate JavaDoc and type the following in the *Other command line arguments*: `@"C:\hlcd-project\src\javadocs-tools\options"`
- In the generated Javadoc folder, there is a `script.js` file. Open it and at the very end, append the code found in `syntax-highlighter.txt` that is accessed from here

5 Program Testing

The testing of this program can be found in the package `hlcd.testing`. It contains two different tests. The first uses Matlab for confirmation and the other uses Java.

5.1 Operations Tester

The operations of the program are tested and compared to what Matlab generates. It will test the modified Bareiss Algorithm for finding the determinant of a matrix as well as matrix multiplication. The Matlab files are generated to be *manually* opened in Matlab and executed. It is not done automatically. This assumes that Matlab is installed on the computer. It is not recommended to use Matlab's online version because files with gigabytes in size can be generated.

The test `hlcd.testing.matlabChecker.MatlabDeterminantTester` will generate matlab(s) files and matrices to test the determinant found based on the instance variables: `path`, `runs`, `iterations`, `minSize` and `maxSize`. To execute the Matlab files generated, run `matlab_determinant_script_execution.m` that is found along with the files generated.

The test `hlcd.testing.matlabChecker.MatlabMatrixTester` will generate matlab(s) files and matrices to test matrix operation (transpose, Hermitian transpose and matrix multiplication) based on the instance variables: `path`, `runs`, `iterations`, `minSize` and `maxSize`. To execute the Matlab files generated, run `matlab_matrix_multiplication_script_execution.m` that is found along with the files generated.

5.2 Optimization Tester

The optimizations of the program are thoroughly tested. Everything here is benchmarked as well.

The first is testing the code for finding the weight of quaternary vectors. The optimal approach is used in the program and slow approach uses a for-loop and iteratively finds the weight. Both are compared and should return the same output. There is the possibility to randomly generate vectors for testing or selecting specific vectors instead.

The last test is checking whether the multiplication of quaternary vectors is valid. Similar to the first test, this will compare the optimal approach used in this program with the slow approach that uses a for-loop. Both results are compared and should be equal.

The last test focuses on accessing matrix cell. Since a matrix is defined as 1-D array, accessing and/or setting a specific cell is done via binary manipulation. A random matrix will be created as well as another null matrix with equal dimension. The null matrix will be populated with the cells of the random matrix. This will get and set cells. At the end, the two matrices are should be equal and the rows of the matrices (which are `long`s) are checked.

5.3 Bug in Matlab

A significant bug has been encountered in Matlab. Even in the latest version R2021b Update 2 (9.11.0.1847648) Dec 31 2021. The Hermitian Transpose of a matrix doesn't work when the specified matrix is a part of the Galois Field package in the Communications Toolbox. Once applied, only the transpose will be applied, *not* the Hermitian portion. In essence, applying the Hermitian property on quaternary elements will keep 0 and 1 unchanged but swaps ω with $\overline{\omega}$ and vice-versa. Here is a code snippet to encounter the bug:

```
q = 2;
m = gf ( [
  0 0 0 0
  1 1 1 1
  2 2 2 2
  3 3 3 3
  0 1 2 3
] , q);

ctranspose(m)
```

Adding `.^2` will square each element in the matrix. In quaternary logic, $0^2 = 0 \cdot 0 = 0$, $1^2 = 1 \cdot 1 = 1$, $2^2 = 2 \cdot 2 = 3$ and $3^2 = 3 \cdot 3 = 2$, provided that $\omega = 2$ and $\overline{\omega} = 3$. Here is the correct code snippet that overcomes the bug:

```
q = 2;
m = gf ( [
  0 0 0 0
  1 1 1 1
  2 2 2 2
  3 3 3 3
  0 1 2 3
] , q);

ctranspose(m.^2)
```

6 Caveat and Limitation

- Only base 4 Hermitian LCD codes are tested, but we can simply expand to ordinary quaternary codes
  - This will not use the Bareiss Algorithm and require the modification of the base case in the `backtrack` method under `hlcd.linearCode.Code`

- Reaching `r >= K` will yield a valid quaternary code, there is no need to check for whether the matrix is invertible
- The code snippet is:

```
if (r >= K) {
    if (matrix.getGPrime().isInvertible()) {
        return true;
    } else {
        matrix.setRow((byte) (r - 1), 0);
        return false;
    }
}
```

- The minimum distance that can be used here is `d >= 3`. The program doesn't support minimum distances of `d = 2` or `d = 1`, due to the `appendIdentity` and `restrictCodewordGeneration` parameters
  - Maybe the logic of `hlcd.operation.VectorGenerator` can be modified to include all values of `d`
- Binary codes are implemented but not completely tested
- In the case where the filepath specified doesn't exist, the program will still execute, show no warnings but will **not** create the files
  - Make sure to test the with simple codes such as $[7, 4, 3]$
- There doesn't exist a GUI
- Running the code requires to programmatically change the values of the source code
  - On the bright side, the change is simple and is outlined in the execution section
- No library is created because of the hardcoding of paths, parameters, etc.
- There is an option of multithreading and has been implemented but cannot be used because it wasn't tested thoroughly

7 Future Work

- Include a way to obtain the parity-check matrix from the generator matrix that is in standard form
- Open the generated `.bin` files and continue the program execution from there
- Implement binary LCD and quaternary LCD (quaternary *Hermitian* LCD is what this program implemented)
- Include support to codes with minimum distance 1 or 2
- Add GUI to make the program more user-friendly
- Complete all the TODO suggestions in the code (there are about ten)
- Include a multithreading option where it is used when the check of whether a vector is linearly orthogonal to other codewords is being performed
- Instead of using an array to store the combination, implement the option for not using an array and recalculating the linear combinations each time. It will consume more time but there wouldn't be the need to have about 10 million gigabytes of RAM when `k = 25`
- A more significant addon is to use 128-bit vectors which would be a combination of two `long`s to represent a single vector. That way, we can increase the maximum `n` value to about 60
  - Don't use objects to represent a vector, stay with using simple `long`s
  - They will consume a lot of extra space (with the assumption that a combination array is being used)
  - Even-numbered indices in the generator matrix and combination array represent the beginning of a codeword whereas the odd-numbered indices represent the second portion of the same vector
- Instead of using Java for 128-bit codewords, you can use C/C++ and reply on GCC 4.6 or later
  - Use `unsigned __int128` as the type
    - Ensure the processor in the machine supports this kind of variable
  - It requires to translate everything into C/C++