CLASSIFYING CONFIGURATIONS FOR TORIC CODES

CAMERON JAY TUCKERMAN
C.TUCKERMAN@CSUOHIO.EDU

MTH 496: Senior Project
Advisor: Professor I. Soprunov, PhD

Date: May 9, 2013.

Abstract. Toric Codes, constructed using a lattice polytope P and the method developed by Hansen [5], and Generalized Toric Codes, constructed using configurations $S \subset (P \cap \mathbb{Z}^n)$ being explored by Little [6], are classes of codes from which many best-known codes are found; currently, random sampling methods are used to find these champions. We describe an optimized approach to enumerate all non-equivalent configurations used to describe these Generalized Toric Codes.

Contents

1. Introduction

Recently, there has been increased interest in the possibility of finding new, best known error correcting codes which are Generalized Toric Codes. Little, in [6], determines configurations which generate Generalized Toric Codes with poor parameters, however he does not find methods to generate codes with good parameters. Undergraduate students working at MSRI-UP, in [3] and [1], use randomized searching to generate codes — their search discovered two new best known codes. Random heuristic searching is currently the best method for finding champions which are Generalized Toric Codes.

We will discuss the basics of error correcting codes, as well as the construction of Toric Codes and Generalized Toric Codes. We also give algorithms to find all non-equivalent lattice point configurations to enumerate all possible Generalized Toric Codes.

For this project, we created optimized python methods to enumerate all Generalized Toric Codes over some finite field with given block length and dimension, and allow for the calculation of their minimum distances. We visualize the results to allow for human-guided, heuristic searching for new best known codes. Included in the appendices are the methods and classes developed for this project, as well as example output, consisting of all Generalized Toric codes of block length 49 and dimesnion 4 over $\mathbb{F}_8$, along with their respective minimum distances.

2. Error Correcting Codes

In todays technology connected world, the communication of information plays a vital role. Often times, during communication, imperfect communication channels introduce errors in the communicated information. *Error correcting codes* are the mathematical construct that encodes the to-be-transmited information, by mapping the original information to a *codeword*, in such a way that the original information can be decoded even when errors are introduced due to the communication channel.

The *alphabet*, A, is the set of symbols from which codewords are built up. Often times, since we are interested in encoding information to be transmitted by classical computers, we would like our codewords to be made up of zeros and ones. Here $A = \{0, 1\}$, which is the finite field of two elements, $\mathbb{F}_2$.

All codewords will have the same length, the *blocklength* of the code. Therefore, the codewords will all be in $\mathbb{F}_2^n$, where n is the block length. Now we can think of encoding as a map,

$$ev : \mathbb{F}_q^k \to \mathbb{F}_q^n$$

where $\operatorname{Im} ev$ is the set of all possible codewords, called the *code*, C.

The *Hamming distance*, $d(c_1, c_2)$ where c_1 and c_2 are codewords, is the number of coordinates in which the codewords differ. We can then define the *minimum distance* for our code, $d(C)$,

$$d(C) = \min\{d(c_i, c_j) \mid c_i, c_j \in C, c_i \neq c_j\}$$

Hereafter, we will be concerning ourselves with a class of codes called *linear codes*, where $\operatorname{Im} ev$ forms a subspace of $\mathbb{F}_q^k$. These codes are often referred to as $[n, k, d]$ codes, where n is the block length, k is the dimension, and d is the minimum distance. As suggested by its name, the evaluation maps of linear codes are able to represented as multiplication on the right by a $k \times n$ matrix. We will see an example of this in the archetypical example of linear codes, the Reed-Solomon Codes:

Example 2.1. *Reed-Solomon Codes*

Working over some field, $\mathbb{F}_q$, with primitive element α

$$\mathbb{F}_q = \left\{0, 1, \alpha, \alpha^n, ..., \alpha^{q-2}\right\}$$

let $\mathcal{L}$ be the space of polynomials of degree less than k, i.e.

$$\mathcal{L} = \{f \in \mathbb{F}_q[x] \mid \deg f < k\}$$

We can then define an evalution map,

$$ev : \mathcal{L} \to \mathbb{F}_q^{q-1}, f \mapsto \left(f(1), f(\alpha), ..., f(\alpha^{q-2})\right)$$

This lends itself to a natural matrix representation, using a standard basis of k-degree polynomials, $\left\{1, x, x^2, ..., x^k\right\}$, e.g. $1 + 2x^3$ is the row-vector $(1, 0, 0, 2, \cdots, 0)$,

$$ev(f) = f \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \alpha & \alpha^2 & \cdots & \alpha^{q-2} \\ 1 & \alpha^2 & \alpha^4 & \cdots & (\alpha^{q-2})^2 \\ \vdots & & & & \\ 1 & \alpha^k & \alpha^{2k} & \cdots & (\alpha^{q-2})^k \end{pmatrix}$$

C, the image of ev, is the Reed-Solomon code with block length $n = q - 1$ and degree k.

Proposition 2.2. *The Reed-Solomon code is a $[n, k, n - k + 1]$ code.*

Proof. Since distinct polynomials of degree at most k can agree in at most k points, they must disagree in at least $q - k$ spots. By construction, $n = q - 1$ so $q = n + 1$, hence any two polynomials must disagree in at least $q - k = n + 1 - k$ spots. This is the definition of the minimum distance. Clearly there exists a polynomial with k nonzero, distinct roots. □

3. Toric Codes

Consider the r-cube $K_q^r = ([0, q-2])^r$, where q is some power of a prime, and $P \subset K_q^r$, some convex lattice polytope. Consider monomials of r variables of the form $\mathbf{x}^s = x_1^{s_1} x_2^{s_2} \cdots x_r^{s_r}, s = (s_1, s_2, \cdots, s_r)$, and the polynomial vector space over $\mathbb{F}_q$

$$\mathcal{L}(P) = \text{span}\{\mathbf{x}^s \mid s \in P \cap \mathbb{Z}^r\}$$

We can define an evaluation map by mapping each polynomial f to the tuple consisting of the values of f when evaluated at all points in the algebraic torus, $(\mathbb{F}_q^*)^r$, in lexicographic order, $(\mathbb{F}_q^*)^r = \{p_1, p_2, ..., p_n\}$):

$$ev : \mathcal{L}(P) \to \mathbb{F}_q^n, f \mapsto (f(p_1), f(p_2), ..., f(p_n))$$

where $n = (q-1)^r$. The image of this map is our code, C_P. Such a code is called a *Toric Code*.

Example 3.1. *Working over $\mathbb{F}_3$, consider $P = \text{conv}\{(0,0),(0,1),(1,0)\}$. Clearly $P \subset K_3^2$, so P should define a toric code.*

$$\mathcal{L}(P) = \text{span}\{1, x, y\}$$

In general, a polynomial f in r variables can have at most $(\deg f)(q-1)^{r-1}$ zeros in $(\mathbb{F}_q^*)^r$ [7]. In our case, where the maximum degree is 1, gives us an upper bound of 2 zeros. In fact, we do have a polynomial with 2 zeros. Consider the polynomial $1+x+y$ which vanishes at both $(0,1)$ and $(1,0)$, both points in the torus $(\mathbb{F}_3^*)^2$.

Therefore C_P is a $[3,3,2]$ linear code over $\mathbb{F}_3$.

Proposition 3.2. *Reed-Solomon codes are a special case of 1-Dimensional Toric Codes.*

Proof. Recall the construction of the $[n, k, n-k+1]$ Reed-Solomon code, C, over a finite field $\mathbb{F}_q$, with block length $n = q-1$ and dimension at most n.

$$C = \left\{\left(f(1), f(\alpha), ..., f(\alpha^{q-2})\right) \mid f \in \mathcal{L}\right\}$$

where

$$\mathcal{L} = \{f \in \mathbb{F}_q[x] \mid \deg f \leq k\}$$

Now, we construct a toric code C_P, using a polytope $P = [0, k]$, the line segment from 0 to k.

$$\begin{aligned} \mathcal{L}(P) &= \text{span}\{x^p \mid p \in P\} \\ &= \{f \in \mathbb{F}_q[x] \mid \deg f \leq k\} \end{aligned}$$

Clearly, $\mathcal{L} = \mathcal{L}(P)$.

□

4. Generalized Toric Codes

In the same way that Toric Codes can be seen as a logical extension of Reed-Solomon codes, so too may we consider the extension of generalized Reed-Solomon codes, which we will now define.

Working over some field, $\mathbb{F}_q$, consider some finite set of points $S \subset \mathbb{F}_q$, where for some $k < n$, $0 < s \leq k \ \forall s \in S$, and consider the vector space spanned by the monomials $x^s, s \in S$:

$$\mathcal{L}(S) = \text{span}\left\{x^s \mid s \in S\right\}$$

This is clearly a subspace of the vector space spanned in the construction of $RS(n,k)$. From this vector space, we can construct a code, a *generalized Reed-Solomon code* in the same way in which the original Reed-Solomon Code is constructed, using the same map ev. It is this procedure we will follow to define Generalized Toric Codes.

Consider K_q^r, where q is some prime power, and some set $S \subset K_q^r \cap \mathbb{Z}^r$. Now, our vector space, $\mathcal{L}(S)$ is constructed in the expected manner,

$$\mathcal{L}(S) = \text{span}\left\{\mathbf{x}^s \mid s \in S\right\}$$

and we can define the evaluation map by evaluating each polynomial at every point in $(\mathbb{F}_q^*)^r$ in lexiocgraphic order, $(\mathbb{F}_q^*)^r = \{p_1, p_2, ..., p_n\}$, i.e.

$$ev : \mathcal{L}(S) \to \mathbb{F}_q^n, f \mapsto (f(p_1), f(p_2), ..., f(p_n))$$

where, as before, $n = (q-1)^r$.

The image of this map is a *generalized toric code*, C_S.

5. Choosing Configurations

5.1. **Equivalent Configurations.** For a linear code C with generating matrix G, if C' is a code with generating matrix G' such that there is a map from G to G' consisting only of elementary row operations, then C is said to be equivalent to C'.

Let $T(s) = v + sA$ be an affine linear transformation, where v is some vector in $\mathbb{Z}_{q-1}^n$ and A is matrix, the determinant of which is in $U(\mathbb{Z}_{q-1}^n)$, the invertible elements of $\mathbb{Z}_{q-1}^n$, and hence a member of $\text{GL}(n, q-1)$. The set of all such transformations is the Affine General Linear Group, $\text{AGL}(n, q-1)$, which forms a group under composition. For some set $S = \{s_1, s_2, \cdots, s_n\} \subset \mathbb{Z}_q^r$, can define $T.S$ to be $\{T(s_1), T(s_2), \cdots, T(s_n)\}$. Because $T \in \text{AGL}$ is invertible, it also follows that $|S| = |T.S|$. It is shown in [3] that C_S is equivalent to $C_{T.S}$

Example 5.1. *1-dimensional codes*

Consider $f(x) = t_0 + t_1x + ... + t_mx^m \in \mathcal{L}(S)$, a vector space over $\mathbb{F}_q$ with roots $\alpha_i \in \mathbb{F}_q^*$. Then the polynomial $x^v f(x) = t_0x^v + t_1x^{1+v} + ... + t_mx^{m+v}$ clearly vanishes at every α_i, hence the set of zeros of $f(x)$ and $x^v f(x)$ are the same.

Additionally consider the polynomial $f(x^a) = t_0 + t_1x^a + ... + t_mx^{am}$, $a \in U(\mathbb{Z}_{q-1})$. Then $f(x^a)$ will vanish at $\alpha_i^{a^{-1}}$, where a^{-1} is the inverse of a, which exists because a is a unit. Hence the set of zeros of $f(x)$ and $f(x^a)$ are the same.

We can combine these monomial transformations as a map $x^s \mapsto x^{v+sa}$, which is a member of $\mathrm{AGL}(1, q-1)$. *It is clear that the dimension and block length of two codes C_S and $C_{T.S}$ are equal, and we have shown that their minimum distances must be the same, hence they will generated codes with the same parameters, in fact they are equivalent.*

5.2. **"Bad" Configurations.** When considering a set from which one wants to construct a generalized toric code, one can view that set S as a subset of $(\operatorname{conv} S) \cap \mathbb{Z}^r$. Research has been done into which points from $(\operatorname{conv} S) \cap \mathbb{Z}^r$ should not be excluded in the construction of a generalized toric code. By decreasing the size of S, the dimension of the code is therefore decreased. The benefit of a generalized toric code would therefore be if there were some set S for which $d(C_S) > d(C_{\operatorname{conv} S})$. It is explored by Little in [6] when removing points is not advantageous to the resultant code. His work is partially based off of the work of Cohen on polynomials over finite fields in [4], as well as the relationship between Minkowski length of polytopes and the minimum distance of the toric code they define explored by Soprunov and Soprunova in [8].

The first remark is that for fields of sufficiently large characteristic, all polynomials factor completely. The statement is made stronger by proving that if a particular polynomial will not factor of a field, there is a finite extension of the field over which it will.

Let f be some polynomial of degree l. It has an associated factorization pattern $\lambda = 1^{a_1}2^{a_2} \cdots l^{a_l}$, where f factors into a_i irreducible polynomials of degree i (not necessarily distinct). Cohen, in [4], provides a bound for the number of polynomials in our vector space which have any specific factorization pattern, in particular when $\lambda = 1^l$ which ensures that there exists a q for which there will be polynomials in our vector space which will factor completely. This means that for sufficiently large fields, $d(C_S) = d(C_{S'})$ where S' is a strict subset of S. Generalized Toric Codes are therefore worse in general than their complete counterparts over large fields.

Even over small fields, Little finds strict subsets $S' \subset S$ for which $d(C_{S'}) = d(C_S)$ based on the Minkowski decomposition of $\operatorname{conv} S$. One can define the *Minkowski sum* of two polytopes P and Q, where $P + Q = \{p + q \mid p \in P, q \in Q\}$. In this way, one can decompose a polytope into the sum of other polytopes, called a *Minkowski decomposition*, the *Minkowski length* being the number of polytope addends in the decomposition.

The *full Minkowski length* of a polytope is defined to be the largest Minkowski length of all Minkowski decompositions of all subpolytopes of P — this term is closely related to the minimum distance based on the Soprunov-Soprunova theorem [8].

Little develops the following theorem:

Theorem 5.2. *[6] Let P be a lattice convex polygon in $\mathbb{R}^2$ of full Minkowski length l. Suppose in addition that there is a unique $Q \subset P$ which decomposes as a sum of l nonempty polygons, and that each of them is a copy of a primitive lattice segment I, so $Q = lI$. Let $S \subset P \cap \mathbb{Z}^2$ satisfy*

1. *S contains the endpoints of Q, and*
2. *The k_i and l are not all multiples of any fixed integer $j > 1$.*

Then for all primes p sufficiently large and all $h \geq 1$, letting $q = p^h$, we have

$$d(C_S(\mathbb{F}_q)) = d(C_P(\mathbb{F}_q)) = (q-1)^2 - l(q-1).$$

Moreover, for all q, there exists $h \geq 1$ such that the same statement is true if we replace q by q^h.

Therefore, its clear that when constructing S by removing points from a polytope, it would not be advantageous to remove points on Q, if some Q exists.

5.3. **"Good" Configurations.** The question then still remains how to find "good" points to remove from a polytope, i.e. which points if any can be removed to create $S \subset P \cap \mathbb{Z}^r$, such C_S has a greater minimum distance than C_P.

Undergraduate researchers using Magma, in [3] and [1], discovered codes with best known minimum distance for $[64, 8]$ codes over $\mathbb{F}_5$ and $[49, 8]$ codes over $\mathbb{F}_8$. Due to computational limitations, students were forced to randomly generate configurations S and compute their parameters — a complete survey over these fields is not computationally feasible.

6. Generating Non-Equivalent Configurations

Because a complete survey of all toric codes and their parameters is not feasible, we instead offer an alternative method to finding new, "good" generalized toric codes. For given code parameters $[n, k]$, it is possible to generate a list of all non-AGL-equivalent configurations and present a user with a representative configuration and its convex hull. A user could then heuristically search the representatives for ones that might yield a better code as compared with its convex hull.

The user is prompted for the size of the field over which calculations are being performed q, the size of the configuration k, and the dimension of the ambient space for our configuration r. Our code will therefore be a $[(q-1)^r, k]$ code.

Naively, we could generate all non-equivalent configurations with the entered paramaters as follows:

(1) Select the first configuration of k points, sorted lexicographically, denoted S_1.
(2) Generate the AGL orbit of S_1, denoted $\mathcal{O}(S_1)$.
(3) Select the element of $\mathcal{O}(S_1)$ with the smallest convex hull, denoted R_1; this is the first representative.
(4) Now, select the next configuration of k points. If it is in the previously computed orbit, $\mathcal{O}(S_1)$, choose the next configuration. Continue this process until you have some element not in $\mathcal{O}(S_1)$, denoted S_2.
(5) Like before, generate the AGL orbit of S_2, denoted $\mathcal{O}(S_2)$.
(6) Select the element of $\mathcal{O}(S_2)$ with the smallest convex hull, denoted R_2; this is the second representative.
(7) Continuing, find S_3, the next element not in any of the previously computed orbits, and generate its orbit to find R_3.
(8) Continue this process until all configurations of k points are partitioned into m orbits, and return a list of m representatives, $R_1, R_2, ..., R_m$.

Remark 6.1. *Because the set of all configurations partitions into disjoint orbits, the sum of the sizes of all the orbits will be the size of all possible combinations. Once this value is attained, we notice that all subsequent configurations will lie in one the previously computed orbits, so we can stop. The bound is in fact* $\binom{(q-1)^r}{k}$.

The user can then visualize each $R_i \in \operatorname{reps} \mathbf{S}$. In practice, this computation is still very long. An improvement can be made by only considering a subet of S and a subset of affine general linear transormations.

6.1. **Origin-Inclusive Configurations.**

Remark 6.2. *Because the affine general linear group will include all translations of the configurations, the orbit of any configuration under the action of AGL will always contain configurations containing the origin.*

Therefore, instead of considering $\mathbf{S}$, we can consider $\mathbf{S_0}$, all configurations containing the origin. We also introudce the notion of $ZAGL_S \subset AGL$, the set of all AGL transformations that bring the configurations S to a configuration containing zero:

$$\mathrm{ZAGL}_S = \{T(x) = (-s)A + xA \mid s \in S, A \in \mathrm{GL}\}$$

This can be verified as being an appropriate construction of ZAGL_S.

Proof. To generate the orbit of S only considering those configurations that contain the origin, notice the following:

(1) If a configuration contains the origin, a GL tranformation will give a configuration which also contains the origin.
(2) All configurations containing the origin can be constructed by first performing all translations of S to configurations containing the origin, and then looking at all GL-equivalent configurations of those configurations.

Let T be a translation of S to another configuration containing the origin. Its clear that we can write T as an element of AGL; $T(x) = (-s) + xI = (-s) + x$, where $s \in S$ and I is the identity element of GL. Now let G be some element of GL, which can be written as an element of AGL; $G(x) = \bar{0} + xA$, $A \in \mathrm{GL}$.

We can now use the associativity of group actions, $G.(T.S) = (GT).S$,

$$\begin{aligned} G.(T.S) &= G.\{T(p) \mid x \in S\} \\ &= G.\{(-s) + x \mid s, x \in S\} \\ &= \{G(y) \mid y \in \{(-s) + x \mid s, x \in S\}\} \\ &= \{\bar{0} + ((-s) + x)\, A \mid s, x \in S, A \in \mathrm{GL}\} \\ &= \{(-s)A + xA \mid s, x \in S, A \in \mathrm{GL}\} \end{aligned}$$

□

(N.B. ZAGL_S is not a subgroup of AGL and is different for different S.)

We can now see an imporved method of generating reps $\mathbf{S}$:

(1) Select the first configuration of $k-1$ points without the origin, sorted lexicographically. Let S_1 be that configuration plus the origin.
(2) Generate the ZAGL orbit of S_1, denoted $\mathcal{O}(S_1)$.
(3) Select the element of $\mathcal{O}(S_1)$ with the smallest convex hull, denoted R_1; this is the first representative.
(4) Now, select the next configuration of $k-1$ points, and consider that configuration plus the origin. If it is in the previously computed orbit, $\mathcal{O}(S_1)$, choose the next configuration plus the origin. Continue this process until you have some element not in $\mathcal{O}(S_1)$, denoted S_2.
(5) Like before, generate the ZAGL orbit of S_2, denoted $\mathcal{O}(S_2)$.
(6) Select the element of $\mathcal{O}(S_2)$ with the smallest convex hull, denoted R_2; this is the second representative.
(7) Continuing, find S_3, the next element not in any of the previously computed orbits, and generate its orbit to find R_3.
(8) Continue this process until all configurations of k points are partitioned into m orbits, and return a list of m representatives, $R_1, R_2, ..., R_m$.

In practice, this method has been found to be considerably faster at generating non-equivalent configurations, however is still prohibitabily slow for field size $q > 13$ or dimension $d > 2$. Note that the number of combinations including the origin is different than the total number of configurations, so the point at which we can stop searching the configuration space is different; it is in fact $\binom{(q-1)^r-1}{k}$.

6.2. **Choosing Representatives.** The choice of representatives is important because it will be these configurations which can be heuristically searched by humans. Originally, the same total lexicographic ordering was imposed on each orbit, and the first configuration was selected.

Generally, small polygons tend to generate better complete toric codes, evident because they should have a relatively greater minimum distance because of fewer zeros. We therefore now choose the first lexicograhically ordered configuration with the fewest number of lattice points in its convex hull.

7. Future Research

It would be advantageous to precompute a large number of equivalent configurations for small fields — larger fields tend not to have interesting generalized toric codes as previously discussed. This would eliminate a significant amount of time in the search of champions.

In the same manner as "codetables.de" and the small lattice polygons utility created in [2], a web application wherein multiple researchers could be searching for new champions would also increase the odds of finding a champion.

8. Acknowledgements

The author thanks Professor Ivan Soprunov for his advising of this project.

References

[1] James E Amaya, April J Harry, and Brian M Vega. A systematic census of generalized toric codes over f4, f5 and f16. *MSRI-UP 2009*, page 9, 2009.
[2] Gavin Brown and Alexander M Kasprzyk. Small polygons and toric codes. *Journal of Symbolic Computation*, 2012.
[3] Alejandro Carbonara, Juan Pablo Murillo, and Abner Ortiz. A census of two dimensional toric codes over galois fields of sizes 7, 8 and 9. *MSRI-UP 2009*, page 35, 2009.
[4] S. Cohen. Uniform distribution of polynomials over finite fields, 1972.
[5] Johan P Hansen. *Toric surfaces and error-correcting codes*. Springer, 2000.
[6] John B Little. Remarks on generalized toric codes. *arXiv preprint arXiv:1107.4530*, 2011.
[7] J.P. Serre. Lettre à m. tsfasman. *Astrisque*, pages 351,353, 1991.

[8] Ivan Soprunov and Jenya Soprunova. Toric surface codes and minkowski length of polygons. *SIAM Journal on Discrete Mathematics*, 23(1):384–400, 2009.

Appendix A. representatives.py

```
nCk = lambda n,k: int(round(
    reduce(mul, (float(n-i)/(i+1) for i in range(k)), 1)
))

def generate_all_configurations(q,l,d):
    print("Generating all possible configurations... ",end="")
    all_points = [list(i) for i in product(range(q-1),repeat=d)]
    zero = all_points[0]
    all_l_combinations = [[zero] + list(i) for i in combinations(all_points[1:],l-1)]
    print("Done!")
    return sorted([Config(q,i) for i in all_l_combinations])

def gen_configs(q,l,d):
    all_points = [list(i) for i in product(range(q-1),repeat=d)]
    zero = all_points[0]
    for i in combinations(all_points[1:],l-1):
        yield Config(q, [zero] + list(i))
```

Appendix B. groups.py

```
def Z(n):
    """
    Additive group of integers mod n
    """

    return [EC(i,n) for i in range(n)]



def U(r):
    """
    Multiplicative group of integers mod n
    """

    return [EC(i,r) for i in range(r) if gcd(i,r) == 1]



def GL(n,m):
```

```
    """
    General linear group of n x n matrices over classes of integers mod m
    """
    all_arrays = [list(i) for i in product(Z(m), repeat=n**2)]
    all_2d_arrays = [[i[j:j+n] for j in range(0, n*n, n)]  for i in all_arrays]
    all_matrices = [M(i) for i in all_2d_arrays]
    return [i for i in all_matrices if i.det() in U(m)]




def AGL(n,m,g=None):
    """
    Semidirect product of the nth direct power of classes of integers mod m with GL(n,
    """
    if g is None:
        g = [list(i) for i in product(Z(m), repeat=n)]
        gl = GL(n,m)
        return [[M([i]),j] for i in g for j in gl]
    else:
        gl = GL(n,m)
        return [[i,j] for i in g for j in gl]
```

Appendix C. convexhull.py

```
def compare(a,b):
    if a < b: return -1
    elif a > b: return 1
    else: return 0

def between(a,b,x):
    """
    Returns True if a-x-b
    """

    check_slope = (b[0] - a[0]) * (x[1] - a[1]) == (x[0] - a[0]) * (b[1] - a[1])
    check_x = abs(compare(a[0], x[0]) + compare(b[0], x[0])) <= 1
    check_y = abs(compare(a[1], x[1]) + compare(b[1], x[1])) <= 1
    return (check_slope and check_x and check_y)

def right_turn(p,q,r):
```

```
    if (q[0]*r[1] + p[0]*q[1] + r[0]*p[1] - q[0]*p[1] - r[0]*q[1] - p[0]*r[1]) < 0:
        return True
    else:
        return False

def vertices_convex_hull(P):
    points = sorted(P)
    upper = [points[0], points[1]]
    for p in points[2:]:
        upper.append(p)
        while len(upper) > 2 and not right_turn(*upper[-3:]):
            del upper[-2]

    points = points[::-1]
    lower = [points[0], points[1]]
    for p in points[2:]:
        lower.append(p)
        while len(lower) > 2 and not right_turn(*lower[-3:]):
            del lower[-2]

    return upper + lower[1:-1]

def contained_in_polygon(x,V):
    if len(V) == 2:
        return between(V[0],V[1],x)

    for i in range(len(V[:-1])):
        a, b = V[i], V[i+1]
        if between(a,b,x):
            break
        if not right_turn(a,b,x):
            return False
    a, b = V[-1], V[0]
    if between(a,b,x):
        return True
    if not right_turn(a,b,x):
        return False

    return True

def convex_polytope(V):
```

```
    xs = sorted([p[0] for p in V])
    ys = sorted([p[1] for p in V])
    minx, maxx = xs[0], xs[-1]+1
    miny, maxy = ys[0], ys[-1]+1
    possible_points = [[x,y] for x in range(minx,maxx) for y in range(miny,maxy)]
    return [p for p in possible_points if contained_in_polygon(p,V)]

def points(P):
    V = vertices_convex_hull(P)
    hull = convex_polytope(V)
    missing = [i for i in hull]
    for p in P:
        missing.remove(p)
    return [hull,missing]
```

Appendix D. elements.py

```
class EC:
    """
    Equivalence class of integers mod n
    """

    def __init__(self, i, n):
        self._i = i
        self._n = n

    @property
    def i(self):
        return self._i % self._n

    @property
    def n(self):
        return self._n

    def __repr__(self):
        return "<EC " + str(self.i) + " mod " + str(self.n) + ">"

    def __hash__(self):
        return hash(int(self))

    def __str__(self):
```

```
    return str(self.i)

def __int__(self):
    return self.i

def __eq__(self, other):
    if type(other) is type(self):
        if (self.n == other.n):
            if (self.i == other.i):
                return True
            else:
                return False
        else:
            return NotImplemented
    else:
        return NotImplemented

def __lt__(self, other):
    if type(other) is type(self):
        if (self.n == other.n):
            if (self.i < other.i):
                return True
            else:
                return False
        else:
            return NotImplemented
    else:
        return NotImplemented

def __add__(self, other):
    if type(other) is type(self):
        if self.n == other.n:
            return EC(int(self)+int(other), self.n)
        else:
            return
    else:
        return EC(int(self)+int(other), self.n)

def __radd__(self, other):
    return self+other
```

```
    def __sub__(self, other):
        if type(other) is type(self):
            if self.n == other.n:
                return EC(int(self)-int(other), self.n)
            else:
                return
        else:
            return EC(int(self)-int(other), self.n)

    def __mul__(self, other):
        if type(other) is type(self):
            if self.n == other.n:
                return EC(int(self)*int(other), self.n)
            else:
                return
        else:
            return EC(int(self)*int(other), self.n)

    def __rmul__(self, other):
        return self*other

    def __pow__(self, other):
        return EC(pow(self.i,int(other),self.n),self.n)

    def __neg__(self):
        return EC(-int(self), self.n)



class M:
    """
    2 dimensional matrix
    """

    def __init__(self,m):
        self._m = m

    @property
    def size(self):
        return (len(self._m),len(self._m[0]))
```

```
@property
def m(self):
    return self._m

def __repr__(self):
    return "<Matrix " + str(hash(self)) + ">"

def __hash__(self):
    t = tuple([tuple(i) for i in self])
    return hash(tuple(t))

def __str__(self):
    return str(self._m)

def __iter__(self):
    return iter(self._m)

def __getitem__(self, key):
    return self._m[key]

def __len__(self):
    return len(self._m)

def __eq__(self, other):
    return self.m == other.m

def __add__(self, other):
    if self.size == other.size:
        matrix = [[self[i][j] + other[i][j] for j in range(self.size[1])] for i in
        return M(matrix)
    else:
        return NotImplemented

def __mul__(self, other):
    if (type(other) is type(self)) or (type(other) is type([1])):
        if self.size[1] == other.size[0]:
            m = self.size[0]
            n = self.size[1]
            p = other.size[1]
            matrix = [[sum([self[i][k] * other[k][j] for k in range(n)]) for j in
            return M(matrix)
```

```
            else:
                return NotImplemented
        else:
            return NotImplemented

    def __rmul__(self, other):
        return M([[other*j for j in i] for i in self])

    def __neg__(self, other):
        return -1*self

    def det(self):
        if self.size[0] == self.size[1]:
            if self.size[0] == 1:
                return self[0][0]
            elif self.size[1] == 2:
                return self[0][0]*self[1][1] - self[0][1]*self[1][0]
            if self.size[0] == 3:
                return self[0][0]*(self[1][1]*self[2][2]-self[1][2]*self[2][1]) - self
        else:
            return NotImplemented


class FE:
    """
    Element of a Finite Field
    """

    def __init__(self,f,a,n):
        self._a = a
        self._n = n
        self._f = f

    @property
    def field(self):
        return self._f

    @property
    def a(self):
        return self._a
```

```
    @property
    def n(self):
        return self._n

    def __repr__(self):
        return str(self.a)+"^"+str(self.n)

    def __hash__(self):
        return self.i

    def __int__(self):
        if self.a == 0:
            return self.field[0]
        else:
            return self.field[1:][self.n]

    def __eq__(self,other):
        return (self.field == other.field) and (self.a == other.a) and (self.n == othe

    def __add__(self,other):
        return self.field.add(self,other)

    def __mul__(self,other):
        return self.field.mul(self,other)

    def __pow__(self,other):
        return self.field.pow(self,other)

if __name__ == "__main__":
    m1 = M([[2,0]])
    m2 = M([[2,3],[3,2]])
    print(m1*m2)
```

Appendix E. fields.py

```
class GF:
    """
    Galois Field
    """
```

```
    def __init__(self,q):
        if q in [2,3,5,7,9,11]:
            self._p = q
            self._n = 1
        elif q == 4:
            self._p = 2
            self._n = 2
        elif q == 8:
            self._p = 2
            self._n = 3
        a = prime_gen[self.q]
        self._add_table = add_table[self._p**self._n]
        self._e = [FE(self,0,0)]+[FE(self,a,i) for i in range(self.q-1)]

    @property
    def p(self):
        return self._p

    @property
    def n(self):
        return self._n

    @property
    def q(self):
        return self.p**self.n

    def __repr__(self):
        return "GF("+str(self.n)+")"

    def __hash__(self):
        return self.q

    def __iter__(self):
        return iter(self._e)

    def __len__(self):
        return self.q

    def __getitem__(self,key):
        return self._e[key]
```

```
    def mul(self,x,y):
        if x.a == 0 or y.a == 0:
            return self[0]
        else:
            return self[1:][(x.n+y.n)%(self.q-1)]

    def pow(self,x,n):
        if x.a == 0:
            return self[0]
        else:
            return self[1:][(x.n*int(n))%(self.q-1)]

    def add(self,x,y):
        if x.a == 0:
            return y
        elif y.a == 0:
            return x
        else:
            return self[self._add_table[(x.n+1)][y.n+1]]
```

Appendix F. polynomials.py

```
class Polynomial:
    def __init__(self,q,powers):
        self._q = q
        self._powers = powers
        self._field = GF(q)
        self._l = len(powers)
        self._d = len(powers[0])

    def eval(self,point,c):
        field = self._field
        result = field[0]
        for i in range(self._l):
            m = c[i]
            for j in range(self._d):
                m = m * (point[j]**self._powers[i][j])
            result = result +  m
        return result

    def max_zeros(self):
```

```
        field = self._field
        zero = field[0]

        coefficients = list(product(list(field),repeat=self._l))[1:]
        torus = list(product(list(field)[1:],repeat=self._d))

        max_num_zeros = 0
        max_cos = []
        max_roots = []

        for c in coefficients:
            num_zeros = 0
            roots = []
            for p in torus:
                if self.eval(p,c) == zero:
                    num_zeros = num_zeros + 1
                    roots.append(p)

            if num_zeros > max_num_zeros:
                max_num_zeros = num_zeros
                max_cos = c
                max_roots = roots

        return(max_num_zeros,max_cos,max_roots)
```

Appendix G. configurations.py

```
class V:
    """
    Position vector of a point in a configuration
    """

    def __init__(self,q,p):
        self._q = q
        if isinstance(p,M):
            self._v = p
            self._p = [i.i for i in p[0]]
        else:
            self._p = p
```

```
        self._v = M([[EC(i,q-1) for i in p]])

def __repr__(self):
    return "<V " + str(self._v) + ">"

def __hash__(self):
    return hash(tuple(self._p))

def __str__(self):
    toreturn = ""
    for i in self:
            toreturn = toreturn + str(i) + ","
    return "(" + toreturn[:-1] + ")"

def __iter__(self):
    return iter(self._v[0])

def __getitem__(self, key):
    return self._v[0][key]

def __len__(self):
    return len(self._v[0])

def __eq__(self, other):
    return (self._v == other._v)

def __lt__(self, other):
    if len(self) < len(other):
        default = True
        l = len(self)
    else:
        default = False
        l = len(other)
    for i in range(l):
        if self[i] < other[i]:
            return True
        elif self[i] > other[i]:
            return False
    return default

def __rmul__(self, other):
```

```
        return V(self._q,other[0] + (self._v * other[1]))

class Config:
    """
    Configuration of points
    """

    def __init__(self,q,P):
        self._q = q
        if isinstance(P[0],V):
            self._V = sorted(P)
            self._P = [v._p for v in self._V]
        else:
            self._P = sorted(P)
            self._V = [V(q,p) for p in self._P]
        self._d = len(self._P[0])

    def __repr__(self):
        return "<Config " + str(self._V) + ">"

    def __hash__(self):
        return hash(tuple(iter(self)))

    def __str__(self):
        toreturn = ""
        for i in self:
            toreturn = toreturn + str(i) + ","
        return "{" + toreturn[:-1] + "}"

    def __iter__(self):
        return iter(self._V)

    def __getitem__(self,key):
        return self._V[key]

    def __len__(self):
        return len(self._V)

    def __eq__(self,other):
        return (self._V == other._V)
```

```
def __lt__(self, other):
    if len(self) < len(other):
        default = True
        l = len(self)
    else:
        default = False
        l = len(other)
    for i in range(l):
        if self[i] < other[i]:
            return True
        elif self[i] > other[i]:
            return False
    return default

def __rmul__(self,other):
    return Config(self._q,[other*v for v in self])

def orbit(self,gl=None):
    if hasattr(self,"_orbit"):
        return self._orbit

    if gl is None:
        gl = GL(self._d,self._q-1)
    translations_to_zero = [-1*(i._v) for i in self]
    ZAGL = [[i*j,j] for i in translations_to_zero for j in gl]
    self._orbit = sorted(list(set([g*self for g in ZAGL])))
    #for c in self._orbit: c._orbit = self._orbit
    return self._orbit

def aglorbit(self):
    if hasattr(self,"_aglorbit"):
        return self._aglorbit

    agl = AGL(self._d,self._q-1)
    self._aglorbit = sorted(list(set([g*self for g in agl])))
    for c in self._aglorbit: c._aglorbit = self._aglorbit
    return self._aglorbit

def aglorbit_gen(self):
    agl = AGL(self._d,self._q-1)
    for g in agl:
```

```
            yield g*self

    def missing_points(self):
        if self._d == 2:
            return convexhull.points(self._P)[1]
        else:
            return None

    def complete_configuration(self):
        if self._d == 2:
            return Config(self._q,convexhull.points(self._P)[0])
        else:
            return None

    def always_generalized(self):
        if self._d != 2: return None

        if len(self.missing_points()) == 0:
            return False
        for i in self.aglorbit_gen():
            if len(i.missing_points()) == 0:
                return False
        return True

    def size_convex_hull(self):
        return len(self._P) + len(self.missing_points())

    def z(self,regen=False):
        if hasattr(self,"_zeros") and not regen:
            return self._zeros
        polynomial = Polynomial(self._q,self._P)
        self._zeros = polynomial.max_zeros()
        return self._zeros
```

Appendix H. Example Output

All Generalized Toric Codes of block length 49 and dimension 4 over $\mathbb{F}_8$, see below.