Classification of Doubly-Even Linear Binary Codes: An Analysis of the SageMath Implementation

Tom Gadron
Bard College

Classification of Doubly-Even Linear Binary Codes: An Analysis of the SageMath Implementation

A Senior Project submitted to
The Division of Science, Mathematics, and Computing
of
Bard College

by
Tom Gadron

Annandale-on-Hudson, New York
May, 2023

Abstract

Classification of doubly-even linear binary codes involves finding and enumerating permutation equivalence classes of subspaces of the vector space F_2^n. This project provides an analysis and explanation of the SageMath functions written by Robert Miller that implement the algorithm used for generating these codes.

Contents

Acknowledgments

Thank you to my advisor Charles Doran for all your patience, guidance, and encouragement.

The LaTeX template used by this paper is built from combining the LaTeX template from Prof. Ethan Bloch's webpage `https://faculty.bard.edu/bloch/tex/` [2] and the bibtex citation `https://www.overleaf.com/learn/latex/Bibliography_management_with_bibtex`.

1
Introduction

With classification of doubly even binary linear error correcting codes, the goal is to find and count the equivalence classes of subspaces of the vector space F_2^n, where each vector has a weight divisible by 4, for a given length of n and dimension k. An additional condition for these subspaces is that $k \leq \frac{n}{2}$. Two codes are said to be equivalent if their columns can be permuted to arrive at the same code. This project is based off of the work by Charles Doran and Robert Miller in "Codes and Supersymmetry in One Dimension"[3], where they were originally studying Adinkras, they found and enumerated equivalences classes of these codes for up to n=28. The focus of this project is on the SageMath[5] implementation of the algorithm (written in Cython) used to generate these codes. Some goals of this project are to analyze this implementation to give better exposition on how exactly this algorithm works, point out some optimizations that could be made, translate the algorithm to the Julia[1] programming language, and additionally begin to implement parallelization in the algorithm. Hopefully this project could also serve as a "quickstart" guide to anyone else that wants to pick up this problem to try to make progress on it.

2 Explanation of Current SageMath Implementation

2.1 Some Terms and Definitions

Even though a code is a full vector subspace, it is practical to refer to codes by their basis matrix as a shorthand, since a basis defines a unique vector space.

Codes are vector subspaces, and often a vector in binary code will be referred to as a word of the code. The term ”word” will primarily be used in this paper instead of vector.

Child code: As defined in Appendix B.4 of ”Codes and Supersymmetry in One Dimension”[3], a child of a doubly-even code, D, is the doubly even code equal to the span of $D \bigcup \{x\}$, where x is not in D.

Something that comes up a few times within SageMath is using integer values as booleans. Cython treats 0 as False, and any other value as True.

Dimension: The dimension of a code is the number of rows in its basis matrix.

Length: The length of a code is the length of any word within the code.

Degree: Within the binary representation, all words are represented as unsigned 32-bit integers, so the degree of a code is the number of non-zero columns its basis matrix contains.

Weight of vector: The weight of a vector is the number of 1s it contains.

2.2 self_orthogonal_binary_codes

Found in the directory /sage-9.8/src/sage/coding in the databases.py file, the main function called that generates the equivalence classes of doubly-even codes is self_orthogonal_binary_codes . The parameters of this function are the length of the code n, the dimension of the code k, the divisibility, b, of the weight of all the basis vectors in the code (this is set to 4 for doubly-even codes), the parent code, parent, to recursively generate codes from, a BinaryCodeClassifier object BC, a boolean, equal, to determine whether to also return codes with length and dimension less than n and k, and a function, in_test, to test the size of codes. A user would generally only need to specify n, k, b, and equal when calling the function from Sage. Given n, k, the function will optionally return all codes with size less than or equal to n and k, or only the codes with size n and k. Looking at how this function is written, we see that for the original call of the function by a user to get doubly-even codes of size n, k would look like self_orthogonal_binary_codes(n, k, b=4, equal=True) Following the text of this function, because the parameter equal is True, this if-statement will be evaluated:

```
if equal:
        in_test = lambda M: (M.ncols() - M.nrows()) <= (n-k)
        # test for recursion: see if parent code has size
        # similar to size of codes we want
        out_test = lambda C: (C.dimension() == k) and (C.length() == n)
        # test for output: we only output codes that have the specified
        # dimensions that we want
```

As we can see, in_test tests the difference between the length of the code (number of columns in the basis matrix) and the dimension of the code (number of rows in the basis matrix). Because we know $k \leq \frac{n}{2}$, we have $n - k \geq n - \frac{n}{2} = \frac{n}{2}$. This function is used in determining which codes will be passed into the next level of recursion. The out_test function is just used to check that codes have the proper requested size, and is used when returning output from the function. Because the parameter parent defaults to None, this original call of the function will result in the evaluation of this if-statement:

```
if parent is None:
    # initial function call before recursion
    for j in range(d, n+1, d):
        M = Matrix(FiniteField(2), [[1]*j])
        # initial parents are 1 dimensional
        # one parent for each multiple of d (4 for doubly even)
        if in_test(M):
            for N in self_orthogonal_binary_codes(n, k, d, M, BC, in_test=
                                                          in_test):
                if out_test(N):
                    yield N
```

Thus the function will, for each multiple, j, of 4 less than n, begin by creating a code of j 1s of dimension 1, and then pass it in as a parent code for recursion. Seeing as we are starting with "base-case" codes of dimension 1, this is bottom-up recursion. The `yield` keyword here outputs the code but does not end the function, here is a more thorough explanation of the `yield` keyword.

For the next level of recursion with a `parent` that is not `None`, and because the default value of `equal` is `False`, we go into the `else` statement for both of the above if-statements.

```
else:
    in_test = lambda M: True
    out_test = lambda C: True
```

Here we see that when `equal=False`, all codes pass both `in_test` and `out_test`.

Next, the `else` statement of the other if-statement reads as:

```
else: # recursion
    C = LinearCode(parent)

    if out_test(C):
        yield C
    if k == parent.nrows():
        return
    # exit condition: the parent code has the desired dimension
    for nn in range(parent.ncols()+1, n+1):
        # iterate over codes with degree greater than the parent
        # and <= given degree
        if in_test(parent): # we can move this if statement outside of the loop
            for child in BC.generate_children(BinaryCode(parent), nn, d):
                for N in self_orthogonal_binary_codes(n, k, d, child, BC,
                                                          in_test=in_test):
                    if out_test(N):
                        # this test might be applied to some codes
                        # that have already passed it
                        yield N
```

First, the `parent` code is interpreted as a `LinearCode` object, and then is yielded. If the parent code has dimension equal to `k`, then no more child codes need to be generated, and so the function returns and ends. Otherwise, child codes are to be generated. The outer loop iteration determines the length of the child codes. Then, the next inner loop generates the child codes of length `nn` from the `parent` code, which are then passed back into `self_orthogonal_binary_codes` to continue recursion.

Figure 2.2.1 is a diagram showing the structure that this part of the algorithm takes on:

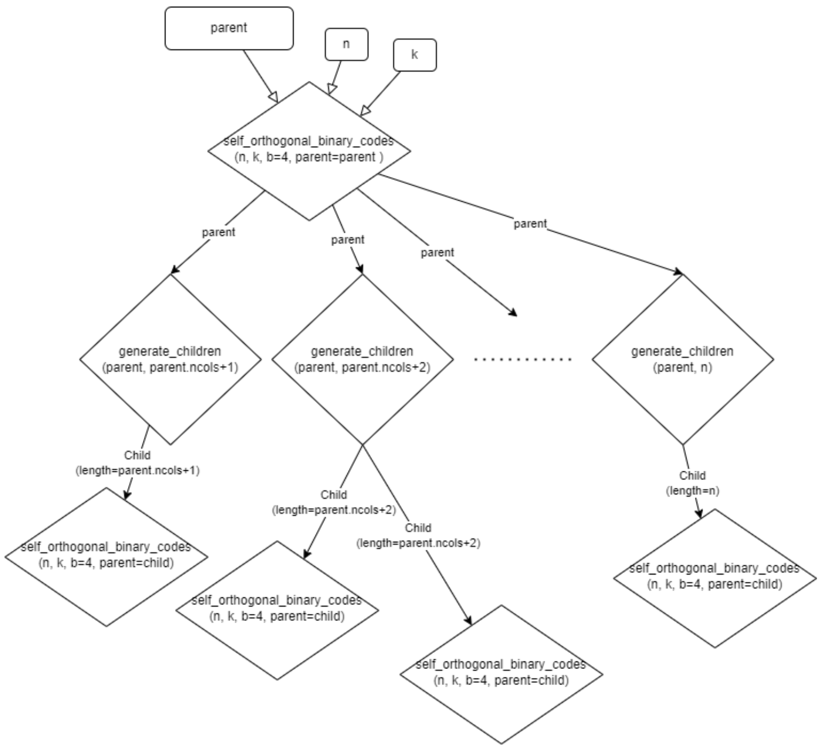

parent
n
k
self_orthogonal_binary_codes (n, k, b=4, parent=parent)
parent
parent
parent
parent
generate_children (parent, parent.ncols+1)
generate_children (parent, parent.ncols+2)
generate_children (parent, n)
Child (length=parent.ncols+1)
Child (length=parent.ncols+2)
Child (length=parent.ncols+2)
Child (length=n)
self_orthogonal_binary_codes (n, k, b=4, parent=child)
self_orthogonal_binary_codes (n, k, b=4, parent=child)
self_orthogonal_binary_codes (n, k, b=4, parent=child)
self_orthogonal_binary_codes (n, k, b=4, parent=child)

Figure 2.2.1: Recursion in the `self_orthogonal_binary_codes` function

This diagram illustrates how for a given parent code, a length n, and a dimension k, there is one call to the `generate_children` function for each each length of code from `parent.ncols+1` (the number of columns in the parent code) to n. Each call to `generate_children` can return some or no child codes, which are then fed back in to `self_orthogonal_binary_codes` to continute recursion.

2.3 `BinaryCode` objects and the `BinaryCodeClassifier` class

Found in the directory `/sage-9.8/src/sage/coding` in the `binary_code.pyx` file, the `BinaryCode` and `BinaryCodeClassifier` serve as computationally efficient ways to represent binary codes. The `.pyx` extension indicates that this is a Cython source file, so it is a mix of Python and C. Since each entry of a word in a binary code is a 0 or 1, then a word in a binary code can be interpreted as an integer in binary. For example, the word $(0, 1, 0, 1, 1, 0, 1)$ as a binary integer would be 1011010, or 90 in base ten. Note that written out, bits appear in the opposite order as a binary integer compared to their vector representation, as the low index entries of a vector appear on the left, but these correspond to the lower-order bits of the integer, which appear on the right when written. Often an integer will be cast to the custom type `codeword`, for example `word = <codeword>1`. `codeword` is just an alias for a 32-bit unsigned integer. Representing words as integers allows efficient computations using bitwise operations instead of operations over 2-dimensional arrays. For example, taking the bitwise XOR of two words is the same as doing vector addition.

`BinaryCode` objects can be initialized from a matrix or another `BinaryCode` object along with a word. The member variables of a `BinaryCode` object are:

- ncols: the number of columns in the code
- nrows: the number of rows in the code
- nwords: the number of words in the code, this is equal to 2^{nrows}
- basis: an array of integers representing a basis for the code
- words: an array of integers listing all the words of the code
- radix: maximum length of word that can be represented

The `BinaryCodeClassifier` class is used in various calculations relating to `BinaryCode` objects, notably containing the class methods:

- `_aut_gp_and_can_label`: calculates the automorphism group and canonical label a given binary code
- `generate_children`: Generate child codes of specified length for a given binary code

An important member variables of `BinaryCodeClassifier` objects is `ham_wts` which is an array containing the Hamming weights for each binary word up to length 16.

2.4 generate_children

Next, arguably the most important function of the algorithm is `generate_children` in the `binary_code.pyx` file. The parameters of this function are a `BinaryCode` object, `B`, which is the parent code to generate children from, an integer `n`, as the desired length of the code, and an integer `d`, which serves the same purpose as in the `self_orthogonal_binary_codes` function. This function returns one code from each permutation equivalence class that is a child of the parent code `B`.

The `generate_children` function starts with the call:

```
B.put_in_std_form()
```

This function performs row reduction on the code, and then permutes columns, which results in an identity matrix on the left and other data on the right.

For example, Figure 2.4.1 shows a $16, 8$ code with rows and columns shuffled:

Figure 2.4.2 shows the same code in standard form:

```
0 1 0 0 0 0 1 1 0 0 0 0 0 0 0 1
0 0 0 1 0 1 1 0 0 0 1 0 1 1 1 1
0 1 0 0 0 0 1 0 0 1 0 0 1 0 0 0
0 1 0 0 1 1 1 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 1 0 0 0 0 1 0 0 1 0
0 1 1 0 0 0 1 0 0 0 1 0 0 0 0 0
0 1 0 0 0 0 1 0 1 0 0 0 0 1 0 0
1 1 0 0 0 1 0 0 0 0 1 0 1 1 1 1
```

Figure 2.4.1: $16, 8$ code with rows and columns shuffled

```
1 0 0 0 0 0 0 0 1 1 1 1 1 1 0 1
0 1 0 0 0 0 0 0 0 1 0 1 0 0 1 0
0 0 1 0 0 0 0 0 0 0 1 1 0 0 1 0
0 0 0 1 0 0 0 0 1 1 1 0 1 1 1 1
0 0 0 0 1 0 0 0 1 0 0 1 0 0 1 0
0 0 0 0 0 1 0 0 0 0 0 1 0 0 1 1
0 0 0 0 0 0 1 0 0 0 0 1 0 1 1 0
0 0 0 0 0 0 0 1 0 0 0 1 1 0 1 0
```

Figure 2.4.2: $16, 8$ code in standard form

The next step in this function is to obtain an orthogonal basis for B:

```
ortho_basis = expand_to_ortho_basis(B, n)
```

In appendix B.3 of "Codes and Supersymmetry in One Dimension"[3], the format of the orthogonal basis generated is explained to look like Figure 2.4.3:

$$\left[\begin{array}{c|cc} I_k & \multicolumn{2}{c}{C''} \\ \hline 0 & I_{N-2k} & \multicolumn{1}{|c}{*} \end{array}\right]$$

Figure 2.4.3: Matrix Representing Structure of Orthogonal Basis

The output from the function `expand_to_ortho_basis` does not match this form exactly, as the columns are not guaranteed to be in the same order.

Here is a code in standard form and an orthogonal basis of length 12 for that code:

```
1 0 1 1 0 0 1
0 1 1 1 0 1 0
1 1 1 0 1 0 0
```

Figure 2.4.4: Code in standard form

```
1 0 0 0 0 1 1 0 1 0 0 0
1 0 0 0 1 0 0 0 0 0 0 0
1 0 0 1 0 0 0 0 0 0 0 0
1 0 1 0 0 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0 0 0 0 0
```

Figure 2.4.5: Orthogonal Basis for above code

Here is a permutation of the columns of the orthogonal basis to match the block matrix diagram:

```
1 0 0 0 0 0 0 0 1 1 0 1
0 1 0 0 0 0 0 0 0 0 0 1
0 0 1 0 0 0 0 0 0 0 0 1
0 0 0 1 0 0 0 0 0 0 0 1
0 0 0 0 1 0 0 0 0 0 0 1
```

Figure 2.4.6: Orthogonal Basis with columns permuted

The next step is to obtain the automorphism group of the original code:

```
aut_gp_gens, labeling, size, base = self._aut_gp_and_can_label(B)
```

The implementation of this function comes from the GUAVA package for GAP, as explained in the beginning of Appendix B in "Codes and Supersymmetry in One Dimension"[3].

As documented by comments in the `binary_code.pyx` file, the outputs are:

```
"""
gens -- a list of permutations (in list form) representing generators
    of the permutation automorphism group of the code CC.
labeling -- a permutation representing the canonical labeling of the
    code. mostly for internal use; entries describe the relabeling
    on the columns.
size -- the order of the automorphism group.
base -- a set of cols whose action determines the action on all cols
"""
```

A copy, in canonical form, of the original code is obtained by permuting the columns of the code according to the permutation represented by `labeling`:

```
    can_lab = create_word_perm(labeling[:B.ncols])
```

```
for i from 0 <= i < B.nrows:
        B_can_lab[i] = permute_word_by_wp(can_lab, B.basis[i])
```

Then `B_can_lab` is put into standard form through row reduction:

```
while row < B.nrows:
    i = row
    # move to next row until we find a 1 in the column
    # that 'current' represents
    while i < B.nrows and not B_can_lab[i] & current:
        i += 1
    if i < B.nrows:
        if i != row:
            swap = B_can_lab[row]
            B_can_lab[row] = B_can_lab[i]
            B_can_lab[i] = swap
        for j from 0 <= j < row:
            if B_can_lab[j] & current:
                B_can_lab[j] ^= B_can_lab[row]
        for j from row < j < B.nrows:
            if B_can_lab[j] & current:
                B_can_lab[j] ^= B_can_lab[row]
        row += 1
    current = current << 1
```

However, to my knowledge, this setting up of `B_can_lab` into standard form does not serve a purpose with regard to generating child codes, as the variable `B_can_lab` is not referenced anywhere else in the function.

Next the permutations representing the generators of the original code's automorphism group are initialized:

```
for i from 0 <= i < len(aut_gp_gens):
    parent_generators[i] =
               create_word_perm(aut_gp_gens[i] + list(xrange(B.ncols, n)))
```

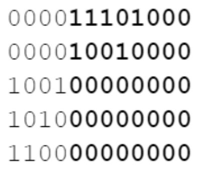
Since `B.ncols` is the number of columns of the original code, this piece concatenates the list representing the automorphism group generator with the list of the form $(B.ncols, \ldots, n-1)$, to account for the length specified for the child codes.

The next step is to determine, for the orthogonal basis based on the original code, how many rows make up the upper part of the basis matrix:

```
while ortho_basis[k] & (((<codeword>1) << B.ncols) - 1):
      k += 1
```

This expression: `((<codeword>1) << B.ncols) - 1` amounts to $2^{\text{B.ncols}} - 1$, which, in binary, looks like a string of `B.ncols` 1s. This type of expression with a bitwise AND with a string of ones amounts to getting the value of the first few bits of the word. The bitwise operation of this value with a row in the orthogonal basis as a boolean is checking for the smallest index of k where the lowest-order `B.ncols` bits are all 0.

For example, for this orthogonal basis for a length 8 code: We see that past the 2nd row, the



0000**11101000**
0000**10010000**
1001**00000000**
1010**00000000**
1100**00000000**

Figure 2.4.7: Orthogonal Basis with first 8 bits highlighted

first 8 bits are all zero, and so the value of k for this would end up being 3.

Thus, with this knowledge, we sum over all the rows `k` and below to get the initial candidate for a word to add for a potential child code:

```
j = k
word = 0
while ortho_basis[j]:
    word ^= ortho_basis[j]
    j += 1
```

Next:

```
log_2_radix = 0
# find smallest power of 2 greater than sizeof(int) * 8
while ((<codeword>1) << log_2_radix) < <codeword>self.radix:
    log_2_radix += 1
```

In the initialization of the BinaryCodeClassifier class, self.radix is defined as

`self.radix = sizeof(int) << 3`.

Using the `log_2_radix` value, we determine the size of the `orbit_checks` array and initialize each entry to be 0.

```
if k < log_2_radix:
    orb_chx_size = 0
else:
    orb_chx_size = k - log_2_radix
orbit_checks = <codeword *>
sig_malloc(((<codeword>1) << orb_chx_size) * sizeof(codeword))
```

```
for temp from 0 <= temp < ((<codeword>1) << orb_chx_size):
    orbit_checks[temp] = 0
```

Before the main loop begins, these values are set:

```
combo = 0
parity = 0
gate = (<codeword>1 << B.nrows) - 1
k_gate = (<codeword>1 << k) - 1
nonzero_gate = ( (<codeword>1 << (n-B.ncols)) - 1 ) << B.ncols
radix_gate = (((<codeword>1) << log_2_radix) - 1)
```

The values of `gate`, `k_gate`, and `radix_gate` will look like a string of 0s followed by a string of 1s, like 00000111.

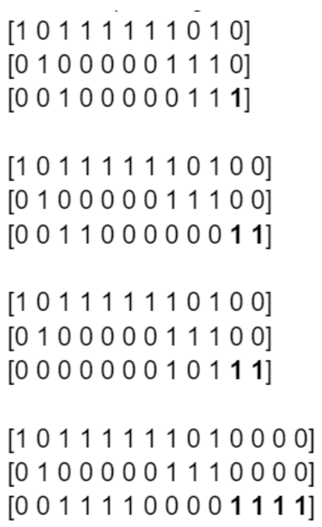
However, `nonzero_gate` is right-shifted again, so it will have padding 0s in the low-ordered bits as well. An example of a value of `nonzero_gate` could be 11000000.

Importantly, `nonzero_gate` will be used to make sure the word being considered does not have any zeros in the indices from `B.ncols+1` to `n`, since the original parent code is of length `B.ncols`, it would have 0s in those columns, and so this ensures that codes are of the desired length. For example, we see that for this parent code:

```
[1 1 1 1 1 1 1 1 0 0]
[0 1 0 0 0 0 0 1 1 1]
```

Figure 2.4.8: Parent code

Its corresponding child codes have 1s in the last row in the extra added columns:



```
[1 0 1 1 1 1 1 1 0 1 0]
[0 1 0 0 0 0 0 1 1 1 0]
[0 0 1 0 0 0 0 0 1 1 1]

[1 0 1 1 1 1 1 1 0 1 0 0]
[0 1 0 0 0 0 0 1 1 1 0 0]
[0 0 1 1 0 0 0 0 0 0 1 1]

[1 0 1 1 1 1 1 1 0 1 0 0]
[0 1 0 0 0 0 0 1 1 1 0 0]
[0 0 0 0 0 0 0 1 0 1 1 1]

[1 0 1 1 1 1 1 1 0 1 0 0 0 0]
[0 1 0 0 0 0 0 1 1 1 0 0 0 0]
[0 0 1 1 1 1 0 0 0 0 1 1 1 1]
```

Figure 2.4.9: Child codes

Ignoring the contents of the main if-statement for now, here is the structure of the main loop to generate child codes:

```
while True:
    # nonzero_gate & word == nonzero_gate: check to see if word
    # agrees with nonzero_gate at every position nonzero_gate
    # has a 1

    # (ham_wts[word & 65535] + ham_wts[(word >> 16) & 65535])%d == 0
    # for d=4, check if both halves of word are doubly-even
    if nonzero_gate & word == nonzero_gate and (ham_wts[word & 65535] + ham_wts[
                                            (word >> 16) & 65535])%d == 0:
    # contents of if-statement block
    # checking if code augmented with word is a valid child code

    parity ^= 1
    i = 0
    # every other loop
    if not parity:
        while not combo & (1 << i): i += 1
        i += 1
        # i = number of "trailing" 0s in combo + 1
    if i == k: break
    else:
        combo ^= (1 << i) #toggle the i-th bit
        word ^= ortho_basis[i]
```

Breaking this down, the loop condition is to continue until we have `i = k`. The variable `i` is calculated each iteration of the loop, and `k` was set before the loop started, and is the index of the row in `ortho_basis` that separates the words from the original code and the words added by the orthogonal basis. The expression `parity ^= 1` is toggling parity between 0 and 1, so we have that every other iteration, we evaluate this block:

```
while not combo & (1 << i):
    i += 1
i += 1
```

The boolean expression `not combo & (1 << i)` is equivalent to `combo & (1<<i)==0`, so we are testing that the i-th position of `combo` is 0. Thus this block of code will count the first `i` places where `combo` has a 0 and then add one.

Overall, this will result in iterating over all values of combo from 0 to $2^k - 1$, and the variable word will take on a value for each different combination of sums of words from the first $k-1$ rows of `ortho_basis`.

Thus we know that this loop will have one iteration for each different combination of words from the first $k-1$ rows of `ortho_basis`.

For example, here is a code in standard form and the associated orthogonal basis:

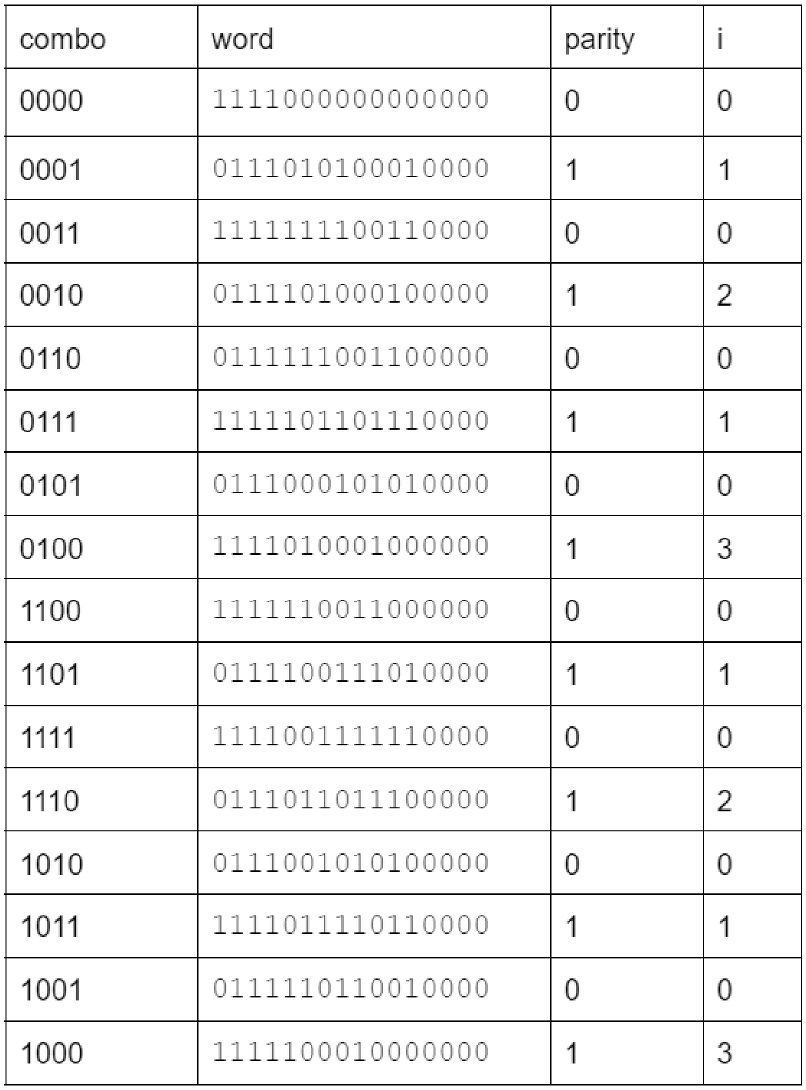
```
[1 0 0 0 1 0 1 0 0 1 0 0]
[0 1 0 0 0 1 0 0 0 0 1 1]
[0 0 1 0 0 0 1 1 0 1 0 0]
[0 0 0 1 0 0 0 0 1 0 1 1]
```

Figure 2.4.10: Code in standard form

```
1000010100010000
1000101000100000
0000010001000000
0000100010000000
1001000000000000
1010000000000000
1100000000000000
```

Figure 2.4.11: Orthogonal basis

In this example, the `n` passed into `generate_children` is 16. The resulting `k` value was 4, and the initial word is 1111000000000000. Here is how `combo, word, i`, and `parity` change with each iteration of this loop:



combo	word	parity	i
0000	1111000000000000	0	0
0001	0111010100010000	1	1
0011	1111111100110000	0	0
0010	0111101000100000	1	2
0110	0111111001100000	0	0
0111	1111101101110000	1	1
0101	0111000101010000	0	0
0100	1111010001000000	1	3
1100	1111110011000000	0	0
1101	0111100111010000	1	1
1111	1111001111110000	0	0
1110	0111011011100000	1	2
1010	0111001010100000	0	0
1011	1111011110110000	1	1
1001	0111110110010000	0	0
1000	1111100010000000	1	3

Figure 2.4.12: Values for each iteration of main loop

Now, looking at the main if statement, we see that the conditions to proceed into the main part of this while-loop are:

```
if nonzero_gate & word == nonzero_gate and (ham_wts[word & 65535] + ham_wts[(
                                        word >> 16) & 65535])%d == 0:

    temp = (word >> B.nrows) & ((<codeword>1 << k) - 1)

    if not orbit_checks[temp>>log_2_radix] & ((<codeword>1) << (temp &
                                            radix_gate)):
```

The outer if-statement checks that `word` has 1s in the last $n - B.ncols$ places, and the second part of the boolean expression checks that `word` has doubly-even weight.

Upon passing both of these if-statements, we create an augmented code by including `word` into the original code. We get the automorphism group and canonical label for this augmented code:

```
B_aug = BinaryCode(B, word)
# new code made from B and word
aug_aut_gp_gens, aug_labeling, aug_size, aug_base = self._aut_gp_and_can_label(
                                        B_aug)

can_lab = create_word_perm(aug_labeling[:n])
```

This augmented code is of length `n`, so it is longer than `B`. From this augmented code we take its basis, permute it to the canonical label, do row reduction, and then apply the inverse of the canonical label to put the columns back.

Now a copy of the basis is made, excluding the newly added word:

```
rs = []
# rs is a copy of the newly created code
# without the added word
for i from 0 <= i < B.nrows:
    r = []
    for j from 0 <= j < n:
        r.append((((<codeword>1)<<j)&temp_basis[i])>>j)
    rs.append(r)
m = BinaryCode(matrix(ZZ, rs))

m_aut_gp_gens, m_labeling, m_size, m_base = self._aut_gp_and_can_label(m)
```

With the automorphism groups for `m` and `B_aug`, we get the intersection of the two groups by converting both of them to permutation group objects:

```
if len(m_aut_gp_gens) == 0:
    aut_m = PermutationGroup([()])
else:
    aut_m = PermutationGroup([PermutationGroupElement([a+1 for a in g]) for g in
                                                     m_aut_gp_gens])

if len(aug_aut_gp_gens) == 0:
    aut_B_aug = PermutationGroup([()])
else:
    aut_B_aug = PermutationGroup([PermutationGroupElement([a+1 for a in g]) for
                                                    g in aug_aut_gp_gens])
H = aut_m._gap_(gap).Intersection2(aut_B_aug._gap_(gap))
```

Example:

Automorphism group for `B_aug`:

`[ (3,5)(6,7), (3,6)(5,7), (2,3)(4,7), (1,2)(6,7) ]`

Automorphism group for `m`:

`[(4,5), (2,4)(5,6), (1,2)(6,7)]`

Intersection of automorphism groups of `m` and `B_aug`:

`[ (1,2)(6,7), (2,6)(4,5), (2,5)(4,6) ]`

As defined by the GAP Reference Manual, given two groups G and U,

"A right transversal t is a list of representatives for the set $U\backslash G$ of right cosets (consisting of cosets Ug) of U in G."[4]

Thus the next step is to get the right transversal of H in `aut_B_aug`:

```
rt_transversal = [
    [int(a) - 1 for a in g.ListPerm(n)] for g in aut_B_aug.RightTransversal(H)
                                            if not g.IsOne()
]
rt_transversal.append(list(xrange(n)))
```

Using the above example, the resulting right transversal is

```
[
    [0, 1, 4, 3, 2, 6, 5],
    [0, 1, 5, 3, 6, 2, 4],
    [0, 2, 1, 6, 4, 5, 3],
    [0, 1, 6, 3, 5, 4, 2]
]
```

These are still permutations but are in list form instead of cycle notation.

Using this we verify whether `B_aug` is a valid child code to be included in the output of the `generate_children` function. This is done within this block of code:

```
bingo2 = 0
for coset_rep in rt_transversal:
    hwp = create_word_perm(coset_rep)
    #dealloc_word_perm(gwp)
    bingo2 = 1
    for j from 0 <= j < B.nrows:
        temp = permute_word_by_wp(hwp, temp_basis[j])
        if temp != B.words[temp & gate]:
            bingo2 = 0
            dealloc_word_perm(hwp)
            break
    if bingo2:
        dealloc_word_perm(hwp)
        break
```

We see that this function is iterating over the elements of the right transversal that was just calculated. For each permutation in the right transversal, the corresponding `WordPermutation` is created, and then we iterate over the rows of `temp_basis`, excluding the last row. If we have `temp == B.words[temp & gate]` for all the rows of `temp_basis`, then `bingo2` will be 1 after the inner loop finishes, which indicates we have found a valid child code to include as output. The augmented code `B_aug` is added to output here:

```
if bingo2:
    from matrix.constructor import Matrix
    from rings.finite_rings.finite_field_constructor import GF
    M = Matrix(GF(2), B_aug.nrows, B_aug.ncols)
    for i from 0 <= i < B_aug.ncols:
        for j from 0 <= j < B_aug.nrows:
            M[j,i] = B_aug.is_one(1 << j, i)
    output.append(M)
```

The BinaryCode object is converted into a Matrix object, which is added to the array representing the output.

The final step is to update the `orbit_checks` array:

```
orbits = [word]
j = 0
while j < len(orbits):
    for i from 0 <= i < len(aut_gp_gens):
        temp = <codeword> orbits[j]
        temp = permute_word_by_wp(parent_generators[i], temp)
        temp ^= B.words[temp & gate]
        if temp not in orbits:
            orbits.append(temp)
    j += 1
for temp in orbits:
    temp = (temp >> B.nrows) & k_gate
    orbit_checks[temp >> log_2_radix] |= ((<codeword>1) << (temp & radix_gate))
```

Recalling the if-statements guarding the main section of while-loop one of the conditions to consider an augmented code for output is

```
temp = (word >> B.nrows) & ((<codeword>1 << k) - 1)
if not orbit_checks[temp>>log_2_radix] & ((<codeword>1) << (temp & radix_gate))
```

Since orbit checks is thus based off of the orbits of child codes that have been generated by function so far, this suggests that this serves to prevent duplicate codes from being included in the output of the function.

This is the end of the while loop, we already covered the next section:

```
parity ^= 1
i = 0
# every other loop
if not parity:
    while not combo & (1 << i): i += 1
    i += 1
    # i = number of "trailing" 0s in combo + 1
if i == k: break
else:
    combo ^= (1 << i) #toggle the i-th bit
    word ^= ortho_basis[i]
```

The `generate_children` function ends with `return output`.

Thus this concludes this section explaining the text of the SageMath implementation of the algorithm.

3
Suggestions for Performance Improvements

3.1 Minor Improvements Based on the SageMath Functions

Firstly looking at how the function `self_orthogonal_binary_codes` is set up, recall that one of the parameters of the function is `in_test`, but every call to the function evaluates this if-else statement:

```
# these if-else blocks overwrite the input in_test function
if equal:
    in_test = lambda M: (M.ncols() - M.nrows()) <= (n-k)
    # test for recursion: see if parent code has size
    # similar to size of codes we want
    out_test = lambda C: (C.dimension() == k) and (C.length() == n)
    # test for output: we only output codes that have the specified
    # dimensions that we want
else:
    # maybe add:
    # if in_test != None:
    in_test = lambda M: True
    out_test = lambda C: True
```

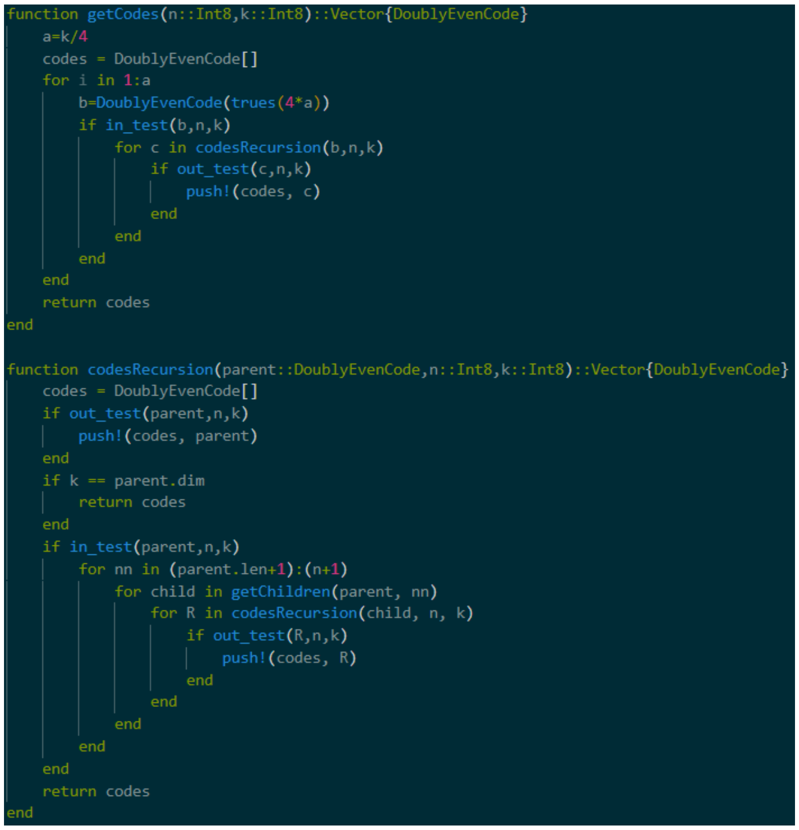
We see that no matter the value of `equal`, the value of `in_test` is being set internally. Additionally, we saw in Section 2.2 that recursive calls use the default value of `False` for `equal`, but recursive calls to `self_orthogonal_binary_codes` do specify a parent code, which results in this block being evaluated:

```
if out_test(C):
    yield C
if k == parent.nrows():
    return
# exit condition: the parent code has maximal dimension
for nn in range(parent.ncols()+1, n+1):
    # iterate over codes with degree greater than the parent
    # and leq given degree
    if in_test(parent): # we can move this if statement outside of the loop
        for child in BC.generate_children(BinaryCode(parent), nn, d):
            for N in self_orthogonal_binary_codes(n, k, d, child, BC, in_test=
                                                  in_test):
                if out_test(N):
                    # this test might be applied to some codes
                    # that have already passed it
                    yield N
```

Since in this case `equal==False`, both `in_test` and `out_test` always evaluate to true. This seems to suggest that this functionality is either not fully implemented or there was some kind of oversight and these if-statements can just be omitted. The overwriting of `in_test` could also be avoided by checking if `in_test==None` so that the parameter's value can be used.

Additionally, treating user-level calls to `self_orthogonal_binary_codes` and recursive calls to the function the same way is causing if-statement conditions to be evaluated more than they need to be. We know that recursive calls always include a parent code, so maybe a solution could involve two versions of `self_orthogonal_binary_codes`, one for user-level calls to the function and one for recursive calls to the function. The user-level version would operate the same as the function currently does, but it would make recursive calls instead to the recursive version of the function, which can make some assumptions to run more quickly.

This is the solution I came up with written in Julia:



```julia
function getCodes(n::Int8,k::Int8)::Vector{DoublyEvenCode}
    a=k/4
    codes = DoublyEvenCode[]
    for i in 1:a
        b=DoublyEvenCode(trues(4*a))
        if in_test(b,n,k)
            for c in codesRecursion(b,n,k)
                if out_test(c,n,k)
                    push!(codes, c)
                end
            end
        end
    end
    return codes
end

function codesRecursion(parent::DoublyEvenCode,n::Int8,k::Int8)::Vector{DoublyEvenCode}
    codes = DoublyEvenCode[]
    if out_test(parent,n,k)
        push!(codes, parent)
    end
    if k == parent.dim
        return codes
    end
    if in_test(parent,n,k)
        for nn in (parent.len+1):(n+1)
            for child in getChildren(parent, nn)
                for R in codesRecursion(child, n, k)
                    if out_test(R,n,k)
                        push!(codes, R)
                    end
                end
            end
        end
    end
    return codes
end
```

Figure 3.1.1: My Julia Version of `self_orthogonal_binary_codes`

Here I make the assumptions in `GetCodes` that $d = 4$, no parent is given, `equal==True`, and `in_test` and `out_test` are already defined. The function `codesRecursion` makes the same assumptions but takes a parent code as one of the parameters. I also moved the `in_test` if-statement outside of the loop, since the value of `nn` does not affect the output of `in_test`.

Looking in the `generate_children` function, the first easy improvement to make is to get rid of the section where `B_can_lab` is created, since we saw that it does not appear anywhere else in function. This could save a non-trivial amount of time since row reduction can be expensive. Also for this block of code:

```
 log_2_radix = 0
# find smallest power of 2 greater than sizeof(int) * 8
while ((<codeword>1) << log_2_radix) < <codeword>self.radix:
    log_2_radix += 1
```

Because `self.radix` is a constant and stored within the `BinaryCodeClassifier` object, it would seem that `log_2_radix` would also be a constant and could be stored within the same `BinaryCodeClassifier` object, instead of being recalculated each time `generate_children` runs.

Finally, in this section:

```
orbits = [word]
j = 0
while j < len(orbits):
    for i from 0 <= i < len(aut_gp_gens):
        temp = <codeword> orbits[j]
        temp = permute_word_by_wp(parent_generators[i], temp)
        temp ^= B.words[temp & gate]
        if temp not in orbits:
            orbits.append(temp)
```

This if-statement:

```
if temp not in orbits:
    orbits.append(temp)
```

Because `temp` is not being inserted in a specific place, and is only being added if not already in the list, this suggests that `orbits` could be implemented as a set instead of a list. Depending on the implementation (such as a hash map), searching on sets is much faster than searching on an unordered list. Especially for codes with large orbits, this could make a difference in the performance of the algorithm.

3.2 Major Possible Improvements on the Structure of the Algorithm

3.2.1 *The Gaborit Mass Formula*

As shown in "Codes and Supersymmetry in One Dimension"[3], the formula in Figure 3.2.1 gives a formula for the number of distinct doubly-even codes for a given n, k. This is different from the number of permutation equivalence classes, but it is still helpful to verify that all the permutation equivalence classes have been found.

$$\sigma(N,k) = \begin{cases} \prod_{i=0}^{k-1} \dfrac{2^{N-2i-2} + 2^{\lfloor \frac{N}{2} \rfloor - i - 1} - 1}{2^{i+1} - 1}, & \text{if } N \equiv 1, 7 \pmod 8, \\ \prod_{i=0}^{k-1} \dfrac{(2^{\frac{N}{2} - i - 1} + 1)(2^{\frac{N}{2} - i - 1} - 1)}{2^{i+1} - 1}, & \text{if } N \equiv 2, 6 \pmod 8, \\ \prod_{i=0}^{k-1} \dfrac{2^{N-2i-2} - 2^{\lfloor \frac{N}{2} \rfloor - i - 1} - 1}{2^{i+1} - 1}, & \text{if } N \equiv 3, 5 \pmod 8, \\ \prod_{i=0}^{k-2} \dfrac{2^{N-2i-2} + 2^{\frac{N}{2} - i - 1} - 2}{2^{i+1} - 1} \cdot \varpi^{+}(N,k), & \text{if } N \equiv 0 \pmod 8, \\ \prod_{i=0}^{k-2} \dfrac{2^{N-2i-2} + 2^{\frac{N}{2} - i - 1} - 2}{2^{i+1} - 1} \cdot \varpi^{-}(N,k), & \text{if } N \equiv 4 \pmod 8, \end{cases} \tag{B.1}$$

$$\varpi^{\pm}(N,k) := \frac{1}{2^{k-1}} + \frac{2^{N-2k} \pm 2^{\frac{N}{2} - k} - 2}{2^k - 1}, \tag{B.2}$$

Figure 3.2.1: The Gaborit Mass Formula for doubly-even codes

For a given code C of length n and its automorphism group $Aut(C)$, we have that the number of codes that can be reached by automorphisms of the code is $\frac{n!}{|Aut(C)|}$. Thus, the Gaborit Mass Formula is equal to the sum of $\frac{n!}{|Aut(C)|}$ for each permutation equivalence class of codes of size n, k. We have seen that in the function `self_orthogonal_binary_codes` that collects all the codes and outputs them, there is no running count or sum that keeps track of these masses for the codes that are generated, we also know that for a given parent code and a length n, `generate_children` will sometimes output no children. Thus there are some cases where, in generating codes, all the codes of a given n, k have already been generated, but the algorithm is still running!

Thus the solution I propose for a way to incorporate the mass formula to the algorithm, is to keep a global variable that is in scope of both `self_orthogonal_binary_codes` and `generate_children` functions, such that `self_orthogonal_binary_codes` will update the total for each code that is generated so far, and `generate_children` will regularly check the value of this global variable, so that it can terminate as soon as it detects that the mass formula has been saturated by the masses contributed by the already generated.

Here is an example of the mass formula calculated for the codes of length 6, dimension 2. According to Robert Miller's Website, the one code of size 6,2 is

```
    1 1 1 1 0 0
    0 0 1 1 1 1
```

For the size of the automorphism group of this code, this amounts to counting the number of different orderings of the columns. We see that the three types of columns present are $\begin{matrix}1\\0\end{matrix}$, $\begin{matrix}1\\1\end{matrix}$, and $\begin{matrix}0\\1\end{matrix}$, and there are two copies of each of these types of columns in the code. Combinatorically, this is counting the number of ways to arrange 3 groups of 2 identical things, which comes out to $\left(\frac{1}{3!}\right)\left(\frac{6*5}{2!}\right)\left(\frac{4*3}{2!}\right)\left(\frac{2*1}{2!}\right) = 15$.

Looking to the Mass Formula for $n = 6, k = 2$, we have

$$\begin{aligned}
\delta(6,2) &= \prod_{i=0}^{k-1} \frac{(2^{\frac{n}{2}-i-1}+1)(2^{\frac{n}{2}-i-1}-1)}{2^{i+1}-1} \\
&= \prod_{i=0}^{2-1} \frac{(2^{\frac{6}{2}-i-1}+1)(2^{\frac{6}{2}-i-1}-1)}{2^{i+1}-1} \\
&= \frac{(2^{\frac{6}{2}-0-1}+1)(2^{\frac{6}{2}-0-1}-1)}{2^{0+1}-1} \frac{(2^{\frac{6}{2}-1-1}+1)(2^{\frac{6}{2}-1-1}-1)}{2^{1+1}-1} \\
&= \frac{(2^2+1)(2^2-1)}{1} \frac{(2^1+1)(2^1-1)}{3} \\
&= \frac{(5)(3)}{1} \frac{(3)(1)}{3} \\
&= 15
\end{aligned}$$

For another example, consider the $7, 3$ case, which also only contains one permutation equivalence class. The size of the automorphism group of the code is 168, and then $\frac{7!}{168} = 30$. Looking at the mass formula, it gives a result of 30 as well.

3.2.2 Memoization

As we saw in the implementation of `self_orthogonal_binary_codes`, every call to the function that doesn't pass in a parent code starts a bottom-up recursion starting at the small dimension 1 codes and generating codes recursively from that starting point each time. However, this means across separate calls to generate various size codes, the algorithm ends up calculating the same codes over and over again. The idea of memoization is that for recursive functions that calculate the same value multiple times, storing those already-computed values allows subsequent requests for those values to be retrieved from memory/storage instead of having to be computed again. The idea for memoization in this case would be not just temporarily storing computed codes, but storing already computed codes on the disk or other type of long term storage. We have seen that codes are generated by other codes of smaller size, but knowing exactly which size codes are necessary in order to generate all the codes of a desired size would suddenly make this idea of memoization very practical.

This leads to:

Theorem 1: Let $l(M)$ be the length of any doubly-even code M. If C is a child code of D (as generated by the SageMath implementation for generating doubly-even codes) and C has length n and dimension k, then $\dim(D) = k - 1$ and $l(D) < n$.

Proof. (Part 1: $\dim(D) = k - 1$)

Let D and C be doubly-even linear binary codes such that C is a child of D, the length of C is n, and the dimension of C is k. By definition of child code in Appendix B.4 in "Codes And Supersymmetry In One Dimension", a child code, C, of D is made up of D and x, where x is a word not contained in D. Let B be a basis for D. Because x is not in D, we know x is linearly independent from the words in B, and so we know that $B \bigcup \{x\}$ is a basis for C. Because C has dimension k and $B \bigcup \{x\}$ is a basis for C, we know $|B \bigcup \{x\}| = k$, and so $|B| = k - 1$. Because B is a basis for D and by definition of dimension, $\dim(D) = k - 1$.

(Part 2: $l(D) < n$)

Looking at the `generate_children` function, we see that the condition to consider an augmented code that might be a child made from D and x is `nonzero_gate & word == nonzero_gate`, and `nonzero_gate` is defined as `( (<codeword>1 << (n-B.ncols)) - 1 ) << B.ncols`. In the self_orthogonal_binary_codes function, we see that all calls to `generate_children` happen in this loop:

```
for nn in range(parent.ncols()+1, n+1):
    if in_test(parent):
        for child in BC.generate_children(BinaryCode(parent), nn, d):
```

The variable `nn` becomes the parameter `n` within `generate_children(parent,nn,d=4)`, and `nn in range(parent.ncols()+1, n+1)` implies that $l(parent) < l(parent) + 1 <= nn <= n$. Thus we have that $nn > l(parent)$, and so looking back in `generate_children`, the name of the parameter representing the parent code is `B`, and the desired length of child codes is `n`. Because `B.ncols` is the length of the parent code, $nn > l(parent)$ implies $n > B.ncols$. Thus `n-B.ncols > 0` , and so, for the expression that defines
`nonzero_gate = ( (<codeword>1 << (n-B.ncols)) - 1 ) << B.ncols`, the inner expression `(<codeword>1 << (n-B.ncols)) - 1 )` is a string of 1s, of length `n-B.ncols`. Then, that value is shifted left by `B.ncols`. The resulting value has `B.ncols` 0s on the right, and `n-B.ncols` 1s on the left. Going back to the if-statement that determines whether a word will be considered for a potential augmented code to return as a child `nonzero_gate & word == nonzero_gate`, the bitwise AND between `nonzero_gate` and `word` will return the value of the bits in `word` for the indices where `nonzero_gate` has a 1. Thus, testing for equality between `nonzero_gate & word` and `nonzero_gate`, will ensure that for the indices from `B.ncols+1` to `n`, `word` will also have 1s in those indices. The code that `word` would be augmenting has length `B.ncols`, so we know that for all vectors in `B`, they have 0s in the indices past `B.ncols`. Thus, an augmented code made from `B` and `word` will have vectors with non-zero indices past `B.ncols`, since `word` has 1s in the indices from `B.ncols+1` to `n`. Thus we have that the augmented code made from `B` and `word` will have length `n`. In the context of the `self_orthogonal_binary_codes` function,

this means that child codes will have length between $l(parent) < nn <= n$. And so, we have $l(D) < l(C) = n$.

Thus we have shown that for any child code C of D, $\dim(D) = \dim(C) - 1$ and $l(D) < l(C)$.

✰

We now know that if we want to generate all codes of size n, k, we only need to generate the children of codes that have length between 0 and $n - 1$, and dimension $k - 1$.

To make use of this fact and implement it as part of the algorithm, the function `self_orthogonal_binary_codes`, given n, k to generate, would first check on disk to see if the codes of that size have already been generated, otherwise it would recursively generate the children based off of the codes of dimension $k-1$ and all lengths less than n. The Gaborit Mass Formula could also be tied into this, where, for partially completed n, k codes, the mass of codes generated so far could be also stored on disk to track progress.

3.2.3 Parallelization

Looking at Figure 2.2.1, it is noticeable that this algorithm branches quite quickly. However, each branch of the algorithm is independent in the way that the output of two functions generating the children of two codes simultaneous would not be affected by one another, and their outputs do not overlap. Also, we know recursion depth is limited by the size of codes, which we are limiting to $n = 32$ for the purposes of this project, so what makes it so time consuming is the breadth of function calls to be evaluated. Thus having multiple threads working independently would be relatively simple and efficient for making the algorithm parallel. Especially with the codes stored on disk, the main thread can just iterate over the codes of dimension $k - 1$ and length less than n, assigning one code to each thread for that thread to generate the children of. Additionally, there are many ways to split up the workload, such as splitting based on code sizes instead of individual codes, so the breadth of parallelization could vary as needed for the specifications of the system. Consider how arithmetic the algorithm is and with potentially thousands of parallel threads possible, this could be a problem well suited to GPU computing.

4
Conclusion

Because the text in SageMath contains very little comments, the implementation can be rather opaque to decipher. Hopefully the explanation of the SageMath implementation can help others be brought up to speed on the algorithm without having to spend so much time working out the details of the algorithm from the way it is written in SageMath. We also saw that there is a lot of room for different kinds of improvements to be made, such as simple changes editing a little bit of the text, and more major restructurings of the algorithm implementing the Mass Formula, recording codes, and introducing parallel computing. As someone who spent so much time trying to understand the details of the SageMath implementation before I could even start trying to improve the algorithm, this paper is the kind of resource I wish I had, which would have helped me get started testing changes/improvements much sooner.

Appendix A
Translation to Julia and Other Resources

A.1 Why Julia was chosen

Julia is a modern, compiled programming language that is popular for mathematical computations and has strong support for parallel and concurrent programming. These aspects give it an edge over the notoriously slow Python that the SageMath implementation is written in. Additionally, decoupling this algorithm from the large system that SageMath holds and the portability of Julia means after the translation is done, it will be much easier to make changes and optimizations on a program that whose sole purpose is to produce these codes, instead of the general purpose and interlocking parts of SageMath that all depend on each other. For any future people that want to continue to work on this problem, it will be much more convenient to look at and edit an independent program than working within the SageMath environment.

A.2 GitHub Repo

Here is a link to a repository containing a transliteration from SageMath to Julia, and also includes a couple more miscellaneous files that may be useful, like a version of the `databases.py` and `binary_code.pyx` that are thoroughly commented, as well as a version of `binary_code.pyx` that is set up for tracing and prints outputs to a file.

Appendix B
Other Resources and Ideas

Some ideas/links I had floating around while working and thinking about this project.

- Table of Doubly Even Codes on Robert Miller's Website
- Some graph libraries for Julia
- CSetAutomorphisms
- Oscar.jl
- SimpleGraphAlgorithms
- Slides from a lecture by Robert Miller on Graph Automorphisms: link
- Generating Linear Spans Over Finite Fields
- Ideas:
- What are the bounds on computational complexity of the overall algorithm?
- Use some code profilers to see where the most time is spent in the algorithm
- Could a randomized algorithm be effective for this somehow?
- Is there a definitive best language to implement this algorithm?

Bibliography

[1] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah, *Julia: A fresh approach to numerical computing*, SIAM Review **59** (2017), no. 1, 65–98.

[2] Ethan Bloch, *Bard tex style files.* `https://faculty.bard.edu/bloch/tex/`.

[3] Charles F. Doran, Michael G. Faux, Sylvester James Gates Jr., Tristan Hübsch, Kevin M. Iga, Gregory D. Landweber, and Robert L. Miller, *Codes and supersymmetry in one dimension*, Advances in Theoretical and Mathematical Physics **15** (2011), no. 6, 1909–1970.

[4] *GAP – Groups, Algorithms, and Programming, Version 4.12.2*, The GAP Group, 2022.

[5] The Sage Developers, *Sagemath, the Sage Mathematics Software System (Version 9.8)*, 2023. `https://www.sagemath.org`.