0.1. **About these notes.** These notes might be a little incomplete, but do touch on all of the material that has been covered in class so far. There are a few exercises scattered throughout the notes. I don't expect these to be written up and turned in, but I am happy to discuss them. (Some are repeated on a homework sheet, and for those I am expecting solutions.) If an exercise has an asterisk or two next to it that means that it may be difficult and/or vague.

0.2. **A warning about notation.** Notation varies a bit for linear codes, especially for things like generator matrices, check matrices, and words/vectors/strings/messages. In this course, a generator matrix for an (n, m) linear code, which has length n and rank (or dimension) m is always a $n \times m$ matrix of full rank, a check matrix always has n columns, and a word/vector/string/message is always a "column vector", even if we sometimes write something like $v = (0, 0, 1, 0, 1, 1, 0)$ or $w = 00101101$. In particular for a generator matrix G and a check matrix H, we encode with the multiplication Gv and check with the multiplication Hw, and $HG = 0$.

Other authors might transpose things in different ways, and may end up writing wH or vG or $Hw^\intercal$ or $Gv^\intercal$.

1. Introduction: Error detection and correction

1.1. **Motivational stuff. What is this all about?** Error correcting codes are a method of encoding data in a way so that errors in transmission can be automatically corrected.

Consider the following example. When you talk on a mobile phone, sometimes when there is a poor signal or some other sort of interference, there is lots of static in the transmission. Words may be garbled and impossible to understand. However, you have probably never received a text message or an email which was improperly transmitted. Why is this? The connection that your phone uses to send text messages suffers from the same static problems as the voice connection. The connection may be very poor, and yet, when a text message is sent and received we can be pretty certain that the message which is sent is the same as the message which is received.

We might at first think that this kind of reliability comes from simply repeating the message until it is correctly received. The receiver can acknowledge correct transmission by sending back a message in return; however, we still have a problem: how is the receiver to know that the message was correctly received? A computer sends a sequence of ones and zeros; a computer receives a sequence of ones and zeros; how are we to know that they are the same?

The problem is exacerbated when it takes a long time to send a message, or when the message can only be sent once. It takes about seventeen and a half hours for a message sent by the Voyager 1 probe to reach Earth (and another seventeen and a half for a response.) When we store data on a computer hard drive, write it once and expect it be the same when we read it later. (We can think of this as sending a message through time.) In these cases, it is cumbersome or impossible to request retransmission, and we must take additional steps to ensure correct transmission on the first try.

In the first case, then, we are concerned with error detection. In the second case, when we know that an error has occurred, we are concerned with error correction. We will see that these problems are similar, though error correction is somewhat more complicated. Both

problems are dealt with by choosing beforehand a specific set of allowable messages; if this set of legal messages is chosen carefully, we will be able to automatically correct many errors that occur in practice.

In short, then, the theory of error detecting and correcting codes is the theory of how to encode arbitrary information in a special way, and then to decode it, and to do all of this in such a way that the intermediate format is robust against errors from noise or degradation.

1.2. **First examples.** simple parity check. repetition. multidimensional parity check.

In these first examples, the symbols that we use in our code are 0s and 1s. We think of these symbols as living in the field with two elements, $\mathbb{F}_2$, so that

$$0+0=0$$
$$0+1=1$$
$$1+1=0$$

and

$$0\times 0=0$$
$$0\times 1=0$$
$$1\times 1=1.$$

Example 1.1 ($(8,7)$-parity check). Take 7 bits and add one more so that the total number of ones is 8, e.g.

$$0010101 \longrightarrow 00101011.$$

Or in general

$$e_0, e_1, \ldots, e_6 \longrightarrow e_0, e_1, \ldots, e_6, \sum e_k.$$

The receiver checks if the received word has an even number of ones. If it does not, it knows that an error occurs. By itself, this is only an error detection code, and it can detect any odd number of errors. In the presence of some extra information — namely, the location where an error occurred — this code can be used to correct errors as well. For example, on a computer, data may be spread out over multiple hard drives; using a parity check like this, if each parity bit is stored on a different hard drive then any single hard drive can fail and no data will be lost.

Example 1.2 (repetition code). A simple method to try to make sure that a message gets through is to repeat it. If just one extra copy of a message is sent, then it can be used to detect errors, but not correct errors. If there are two extra copies, then a "majority vote" can be taken. We encode

$$0 \longrightarrow 000, \qquad 1 \longrightarrow 111.$$

If a receiver sees the message 010 it can make a pretty good guess that the message should have been 000; in this way, the triple repetition code can correct any single error.

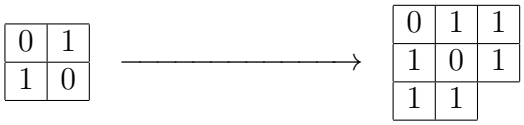
Example 1.3 (Multidimensional parity check). We can encode a message by imagining it in a rectangular grid and then adding parity checks for each row and each column in the grid. For example, we can encode a message of length 4 with a 2×2 parity check:

e_0	e_1
e_2	e_3

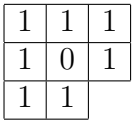
$\longrightarrow$

e_0	e_1	e_0+e_1
e_2	e_3	e_2+e_3
e_0+e_2	e_1+e_3	

e.g.,



Suppose that computer receives the encoded message



When we check all of the additional parity bits, we will notice that there is an error in the topmost row and the leftmost column. There is one entry in the intersection of those two columns, so changing it from a 1 to a 0 will make all of the parity checks pass. We should also check what happens when one of the check bits is mistransmitted: in such a case, only a single row or column will contain an error, so we will know that the error occurred in a check bit and not in a message bit.

So this code can correct any single error. In contrast with the triple repetition code, which will always miscorrect any double error, this code will also correctly notice the presence of many errors instead of misrecognizing them as single errors and miscorrecting them.

It is also clear how to generalize this code to a $k \times l$ parity check for any k and l: we simple think of the data is being put into a k by l array. Then the code takes a message of length kl and encodes it as a message of length $kl + k + l$. We can also generalize to higher dimensions and define a $k_1 \times k_2 \times \cdots k_d$ multidimensional parity check.

Exercise 1.4. For a message encoded with the 2×2 parity check, write down an example of (i) a double error that is not miscorrected and (ii) a double error which may be misinterpreted as a single error, and hence miscorrected.

***Exercise* 1.5. Think about a d-dimensional parity check. It may be a bit cumbersome to write down — maybe you can find a nice way to do it. How many errors can it correct? How many extra bits does it add to encode a message?

1.3. **Some brief formal stuff.** alphabet. code. encoder. decoder. minimum distance. error detection. error correction. rate. rank. length.

Definition 1.6 (Alphabet)**.** *An alphabet A is a set of symbols.*

For most of this course, we will use the binary alphabet $\mathbb{F}_2$, which consists of the symbols 0 and 1. We write this as $\mathbb{F}_2$ because it is very convenient to think of it as the field with 2 elements. In general, for any q which is a prime power, $q = p^k$, there is a finite field with q elements; these are nice alphabets to use in theory.

In "real life", the alphabet used for a code will often be $\mathbb{F}_{2^k}$ for some k, since it is usually most convenient for computers to work with things which are powers of 2. However, the alphabet is taken to be the digits 0 through 9 in some real examples, and can in principle be anything.

Definition 1.7 (Code)**.** *A (block) code C of length n with alphabet A is a subset of $A^n = A \times A \times \cdots \times A$; that is, it is a set of strings of symbols of length n. We call such strings of symbols a* string *or a* word *or sometimes a* vector*. We refer to a word which is in the code as a* codeword*. (Whether or not a given string is a codeword depends on context, of course.)*

To make use of a code, we often want a way to map a message to a codeword, and a codeword back to a message. We call such functions an encoder and a decoder.

Definition 1.8 (Encoder, Decoder, Rank)**.** *An* encoder $f(v)$ *for a code C of length n is an injective function from A^m to C, where $m \leq n$, and a corresponding* decoder $g(u)$ *is a function which inverts this. That is,*

$$f : A^m \longrightarrow C$$

and

$$g(f(v)) = v.$$

For a given code C, the largest m for which there exists an encoder with domain A^m is called the rank *of the code.*

We do not enforce any uniqueness on the encoder or decoder. For a given code there may be multiple encoders and decoders. The domain and range of the decoder are also not fully specified. The decoder need only be defined on the range of f. For the sorts of codes that we are interested in here, though, we should think of the decoder as being defined on all of A^n and taking values in some set $A^m \cup E$; the decoder may *detect* errors by returning some string which is not in A^m, or it may *correct* errors by returning an element of A^m even when the input to the decoder is not a code word.

One goal will be to describe codes for which there exist decoders with good error detection and correction properties. We have in mind a model where a codeword is transmitted by some method and an error occurs: some of the transmitted symbols become changed in some way. Exactly what sort of error might occur is a subtle question, but a simple model to have in mind is that every transmitted symbol might have some fixed probability of being transformed into a random symbol, and that these probabilities are all independent. (We might call this *white noise*.)

When an error occurs, we would, first of all, like to know the decoder to "know" that an error has occurred. Moreover, we would like to correct this error, if possible. For the decoder to know that an error has occurred simply means that the received word is not a codeword. In the white noise model, we can see that a decoder might try to correct an error by finding a codeword which is close to the received word, where white noise suggests that we should consider two words as close if they have few different symbols. This distance is called the Hamming distance.

Definition 1.9. *For two strings $u = (u_0, u_1, \ldots, u_{n-1})$ and $v = (v_0, v_1, \ldots, v_{n-1})$, the Hamming distance $d(u, v)$ is the number of entries in which they differ; that is, it is the number of k for which $u_k \neq v_k$; that is*

$$d(u, v) = \#\left\{k \text{ such that } u_k \neq v_k\right\}.$$

With a notion of distance, we can begin to think more about what it means for a code to be capable of detecting and/or correcting errors. Error detection is a fairly simple matter: if the input to the decoder is not a codeword, an error has occurred. Error correction may be more subtle, but if we think that errors do not occur too frequently, then it is reasonable way

to correct errors is for the decoder to return the message that corresponds to the codeword which is closest to the received string. [1]

For any code and encoder, then, we can define the *minimum distance decoder*[2], which is a decoder which does exactly what we just described.

Definition 1.10. *For a code C with encoder $f(v)$, a minimum distance decoder $d(u)$ is a function from A^n to $A^m \cup E$ such that*

$$d(u) = v \text{ such that } d(f(v), u) = \min_{c \in C} d(c, u)$$

when such a v exists and is unique, and $d(u) \in E$ otherwise.

The notion of a minimal distance decoder is a good thing to have in the back of one's mind, but it is not something we'll work with again, and we will hardly mention it again. For a given code, minimal distance decoding for every possible word is typically an extremely difficult algorithmic problem. However, it is useful, for example, for the purposes of thinking about the best possible performance that a code could have in a worst case scenario.

When it comes to a worst case scenario, it is useful to measure a code by the number of errors is can correct or detect. To measure this, we want to know: if a number of errors occur in the transmission of a code word, will it be mistaken for another code word? This makes the notion of minimal distance a useful one for study.

Definition 1.11 (Minimal distance)**.** *The minimal distance d of a code C is the integer*

$$d = \min_{\substack{u \neq v \\ u,v \in C}} d(u, v).$$

If a code word c is transmitted and d errors occur, then it is possible that the received word will actually be another code word; if more than $d/2$ errors occur, then a minimal distance decoder may mistake the received word for the wrong code word. For these reasons we say that a code with minimal distance d can always detect the presence of $d-1$ errors; meanwhile, if at most $(d-1)/2$ errors occur, then these errors can be corrected by replacing the received word with the closest code word, so we say that this code can correct $\lfloor (d-1)/2 \rfloor$ errors.

2. Linear codes

field. vector space. linear code. generator matrix. check matrix. error syndrome...

We have already noted that it is useful for our alphabet to have some nice structure. Even in our first examples, we used the operation of addition modulo 2. We will now start to think of our alphabet as a finite field. Recall that a field is a set with addition and multiplication such that every element has an additive inverse and all the nonzero elements have a multiplicative inverse.

The most familiar finite fields are the integers mod a prime: $\mathbb{Z}/p\mathbb{Z}$. There is also a finite field with q elements whenever $q = p^k$ is a prime power. Most of the theory of this section

[1] Specifically, if the probability of an error occurring in the transmission of a single symbol is p, we imagine that p is rather small, and especially avoid the pathological situation where $p >= 1/2$.

[2] For the model that we just informally described, this might also be called a maximum likelihood decoder

will apply when the alphabet is any of these fields. In practice, because it is convenient for computers to work with powers of 2, fields with 2^k elements are commonly used.

Most of the examples will use the field $\mathbb{F}_2$, with two elements, and some constructions will rely on the use of this field. However, we will eventually find it useful to use fields with more elements even in the construction of binary codes.

Now, when our alphabet is a field F, the set of strings F^n has the structure of a vector space, which means that we can add two strings or multiply a string by an element of F. Once we can do this, a useful notion is that of a linear code, which is simply a code which is also a subspace of F^n.

Definition 2.1. *A* linear code C *of length n and rank (or dimension) m over a finite field F is a subset of F^n of size $|F|^n$ such that $w_1 + w_2$ and aw are codewords whenever w_1, w_2, and w are codewords and $a \in F$.*

In other words, an (n, m)-linear code is a vector subspace of F^n of dimension m.

There are two natural ways to specify an m-dimensional subspace of an n-dimensional vector space which are useful first notions for constructing and analyzing linear codes. The first is to specify an injective linear map from F^m to F^n, or, more concretely, an $n \times m$ matrix of rank m. The second is to specify the subspace as the kernel of a surjective map from F^n to F^{n-m}, or, again more concretely, as the (right) kernel of an $(n-m) \times n$ matrix with rank $n-m$. We call the first sort of matrix a generator matrix, and the second a check matrix.

Definition 2.2. *A* generator matrix G *for an (n, m)-linear code C is a matrix such that $w \in C$ if and only if $w = Gv$ for some $v \in F^m$.*

Definition 2.3. *A* check matrix H *for an (n, m)-linear code C is a matrix such that $w \in C$ if and only if $Hw = 0$.*

Perhaps some examples will be illustrative. The $(8, 7)$-parity check code, the triple repetition code, and the multidimensional parity check code are all examples of linear codes.

Example 2.4 ($(8, 7)$-parity check). The $(8, 7)$-parity check has a generator matrix

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

and check matrix

$$H = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

Example 2.5. The triple repetition code has a generator matrix

$$G = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

and check matrix

$$H = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}.$$

Example 2.6. The linear code with check matrix

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

and check matrix

$$H = \begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

is equivalent to the 2×2 parity check code.

In this last example our wording was slightly different than in the first two examples. When we gave the example of the 2×2 parity check, we did not write the check bits in any specific order. It should be clear that it makes no difference in which order we place these bits, even if the issue of exactly what it means for two codes to be equivalent might be more subtle.

Note that the generator and check matrices for a code are not unique. For the parity check matrix, for example, we could instead use the generator matrix

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix},$$

or the matrix

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix},$$

which both map $\mathbb{F}_2^7$ to the exact same subset of $\mathbb{F}_2^8$ as the previous matrix. Similarly, there may be many different check matrices for a fixed code. In fact, it should be a straightforward linear algebra exercise to see, for example, that

Proposition 2.7. *Any $n \times m$ matrix G of rank m whose columns are code words for the (n, m) linear code C is a generating matrix for C.*

Similarly, any $(n-m) \times n$ matrix H of rank $(n-m)$ for which $HG = 0$ is a check matrix for C.

Exercise 2.8. Make sure that you understand why the above statements are true.

However, it is nice when we have a generator matrix which consists of the identity matrix at the top, because then if no errors occur the message may be decoded by simply looking at the first m symbols in the code word. We say that such a matrix a *systematic* generating matrix. When we have a systematic generating matrix, it is particularly easy to write down a corresponding check matrix.

Theorem 2.9. *If a linear code is specified by a generating matrix G in the form*

$$G = \left(\frac{I_m}{A}\right),$$

where I_m is the $m \times m$ identity matrix, then a check matrix for the code is given by

$$H = (-A|I_{n-m}).$$

Proof. This is just an exercise in matrix multiplication. H has the correct rank, so we just need to check that $HG = 0$. □

2.1. **Minimum distance for linear codes.** Calculating the minimum distance of a code can be a rather difficult problem. If a code has no structure at all, we might think that we should simply check each pair of codewords and find the distance between them. Fortunately, for linear codes the situation is not quite so bad as that, though the general problem of finding the minimal distance of a linear code can still be quite hard.

Two basic facts give us something to work with.

Theorem 2.10. *The minimum distance of a linear code is the same as the minimal weight of a nonzero vector. That is,*

$$\min_{\substack{u,v \in C \\ u \neq v}} d(u, v) = \min_{\substack{w \in C \\ w \neq 0}} \mathrm{wt}(w)$$

If this were the whole story, we might still have to write down every codeword to determine the minimum distance. This might be acceptable for small codes, but will get rather cumbersome for very large codes. [3]

Theorem 2.11. *The minimum distance d of a linear code with check matrix H is the least integer such that every $d-1$ columns of H are linearly independent and some d columns are linearly dependent.*

[3]Food for thought: suppose that a binary code has rank 200, so that it has 2^{200} different codewords. Will a computer ever be able to write down all the different codewords?

Note that for binary codes, a set of vectors is linearly independent if and only if no subset of these vectors sums to zero. If we examine the check matrix we wrote down earlier for the 2×2 parity check code, for example, we can see that no two columns are the same, so the minimum distance is at least 3. It is not hard to also find three different columns which sum to zero, so the minimum distance is exactly 3.

Even given a check matrix, it gets very difficult to determine the minimum distance of a linear code quickly as the matrix size grows, unless perhaps if the minimum distance happens to be very small. This is not a problem we will be able to solve in general (in fact, it is not a problem that anyone knows how to solve in general), but at the very least it is possible to exploit the above theorem to construct codes with large minimum distance by writing down check matrices in clever ways.

2.2. **Error syndromes.** We now turn our attention briefly to the problem of how to actually correct errors when they occur. In general, when we receive a message which is not a codeword, we would like to hope that only a small number of errors has occurred, so the general problem is: Given a vector $v \in F^n$ which is not a codeword, can we find the codeword which is closest to v, or else determine that there is no such unique vector?

Once again, the structure of linear codes gives us something to work with. We should think of our incorrect vector as v as

$$v = c + e,$$

where c is a codeword and e is the error vector. There are many different possible representations of v in this form, and we seek the ones where $\mathrm{wt}(e)$ is as small as possible. We will know that v is not a codeword because $Hv \neq 0$. Now, if we write $v = c + e$, then we have

$$Hv = H(c + e) = Hc + He = He,$$

as $Hc = 0$ for the codeword c. Thus we see the important principle that the result of multiplying the check matrix by the received string depends only on the error and not on the codeword that v should have been. We give a name to this quantity:

Definition 2.12 (Error syndrome)**.** *The* error syndrome *of a vector v and check matrix H is the vector Hv.*

We emphasize the point we made above as

Theorem 2.13. *The error syndrome of a vector v depends only the error. That is, $Hv = H(v - c)$ for any codeword c.*

Let's look at an example before proceeding too much further. We return to the check matrix that we wrote down earlier for the 2×2 parity check code

$$H = \begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

and the example error correction that we gave when we first wrote down this code. We examine the vector $v = (1, 1, 1, 0, 1, 1, 1, 1)$, and compute the error syndrome $Hv = (1, 0, 1, 0)$. We now want to find a vector of small weight which has the same syndrome. This is easy to do — as the syndrome is a column in the check matrix we can easily see that

$e = (1, 0, 0, 0, 0, 0, 0, 0)$ has exactly the same syndrome. Thus $c = v + e$ is a codeword, and we have found a codeword which is distance 1 away from v.

Note that for a binary code, a single error will always result in a syndrome which is a column in the check matrix, and for a code over F_q a single error will always result in a syndrome which is a multiple of a column in the check matrix.

In general, if we want to correct a double error, we may need to examine the syndromes for all vectors of weight 2; if we want to correct three errors we may need to examine the syndromes for all errors of weight 3, and so on. Once again, we run into trouble fairly quickly, but the situation is not too bad. A decoder might store a table of error syndromes for vector of small weight which will allow it to quickly do a table lookup. For example, for a binary code of length 128 there are 349632 different nonzero vectors of weight ≤ 3, and a computer should be able to keep a table of these in under 50 megabytes. This suggests that it is reasonable for a computer to correct 3 errors in a length 128 just by looking up error syndromes in a table.

As the code length and the number of errors that we want to correct grows, however, the number of different possible error syndromes will grow much too fast for such simple methods to be of use, and we will need other methods of error correction. I hope we'll examine some small examples of clever constructions of check matrices which allow for better algorithms, but we will not have much time to look deeply at the subject of practical error correction algorithms.

2.3. **Equivalence of codes.** We have been a little loose so far when we speak of something like "the" 2×2 parity check code. When we first gave the example of the code by drawing a picture, it was clear what subset of pictures we were talking about. However, when we want to consider this code as a subset of $\mathbb{F}_2^8$, we need to choose some ordering of the message bits and check bits. It should be clear that the ordering of the bits isn't of any fundamental importance.

For this reason we call two linear codes equivalent equivalent if it is possible to move from one to the other by permuting the positions of the code and multiplying all of the symbols in some position by some fixed element of the base field. In terms of the generator matrix for the code, this means that permuting the rows of the generator matrix gives an equivalent code and multiplying one of the rows of the generator matrix by an element of the base field also gives an equivalent code.

We don't focus on this too much, but for when we speak of "the" $k \times l$ parity check code or "the" Hamming code (in the next section) we are really referring to an equivalence class of codes.

2.4. **Hamming codes.** hamming codes. generator and check matrix. how to correct errors. extended hamming code.

We now continue our examination of linear codes by considering what the best sort of single error correction might look like.

Definition 2.14. *The* $(2^k - 1, 2^k - k - 1)$ *binary Hamming code is the code with alphabet* $\mathbb{F}_2$ *determined by the* $k \times (2^k - 1)$ *check matrix whose columns consist of every nonzero vector in* $(\mathbb{F}_2^k)$.

Example 2.15. A $(7, 4)$ Hamming code with check matrix in standard form is given by

$$H = \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}.$$

The Hamming code has minimal distance 3, so it can correct any single error. An interesting property of the Hamming code is explored in the following exercise.

Exercise 2.16. Prove that any vector in $\mathbb{F}_2^{2^k-1}$ is either a code word for the $(2^k-1, 2^k-k-1)$-Hamming code or else is distance one away from a code word.

Hint: This can be done by counting; first count the number of codewords, then count the number of things that are distance one away from a codeword, and then count the number of vectors in $\mathbb{F}_2^{2^k-1}$. How are they related?

This may be a good opportunity to point a rather general method for easily increasing the minimal distance of a code in some situations. Suppose we wish to modified the above check matrix to get something with minimum distance 4. It is already the case that any pair of columns is distinct, so we need to make it so that any three columns do not sum to zero. Suppose that we had a row of all 1s: then only an even number of columns can sum to zero. Thus, the check matrix

$$\begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

gives a code with minimum distance 4. This extra column made the code smaller by removing many code words. However, we can easily augment this matrix now by adding a new row:

$$\begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

This is known as the $(8, 4)$ extended Hamming code. More generally,

Definition 2.17. *A $(2^k, 2^k - k - 1)$ extended Hamming code is a code determined by a check matrix H of size $(k + 1) \times 2^k$ with a row of all 1s and which, when removed this row is removed, consists of every (column) vector in $\mathbb{F}_2^k$.*

The extended Hamming code has an important practical advantage over the regular Hamming code. A system using the Hamming code will always "correct" any number of errors that occur; if more than one error occurs, messages will be incorrectly decoded. A decoder for the extended Hamming code, however, can notice the presence of double errors while still correcting any single error.

The construction of the extended Hamming code can be mimicked for any code with odd minimum distance.

Proposition 2.18. *If H is a check matrix for a binary code with minimal distance d, where d is odd, then the matrix*

$$H' = \begin{pmatrix} & & & 0 \\ & H & & 0 \\ & & & \vdots \\ & & & 0 \\ 1 & 1 & \cdots 1 & 1 \end{pmatrix},$$

is a check matrix for a code with minimum distance $d+1$.

Exercise 2.19. Prove the above proposition.

2.5. **Hamming codes over fields other than $\mathbb{F}_2$.** Over $\mathbb{F}_2$, two vectors are linearly independent if and only if they are different. This does not continue to hold when we consider vector spaces over larger fields. If we want to construct a code analogous to the binary Hamming code over a field that is not $\mathbb{F}_2$, then, we have to keep this in mind. Other that this difference, there is a straightforward construction of codes over other fields which mimics the construction of Hamming codes: instead of choosing every single vector in F^k as a column in the check matrix, we choose a nonzero vector for each "line" through zero in F^k.

A line through zero in F^k is just a set of vectors L such that $av \in L$ whenever $a \in F$ and $v \in L$; in other words, it is a one dimensional subspace of F^k. It is not hard to see that, as in standard Euclidean geometry, two lines which intersect twice are identical, and it is also evident that every line through zero contains q points, where $q = |F|$. As these lines intersect exactly once, they must otherwise form a disjoint cover of the rest of F^k. In other words, the number of lines n satisfies

$$n(q-1) = q^k - 1,$$

or

$$n = \frac{q^k - 1}{q - 1}.$$

This is enough information to describe a check matrix for the q-ary Hamming code, which is an analogue of the binary Hamming code over a field with q elements. The check matrix should have n columns, where n is as above, where each column is chosen from one of the n lines through the origin.

Definition 2.20 (q-ary Hamming code)**.** *The q-ary Hamming code is the linear code over the field $\mathbb{F}_q$ which has a check matrix with $\frac{q^k-1}{q-1}$ columns, where each column is chosen from a different line through zero in $\mathbb{F}_q^n$.*

The check matrix has dimension $k \times n$ and rank k, so the length of q-ary Hamming code is $n = (q^k - 1)/(q - 1)$ and the rank of the code is $n - k = (q^k - 1)/(q - 1) - k$. Note that when $q = 2$, these parameters exactly match the binary Hamming code.

Example 2.21 (A ternary Hamming code). Here we give example check and generator matrices for some ternary Hamming codes. "Ternary" means that $q = 3$, and the alphabet is $\mathbb{F}_3 = \mathbb{Z}/3\mathbb{Z} = \{0, 1, 2\}$. Let's try $k = 2$, so we should have 4 columns of height 2. We can choose two of the columns to be $(1, 0)$ and $(0, 1)$, which are clearly linearly independent.

Another which is independent of each of these is $(1,1)$ and another which is independent of any of those three is $(1,2)$. This gives a check matrix

$$H = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 2 & 1 & 0 & 1 \end{pmatrix}.$$

Since we've written this in standard form, we can easily write a generator matrix

$$G = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \\ 2 & 1 \end{pmatrix},$$

and we can multiply this generator matrix by the 9 different vectors in $\mathbb{F}_3^2$ to get the 9 codewords

$$\begin{gathered} (0,0,0,0) \\ (0,1,1,1) \\ (0,2,2,2) \\ (1,0,1,2) \\ (1,2,0,1) \\ (2,0,2,1) \\ (1,1,2,0) \\ (1,2,0,1) \\ (2,1,0,2). \end{gathered}$$

This is small enough that if we stare at it for a bit we can verify that is it indeed linear and that it has minimum distance 3.

If we keep $q = 3$ and let $k = 3$, we get a check matrix with 13 columns and height 3. For example,

$$H = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 2 & 2 & 1 & 1 & 0 & 1 & 2 & 0 & 1 & 1 & 0 & 1 & 0 \\ 2 & 1 & 2 & 1 & 2 & 2 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

does the trick.

Why should we bother looking at codes over fields other than $\mathbb{F}_2$ when computers work in binary? One answer is "why not?". We look at such codes because we can. Coding theory can be approached from a purely theoretical perspective, and it studies questions which are interesting in their own right.

A different answer is that working over fields other than $\mathbb{F}_2$ can give us more freedom to construct codes, and maybe these codes will be better than binary codes in some ways, even for practical use. Of particular interest here is the base field $\mathbb{F}_{2^k}$, which is a reasonable thing for computers to work with.

We can think of a long binary message as a string of elements of $\mathbb{F}_{2^k}$ by breaking it up into chunks of length k and then use a code where the alphabet is $\mathbb{F}_{2^k}$. Such an encoding will then have the property that k consecutive bit errors will effect at most 2 symbols in the code word, so a code which can correct 2 consecutive (alphabet) errors will be able to

correct any k consecutive bit errors. (We would be able to construct such a code right now by interleaving two Hamming codes over $\mathbb{F}_{2^k}$.) Long strings of consecutive errors are a real life problem, and techniques to deal with them go by the term *burst error correction*.

(Once upon a time, there were small, flat, circular, shiny objects known as *compact discs*, which were used to record things like music and games. A problem with them was that they would get dirty or scratched. By encoding data with interleaved Reed-Solomon codes over a field $\mathbb{F}_{2^k}$, though, they were able to correct many of the large consecutive errors caused by scratches and dirt and achieve a reasonable level of reliability. In fact, errors caused by a scratch as wide as 2 mm would be fully correctable with this coding scheme.)

3. Using algebra

Up to now, we have been considering vector spaces over finite fields as nothing but vector spaces. However, there are some natural sets which form vector spaces over finite fields, and considering our messages and codewords as living in such sets adds extra structure that we can exploit to better construct and decode codes. The two natural vector spaces over a field F that we will consider are polynomials with coefficients in F and extension fields of F.

If F is any field, then the set of polynomials with coefficients in F and degree $< k$ forms a k dimensional vector space. In fact, we can easily map codewords of length k to polynomials of degree $< k$ with the identification

$$(c_0, c_1, \ldots, c_{k-1}) \longrightarrow c_0 + c_1 x + \cdots + c_{k-1} x^{k-1}.$$

Adding two polynomials or multiplying a polynomial by a scalar is exactly the same as adding to codewords or multiplying a codeword by a scalar, so we haven't lost anything. On the other hand, when we think of codewords as polynomials, perhaps we have gained something: we know how to multiply two polynomials to get another polynomial, polynomials have roots, there is a Euclidean algorithm for polynomials, etc.

When the context is clear, we will now consider the above mapping between codewords and polynomials as implied.

Knowing how to multiply polynomials, we can define a function from F^m to F^n by a polynomial $g(x)$ of degree $n - m$: we consider $w \in F^m$ as a polynomial of degree $< m$ and multiplying by g to get a polynomial of degree $< n$. This gives a subset of F^n which is in fact a linear subspace, so it is a linear code. We call a code constructed in this way a *polynomial code*, and we call $g(x)$ a *generator polynomial* for the code.

Definition 3.1. *A* polynomial code C *of length n is a code which consists of all polynomials of degree $< n$ which are divisible by some fixed* generator polynomial $g(x)$*; that is,*

$$C = \{f \in F[x] \text{ such that } \deg(f) < n \text{ and } g(x) | f(x)\},$$

or

$$C = \{q(x)g(x) \text{ such that } h \in F[x] \text{ and } \deg(q) \leq n - m\}.$$

It should be an easy exercise to see that every polynomial code is a linear code, and that if $g(x)$ has degree $n - m$ and messages have length m, then codewords have length n, so the above definition describes an (n, m)-linear code.

Exercise 3.2. Prove that polynomial code is a linear code.

Example 3.3. As an example, we can consider the length 7 polynomial code with generator polynomial $g(x) = 1 + x + x^2 \in \mathbb{F}_2[x]$. We take messages of length 4, so our codewords consist of $g(x)$ multiplied by all of the polynomials in $\mathbb{F}_2[x]$ of degree 3 or less:

$$\begin{aligned}
0(1+x+x^3) &= 0 &&= 0000000\\
1(1+x+x^3) &= 1+x+x^3 &&= 1101000\\
x(1+x+x^3) &= x+x^2+x^4 &&= 0110100\\
(1+x)(1+x+x^3) &= 1+x^2+x^3+x^4 &&= 1011100\\
x^2(1+x+x^3) &= x^2+x^3+x^5 &&= 0011010\\
(1+x^2)(1+x+x^3) &= 1+x+x^2+x^5 &&= 1110010\\
(x+x^2)(1+x+x^3) &= x+x^3+x^4+x^5 &&= 0101110\\
(1+x+x^2)(1+x+x^3) &= 1+x^4+x^5 &&= 1000110\\
x^3(1+x+x^3) &= x^3+x^4+x^6 &&= 0001101\\
(1+x^3)(1+x+x^3) &= 1+x+x^4+x^6 &&= 1100101\\
(x+x^3)(1+x+x^3) &= x+x^2+x^3+x^6 &&= 0111001\\
(1+x+x^3)(1+x+x^3) &= 1+x^2+x^6 &&= 1010001\\
(x^2+x^3)(1+x+x^3) &= x^2+x^4+x^5+x^6 &&= 0010111\\
(1+x^2+x^3)(1+x+x^3) &= 1+x+x^2+x^3+x^4+x^5+x^6 &&= 1111111\\
(x+x^2+x^3)(1+x+x^3) &= x+x^5+x^6 &&= 0100011\\
(1+x+x^2+x^3)(1+x+x^3) &= 1+x^3+x^5+x^6 &&= 1001011.
\end{aligned}$$

We have 16 codewords in a binary code of length 7 and minimum distance 3. If this looks familiar, it's because it is familiar. This happens to be the $(7, 3)$ Hamming code.

There are questions that one might ask upon seeing this example: Is there a way that we could have seen that this is a Hamming code without writing down all of the code words? When we write down other polynomial codes, will they be Hamming codes? The answers to these questions will be "yes" and "sometimes", and the answers rely on understanding the structure of the check matrix of the code, which in turn relies on interpreting the columns of the check matrix as elements of an extension field of F (or collections of such elements).

Recall that a finite field field extension of $\mathbb{F}_q$ is a field $\mathbb{F}_{q^k}$ and can be realized as a quotient

$$\mathbb{F}_{q^k} \cong \mathbb{F}_q[x]/f(x),$$

where $f(x) \in \mathbb{F}_q[x]$ is an irreducible polynomial of degree k. This means that we can identify elements in $\mathbb{F}_{q^k}$ with polynomials in $\mathbb{F}_q[x]$, and vice verse, and thus we can interpret any vector in a vector space over $\mathbb{F}_q$ of dimension k (for example, a code word of length k or a column of height k in a check matrix) as an element of $\mathbb{F}_{q^k}$.

So how should we think about the check matrix of a polynomial code? The answer is to think about the condition that g divides f in terms of the roots of g: the polynomial f is divisible by g if and only if $f(\alpha) = 0$ whenever $g(\alpha) = 0$. Suppose then that α is a root of g. This gives us the condition

$$c_0 + c_1\alpha + c_2\alpha^2 + \cdots + c_{n-1}\alpha^{n-1} = 0,$$

where $f = (c_0, c_1, c_2, \ldots, c_{n-1})$. Now, $\alpha \in \mathbb{F}_{q^k}$ for some k, so we can interpret α and powers of α as vectors of length k. If we write these as columns of a matrix, we find that the condition that $f(\alpha) = 0$ is the same as the condition

$$\begin{pmatrix} | & | & | & & | \\ \alpha^0 & \alpha & \alpha^2 & \cdots & \alpha^{n-1} \\ | & | & | & & | \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{n-1} \end{pmatrix} = \begin{pmatrix} | \\ 0 \\ | \end{pmatrix} = 0.$$

Now, g probably has more than one root, and f needs to have every root of g as a root, so if g has roots $\alpha_0, \ldots \alpha_{k-1}$ we may put all of the conditions into one matrix simultaneously, and get a matrix

$$H = \begin{pmatrix} | & | & | & & | \\ \alpha_0^0 & \alpha_0 & \alpha_0^2 & \cdots & \alpha_0^{n-1} \\ | & | & | & & | \\ | & | & | & & | \\ \alpha_1^0 & \alpha_1 & \alpha_1^2 & \cdots & \alpha_1^{n-1} \\ | & | & | & & | \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ | & | & | & & | \\ \alpha_{k-1}^0 & \alpha_{k-1} & \alpha_{k-1}^2 & \cdots & \alpha_{k-1}^{n-1} \\ | & | & | & & | \end{pmatrix}$$

which will be a check matrix for the polynomial code of length n with generator polynomial g. However, including a set of rows for every single root of g is not necessary. If g is irreducible, α is a root of g, and α is a root of f, then every root of g is in fact a root of f, so we need only include one entry in H for each irreducible factor of g.

Let's pause now to go back and examine the example polynomial code from before, which turned out to be a Hamming code. Our generator polynomial was $g(x) = 1 + x + x^3$, which happens to be irreducible[4], so its roots lie in the field

$$\mathbb{F}_{2^3} = \mathbb{F}_2[x]/g(x).$$

[4]Well, it "happens" to be irreducible because I chose it that way.

We consider a root α of g which we can map to the polynomial x. Under this identification of elements of $\mathbb{F}_{2^3}$ with polynomials in $\mathbb{F}_2[x]$, we find

$$\begin{aligned}
\alpha^0 &\longrightarrow & 1 & &\longrightarrow 100\\
\alpha^1 &\longrightarrow & x & &\longrightarrow 010\\
\alpha^2 &\longrightarrow & x^2 & &\longrightarrow 001\\
\alpha^3 &\longrightarrow & 1+x & &\longrightarrow 110\\
\alpha^4 &\longrightarrow & x+x^2 & &\longrightarrow 011\\
\alpha^5 &\longrightarrow & 1+x+x^2 & &\longrightarrow 111\\
\alpha^6 &\longrightarrow & 1+x^2 & &\longrightarrow 101.
\end{aligned}$$

It turns out[5] that the powers of α give all of the nonzero elements of $\mathbb{F}_{2^3}$; that is, α is a primitive element. A check matrix for the code, then, is

$$H = \begin{pmatrix} | & | & | & & | \\ \alpha^0 & \alpha & \alpha^2 & \cdots & \alpha^6 \\ | & | & | & & | \end{pmatrix},$$

or, more explicitly,

$$H = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{pmatrix},$$

which we can recognize as a check matrix for a $(7, 4)$ Hamming code. A polynomial such as $g(x)$ is known as a primitive polynomial. That is, a *primitive polynomial* $g(x) \in \mathbb{F}_q[x]$ is an irreducible polynomial such that a root of g is a primitive element of the finite field $\mathbb{F}_q[x]/g(x)$. Put another way, a primitive polynomial is the minimal polynomial of a primitive element.

Example 3.4. If $g(x) \in \mathbb{F}_2[x]$ is a primitive polynomial of degree k, then the length $2^k - 1$ polynomial code with generator polynomial $g(x)$ is the $(2^k - 1, 2^k - k - 1)$ Hamming code.

Understanding what the check matrix for a polynomial code looks like, now, we might have a chance of constructing a polynomial code which has minimum distance larger than 3. To do this, we want a fact from linear algebra which we will (in these notes, at least) accept as a matter of faith.

Proposition 3.5. *A matrix V of the form*

$$V = \begin{pmatrix} a_1 & a_2 & \cdots & a_k \\ a_1^2 & a_2^2 & \cdots & a_k^2 \\ a_1^3 & a_2^3 & \cdots & a_k^3 \\ \vdots & \vdots & \cdots & \vdots \\ a_1^k & a_2^k & \cdots & a_k^k \end{pmatrix}$$

[5]Again, it "turns out" this way because I carefully chose g to make this happen.

(called a Vandermonde matrix*) has determinant*

$$\det(V) = \prod_{1\le j\le n} a_j \prod_{1\le i<j\le k} (a_i - a_j).$$

In particular, if all of the a_j are distinct, the determinant is nonzero. (And so the columns of the matrix are linearly independent.)

Having knowledge of this proposition, we might think to construct check matrices which have submatrices which are Vandermonde matrices. The right amount of linear independence will then follow immediately.

Consider, for example, the $2 \times (q^k - 1)$ matrix

$$\begin{pmatrix} 1 & \alpha & \alpha^2 & \cdots \alpha^{q^k-2} \\ 1 & \alpha^2 & \alpha^4 & \cdots \alpha^{2(q^k-2)} \end{pmatrix},$$

where $\alpha \in \mathbb{F}_{q^k}$ is a primitive element. If we take any two columns and put them in a 2×2 matrix, we get something of the form

$$\begin{pmatrix} \alpha^i & \alpha^j \\ \alpha^{2i} & \alpha^{2j} \end{pmatrix},$$

which has nonzero determinant, as $\alpha^i \neq \alpha^j$. This means that any two columns are linearly independent, and therefore the check matrix is the check matrix of a linear code of minimum distance at least 3. Similarly, if we want minimal distance at least 4, we can consider the check matrix

$$\begin{pmatrix} 1 & \alpha & \alpha^2 & \cdots \alpha^{q^k-2} \\ 1 & \alpha^2 & \alpha^4 & \cdots \alpha^{2(q^k-2)} \\ 1 & \alpha^3 & \alpha^6 & \cdots \alpha^{3(q^k-2)} \end{pmatrix}.$$

Now, when we considered the example of the Hamming code as a polynomial code, we labeled the columns of the check matrix with powers of $\alpha \in \mathbb{F}_{2^3}$, but we thought of the matrix as having entries in $\mathbb{F}_2$. So, are we to consider this check matrix with entries in $\mathbb{F}_q$ or $\mathbb{F}_{q^k}$? We can do either, and we either get a code with alphabet $\mathbb{F}_q$ or with alphabet $\mathbb{F}_{q^k}$.

Let's consider first the check matrix with entries in $\mathbb{F}_q$, which we might write as

$$H = \begin{pmatrix} | & | & | & & | \\ \alpha^0 & \alpha & \alpha^2 & \cdots & \alpha^{q^k-2} \\ | & | & | & & | \\ \\ | & | & | & & | \\ \alpha^0 & \alpha^2 & \alpha^4 & \cdots & \alpha^{2(q^k-2)}) \\ | & | & | & & | \\ \\ | & | & | & & | \\ \alpha_0 & \alpha^3 & \alpha^6 & \cdots & \alpha^{3(q^k-2)} \\ | & | & | & & | \end{pmatrix},$$

to emphasize that each symbol represents a (column) vector of height k.

Using the α and the construction of $\mathbb{F}_{2^3}$ from the earlier Hamming code example,

$$\begin{array}{lcl} \alpha^0 \longrightarrow & 1 & \longrightarrow 100 \\ \alpha^1 \longrightarrow & x & \longrightarrow 010 \\ \alpha^2 \longrightarrow & x^2 & \longrightarrow 001 \\ \alpha^3 \longrightarrow & 1+x & \longrightarrow 110 \\ \alpha^4 \longrightarrow & x+x^2 & \longrightarrow 011 \\ \alpha^5 \longrightarrow & 1+x+x^2 & \longrightarrow 111 \\ \alpha^6 \longrightarrow & 1+x^2 & \longrightarrow 101, \end{array}$$

we would have the binary check matrix

$$H = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \end{pmatrix}.$$

(In fact, the removing the middle three rows doesn't change the code at all.) The rank of this check matrix is too large, compared to the number of columns, for this code to be useful. In fact, it just gives the $(7, 1)$ repetition code. To get a less trivial example we would have to consider a primitive polynomial of larger degree. The polynomial $g(x) = 1 + x + x^4$ does the trick, and taking a root $\alpha \in \mathbb{F}_{2^4}$ and doing a similar construction gives a check matrix of width 15 and height 12. In fact, for a binary code constructed in this way, we can remove all of the rows coming from even powers of α, so we can get a check matrix of width 15 and height 8 for a code with minimal distance 5. (That the minimal distance is 5 instead of 4 comes come omitting the 4th row, which we never wrote down.)

Now, we started off with polynomial codes and then went back to constructing and analyzing check matrices. The check matrices that we generated were inspired by what the check matrix for a polynomial code looks like, however, and we can go back and construct a polynomial gives a code with the check matrix that we have constructed. The top row (or block of rows), with entries $\alpha^0, \alpha, \alpha^2, \ldots, \alpha^{q^k-2}$, is equivalent to the condition that $f(\alpha) = 0$, when we consider a codeword as a polynomial. The next row, with entries $\alpha_0, \alpha^2, \alpha^4, \ldots, \alpha^{2(q^k-2)}$, is the condition $f(\alpha^2) = 0$. And so on.

So we want our generator polynomial $g(x)$ to have roots $\alpha, \alpha^2, \ldots, \alpha^k$ for some power of k. We can take g to be the smallest such polynomial, namely

$$g(x) = \mathrm{lcm}(m_\alpha(x), m_{\alpha^2}(x), \ldots, m_{\alpha^k}(x)),$$

where m_α denotes the minimal polynomial of α. By the previous discussion, if our codewords are length $q^k - 1$ and g has degree e, this gives a $(q^k - 1, q^k - e - 1)$ linear code with minimum distance at least $k + 1$.

School of Mathematics, University of Bristol
E-mail address: j.bober@bristol.ac.uk