Matrix Algebra and Error-Correcting Codes

Aaron Fenyes
afenyes@math.utexas.edu

October, 2015

Abstract

These notes started off as an enrichment piece for computer science and electrical engineering majors studying matrix algebra at UT Austin. They're meant to show how the tools you pick up in a first matrix algebra course—things like matrix multiplication, linear equations, column spaces, null spaces, bases, pivots, column operations, and inversion—can be used to design and implement error-correcting codes. A lot of the material is in the exercises, some of which are harder than others, so the notes are probably best read in the company of a more experienced guide.

I learned most of what I know about coding theory from lecture notes by Guruswami [3], Kaplan [4], and others. I'm presenting some of the material in a somewhat original way, however, and I apologize for any errors I've introduced. If you detect or correct any, please let me know.

To stay close to the mindset of elementary matrix algebra, I've occasionally deviated from the conventions of coding theory. Instead of thinking of linear codes as subspaces, for example, I identify them with their generators, and my generators, like Guruswami's, are transposed relative to the usual presentation.

1 Linear algebra with bits

You've already learned a lot about vectors and matrices whose entries are real numbers. In certain areas of computer science, it's very useful to know that pretty much everything you've learned (and, in fact, pretty much everything you're going to learn in M 340L) also applies to vectors and matrices whose entries are *bits*.

1.1 What are bits?

Bits are like real numbers, but different. There are infinitely many real numbers, but there are only two bits: 0 and 1. You can add and multiply bits, using the addition and multiplication tables below.

$$\begin{aligned} 0+0&=0 \qquad & 0\cdot 0&=0\\ 0+1&=1 & 0\cdot 1&=0\\ 1+0&=1 & 1\cdot 0&=0\\ 1+1&=0 & 1\cdot 1&=1 \end{aligned}$$

If you think of the bits 0 and 1 as the logical values FALSE and TRUE, you can think of addition and multiplication as the logical operations XOR and AND.

We've been calling the set of real numbers $\mathbb{R}$, so let's call the set of bits $\mathbb{B}$. Similarly, we've been calling the set of n-entry real vectors $\mathbb{R}^n$, so let's call the set of n-entry bit vectors $\mathbb{B}^n$.

1.2 Algebra with bits

All the algebraic properties of real numbers (in particular, the associative, commutative, and distributive properties) also hold for bits, so you can do algebra with bits just by doing what you'd normally do. There's one thing, however, that you might find a little disconcerting.

The negative of a number a is defined as the solution to the equation $x+a=0$. So, what's the negative of the bit 1? Well, $1+1=0$, so the negative of 1 is 1. In other words, 1 is its own negative. In fact, every bit is its own negative. That means, for bits, addition is the same as subtraction!

For example, let's say we have the equation

$$x+y=1,$$

and we want to write x in terms of y. Normally, we'd do it by subtracting y from both sides of the equation. If we're working with bits, however, subtraction is the same as addition, so we might as well just add y to both sides:

$$\begin{aligned} x+y+y&=1+y\\ x+0&=1+y\\ x&=1+y \end{aligned}$$

Weird.

1.3 Matrix arithmetic with bits

Once you know how to do arithmetic with real matrices, arithmetic with bit matrices is a breeze.

✎ Try working out the matrix-vector product below, just to make sure you've got the hang of it.

$$\begin{aligned}\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} &= 1\begin{bmatrix} 1 \\ 0 \end{bmatrix} + 1\begin{bmatrix} 1 \\ 1 \end{bmatrix} + 0\begin{bmatrix} 0 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} 1+1+0 \\ 0+1+0 \end{bmatrix} \\ &= \begin{bmatrix} 0 \\ 1 \end{bmatrix}\end{aligned}$$

✎ Solve the equation

$$x_1\begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} + x_2\begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} + x_3\begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

by putting the associated augmented matrix in reduced echelon form.

2 Error-correcting codes

When you send a string of bits over a communication channel, there's a chance that some of them might get corrupted; a 1 might become a 0, or vice versa. You can deal with this possibility by adding redundant information to your message, so the intended bit string can be recovered even if a few bits get flipped. A scheme for adding this kind of redundancy is called an *error-correcting code*.

2.1 Repetition codes

One of the simplest ways to protect your message against errors is to just repeat each bit three times:

$$\begin{bmatrix} x \end{bmatrix} \mapsto \begin{bmatrix} x \\ x \\ x \end{bmatrix}$$

If you receive the block

$$\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix},$$

and you're willing to assume that at most one bit got flipped, you can conclude that the block must have been sent as

$$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}.$$

Of course, if two bits got flipped, you're out of luck.

2.2 Parity codes

Here's a slightly more sophisticated code:

$$\begin{bmatrix} x \\ y \end{bmatrix} \mapsto \begin{bmatrix} x \\ x \\ y \\ y \\ x+y \end{bmatrix}$$

This code takes your message two bits at a time and spits out five-bit blocks. That $x+y$ at the end of the error-protected block is called a "parity bit," because it tells you whether the number of 1s in the unprotected block was even or odd.

✎ If you receive the block

$$\begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix},$$

and you're willing to assume that at most one bit got flipped, can you figure out what the block must have been sent as?

3 Linear codes

An error-correcting code is called *linear* if it turns each k-bit block of your message into an n-bit error-protected block by doing the transformation

$$\mathbf{x} \mapsto G\mathbf{x},$$

where G is an $n \times k$ matrix. The matrix G is called the *generator* of the code. Vectors in the range of $\mathbf{x} \mapsto G\mathbf{x}$ are called *codewords*.

Keep in mind that the transformation $\mathbf{x} \mapsto G\mathbf{x}$ has domain $\mathbb{B}^k$ and codomain $\mathbb{B}^n$. In particular, every codeword is a vector in $\mathbb{B}^n$. However, not every vector in $\mathbb{B}^n$ is necessarily a codeword. In fact, we'll soon see that if every vector in $\mathbb{B}^n$ is a codeword, $\mathbf{x} \mapsto G\mathbf{x}$ is totally useless as an error-correcting code.

If you need to get a message out fast, linear codes are your friends, because most CPUs can do bit matrix arithmetic really quickly. We'll see later that with linear codes, you can also use matrix arithmetic to speed up the error-correction process.

✎ The codes described in Section 2 are both linear. Find the generator and codewords of each one.

✎ Write a routine to multiply a 16-entry vector by a 32×16 matrix in your favorite programming language. See how fast you can make it. Bonus points if your favorite language is an assembly language.[1]

3.1 Do you have what it takes?

Let's say you have a linear code with generator G. Can this code actually be used to correct errors? You won't know until you try, so you take a bit vector $\mathbf{x} \in \mathbb{B}^k$, turn it into the codeword $\mathbf{y} = G\mathbf{x}$, and send it over a communication channel. On its way across the channel, $\mathbf{y}$ has its ith entry flipped, becoming $\mathbf{y} + \mathbf{e}_i$.

✎ As usual, $\mathbf{e}_i$ is the ith column of the identity matrix. Why does adding $\mathbf{e}_i$ to $\mathbf{y}$ flip the ith entry of $\mathbf{y}$?

At the other end of the channel, I receive the vector $\mathbf{y} + \mathbf{e}_i$. If this vector is a codeword, it's game over: I'll never even know that an error occurred.

✎ Why?

Now we know the first thing it takes for $\mathbf{x} \mapsto G\mathbf{x}$ to be an error-correcting code:

1. If $\mathbf{y}$ is a codeword, then $\mathbf{y} + \mathbf{e}_i$ is not a codeword, regardless of i.

Let's say your code satisfies this requirement, so $\mathbf{y} + \mathbf{e}_i$ is not a codeword. I immediately realize that an error must have occurred.

✎ Why?

Crossing my fingers, I assume that only one entry of $\mathbf{y}$ got flipped. If I knew it was the ith entry, I could just add $\mathbf{e}_i$ to the vector I received, flipping the corrupted entry back to its original value:

$$(\mathbf{y} + \mathbf{e}_i) + \mathbf{e}_i = \mathbf{y}.$$

Unfortunately, I don't know which entry got flipped, so all I can do is look at the vector $(\mathbf{y} + \mathbf{e}_i) + \mathbf{e}_j$ for each j. If I'm lucky, there will only be one value of j that makes this vector a codeword. In that case, I'll be able to recover $\mathbf{y}$.

✎ Why?

[1] No bonus points if you use the BMM instruction on a Cray X1 supercomputer.

On the other hand, if there's more than one value of j that makes $(\mathbf{y} + \mathbf{e}_i) + \mathbf{e}_j$ a codeword, I won't be able to figure out which value to use. Now we know the second thing it takes for $\mathbf{x} \mapsto G\mathbf{x}$ to be an error-correcting code:

2. If $\mathbf{y}$ is a codeword, and $\mathbf{y} + \mathbf{e}_i + \mathbf{e}_j$ is also a codeword, then $\mathbf{y} + \mathbf{e}_i + \mathbf{e}_j = \mathbf{y}$.

Let's say your code satisfies this requirement, so I can recover $\mathbf{y}$. All I have to do now is work out the vector $\mathbf{x}$ that you started with! In other words, I have to solve the equation $\mathbf{y} = G\mathbf{x}$, where $\mathbf{x}$ is the unknown. If this equation has a unique solution, I'm done. If it doesn't, I'm out of luck. Now we know the last thing it takes for $\mathbf{x} \mapsto G\mathbf{x}$ to be an error-correcting code:

3. For each codeword $\mathbf{y}$, the equation $\mathbf{y} = G\mathbf{x}$ has a unique solution.

By giving it a try, and thinking about what could go wrong at every step along the way, we've found three requirements that tell us whether or not the linear code $\mathbf{x} \mapsto G\mathbf{x}$ can be used to correct errors. Even better, it turns out that each of the three requirements can be simplified a little bit. Here they are, all cleaned up and packed in a fancy box:

Theorem 1. *The linear code with generator G is an error-correcting code if and only if it satisfies the following three requirements.*

1. *The vector $\mathbf{e}_i$ is not a codeword, regardless of i.*
2. *If $\mathbf{e}_i + \mathbf{e}_j$ is a codeword, then $\mathbf{e}_i + \mathbf{e}_j = \mathbf{0}$.*
3. *The transformation $\mathbf{x} \mapsto G\mathbf{x}$ is one-to-one.*

✎ Prove that each of the simplified requirements in Theorem 1 is logically equivalent to the corresponding unsimplified requirement.

HINT: Remember that $\mathbf{0}$ is always a codeword. This makes it really easy to prove that the first two unsimplified requirements imply the corresponding simplified ones.

✎ Use Theorem 1 to decide whether the codes from Section 2 are error-correcting codes.

✎ Prove that if every vector in $\mathbb{B}^n$ is a codeword of $\mathbf{x} \mapsto G\mathbf{x}$, then $\mathbf{x} \mapsto G\mathbf{x}$ is not an error-correcting code.

3.2 Check, please

When we tried out your code in Subsection 3.1, the error-correction procedure I used involved a lot of checking whether things were codewords. First, I had to check the block $\mathbf{y}+\mathbf{e}_i$ you sent me, so I could find out if there was an error. Then, when I saw there was an error, I had to check all the vectors $(\mathbf{y}+\mathbf{e}_i)+\mathbf{e}_j$ as well.

We can streamline this process by using a gadget called a *check matrix*. A check matrix for a linear code is a matrix H with the property that $H\mathbf{z} = \mathbf{0}$ if and only if $\mathbf{z}$ is a codeword.

With a check matrix, error detection and correction is a snap. Let's say you send me the codeword $\mathbf{y}$, and I receive the vector $\mathbf{z}$. The first thing I do is compute the vector $H\mathbf{z}$.[2] If there was no error, $\mathbf{z} = \mathbf{y}$, so

$$\begin{aligned} H\mathbf{z} &= H\mathbf{y} \\ &= \mathbf{0}. \end{aligned}$$

If the ith entry of $\mathbf{y}$ got flipped, $\mathbf{z} = \mathbf{y} + \mathbf{e}_i$, so

$$\begin{aligned} H\mathbf{z} &= H(\mathbf{y} + \mathbf{e}_i) \\ &= H\mathbf{y} + H\mathbf{e}_i \\ &= H\mathbf{e}_i, \end{aligned}$$

which is the ith column of H. To figure out which bit got flipped, all I have to do is look through the columns of H until I find $H\mathbf{z}$.

✎ Wait a minute! What if H has two columns that are the same? Then I'm in trouble. Fortunately, if H is a check matrix for an error-correcting code, all the columns of H have to be different. Prove it.

HINT: Show that if columns i and j of H are the same, then $\mathbf{e}_i + \mathbf{e}_j$ is a codeword.

Here's another use for check matrices—one you probably never expected. If you have a check matrix for a linear code, you can use it to help figure out whether or not the code is error-correcting!

Theorem 2. *Let's say H is a check matrix for a linear code.*

1. *The code satisfies the first requirement of Theorem 1 if and only if all the columns of H are nonzero.*

2. *The code satisfies the second requirement of Theorem 1 if and only if all the columns of H are different.*

[2] People call this vector the *syndrome* of z, possibly just because it sounds cool.

✎ Prove Theorem 2.

✎ The matrix

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

is a check matrix for a one-to-one linear code called the Hamming code. Use Theorem 2 to decide whether or not the Hamming code is error-correcting.

HINT: I told you the code is one-to-one, so you already know it satisfies the third requirement from Theorem 1.

✎ Notice that the ith column of the check matrix above is the number i written in binary. Hamming did that on purpose. Why?

3.3 Cutting checks

Okay, great: check matrices are magic. But how do we get our hands on one? As it turns out, there's a step-by-step procedure you can use to construct a check matrix for any one-to-one linear code. I'll outline the basic strategy first, and then go back to hammer out the details.

We start with a code whose generator G is an $n \times k$ matrix. We *complete* G by finding an invertible $n \times n$ matrix C whose first k columns are the columns of G.

✎ Show that G can be completed if and only if $\mathbf{x} \mapsto G\mathbf{x}$ is one-to-one.

Since the first k columns of C are the columns of G, let's call them $\mathbf{g}_1, \ldots, \mathbf{g}_k$. We'll call the last $n-k$ columns $\mathbf{s}_{k+1}, \ldots, \mathbf{s}_n$. The columns of C are a basis for $\mathbb{B}^n$, so any vector $\mathbf{z} \in \mathbb{B}^n$ can be written as

$$\mathbf{z} = a_1\mathbf{g}_1 + \ldots + a_k\mathbf{g}_k + b_{k+1}\mathbf{s}_{k+1} + \ldots + b_n\mathbf{s}_n,$$

and the weights $a_1, \ldots, a_k, b_{k+1}, \ldots, b_n$ are unique. Multiplication by C^{-1} turns each column of C into the corresponding column of the identity matrix, so

$$C^{-1}\mathbf{z} = a_1\mathbf{e}_1 + \ldots + a_k\mathbf{e}_k + b_{k+1}\mathbf{e}_{k+1} + \ldots + b_n\mathbf{e}_n = \begin{bmatrix} a_1 \\ \vdots \\ a_k \\ b_{k+1} \\ \vdots \\ b_n \end{bmatrix}$$

If $\mathbf{z}$ is a codeword, all the weights $b_\bullet$ are zero. If $\mathbf{z}$ is not a codeword, at least one of the weights $b_\bullet$ is nonzero. Hence, the last $n-k$ entries of $C^{-1}\mathbf{z}$ are all zero if and only if **z** is a codeword. That means we can use the last $n-k$ rows of C^{-1} as a check matrix for our code.

Now that we know what we're doing, let's work out the details. First, how do we complete G?

✎ Show that if $\mathbf{x} \mapsto G\mathbf{x}$ is one-to-one, throwing away the non-pivot columns of the block matrix

$$\left[\begin{array}{c|c} G & I_n \end{array}\right]$$

leaves you with an invertible $n \times n$ matrix whose first k columns are the columns of G.

Next, let's think about the most efficient way to find the inverse of C. We can save time by only computing the last $n-k$ rows of C^{-1}, since those are the only rows we actually need.

✎ Show that if you use column operations to turn the upper block of

$$\left[\begin{array}{c} C \\ \hline I_n \end{array}\right]$$

into the identity matrix, the lower block will become C^{-1}. If you only start with a few rows of I_n in the lower block, you'll end up with the corresponding rows of C^{-1}.

And that's all there is to it. Now you've got a way to find a check matrix for any one-to-one linear code.

✎ Decide whether the codes described in Section 2 are one-to-one. If they are, find check matrices for them, and use Theorem 2 to decide whether they're error-correcting.

✎ The matrix

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

is the generator of the Hamming code—the one you heard about at the end of Subsection 3.2. Find a check matrix for the Hamming code. Is yours the same as the one Hamming found?

3.4 Cashing checks

Check matrices are so useful that code designers sometimes work backwards, starting with a check matrix and building a code around it. Given a matrix H, let's try to find a one-to-one linear code that H is a check matrix for.

Turning around the definition of a check matrix in Subsection 3.2, we see that we're looking for a code with the property that $\mathbf{z}$ is a codeword if and only if $H\mathbf{z} = \mathbf{0}$. In other words, we're looking for a code whose set of codewords is the null space of H.

If the null space of H contains only the zero vector, we're looking for a code that has only one codeword, which would be totally useless for communication. So we're stuck.

On the other hand, if the null space of H contains more than just the zero vector, we can use a standard linear algebra trick to find a matrix G whose columns are a basis for the null space of H.

✎ Show that $\mathbf{x} \mapsto G\mathbf{x}$ is a one-to-one code whose set of codewords is the null space of H—which is exactly what we wanted.

✎ Show that doing row operations on a check matrix gives you another check matrix for the same code.

To see some examples of linear codes that are defined backwards like this, check out Subsections 5.2 and 5.4.

4 Detecting and correcting multiple errors

Until now, we've only been thinking about whether our codes can be used to detect and correct a single error in each protected block. As our blocks get longer and longer, however (and they will, in Section 5), multiple errors will get more likely. How do we figure out how many errors a code can handle?

In Subsection 3.1, we took a linear code and tested it out by encoding a message, flipping a single bit, and then trying to recover the original message without knowing which bit got flipped. We discovered that we could be sure of detecting the error if and only if no standard basis vector was a codeword, and we could be sure of correcting it if and only if summing two standard basis vectors would never give a codeword other than zero. Using the same kind of testing procedure, it's not too hard to find conditions for detecting and correcting multiple errors with a linear code:

1. We can detect up to q errors if and only if summing q or fewer standard basis vectors will never give a codeword other than zero.
2. We can correct up to q errors if and only if summing $2q$ or fewer standard basis vectors will never give a codeword other than zero.

The minimum number of standard basis vectors you have to sum to get a codeword other than zero is called the *minimum distance* of a linear code.

✎ Prove the conditions I just gave for detecting and correcting multiple errors.

✎ Show that if H is a check matrix for a linear code, the minimum number of columns of H you have to sum to get zero is the same as the minimum distance of the code.

✎ If you turn H into a new matrix $\tilde{H}$ by rearranging its columns, do codes with check matrices H and $\tilde{H}$ always have the same minimum distance?

✎ What if you turn H into $\tilde{H}$ by doing column operations?

✎ Find the minimum distance of each of the three codes we've seen so far. (Two are in Section 2, and one is mentioned in Subsections 3.2 and 3.3.)

5 Families of codes

The matrix

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

is the generator of an error-correcting code called the $(2, 4)$ Reed-Muller code. What makes this code work? How did someone come up with it? It's probably hard to imagine. A typical error-correcting code doesn't make much sense if you see it in isolation.

Things often become clearer if you look at a family of codes that are all constructed in basically the same way. In this section, I'll introduce you to some well-known families of error-correcting codes.

5.1 Repetition codes

The first error-correcting code we talked about was the one where you repeat each bit three times:

$$\begin{bmatrix} x \end{bmatrix} \mapsto \begin{bmatrix} x \\ x \\ x \end{bmatrix}$$

But why stop at three?

✎ Find the minimum distance of the code

$$\begin{bmatrix} x \end{bmatrix} \mapsto \left.\begin{bmatrix} x \\ \vdots \\ x \end{bmatrix}\right\} n \text{ rows}$$

Every extra repetition gives us more protection against errors—but every extra bit we send gives another chance for an error to happen. We can investigate the tradeoff by encoding a bit and sending the resulting n-bit block across a channel that has probability ε of flipping each bit. If the bit flips are *independent*—that is, if knowing whether one bit got flipped doesn't help us guess whether another bit got flipped—the probability of accumulating more errors than the code can correct turns out to be at most

$$\left[\frac{\varepsilon(1-\varepsilon)}{4}\right]^{n/2}.$$

That means our chances of guessing the original bit wrong shrink exponentially as the number of repetitions grows, as long as $\varepsilon \leq 1/2$. If you know a little probability theory, or you're willing to learn some, you can see where this estimate comes from in Appendix A.1.

5.2 Hamming codes

In Subsections 3.2 and 3.3, you met the Hamming code, a one-to-one code with a check matrix whose columns are the numbers 1 through 7 written in binary. Let's say an r-digit Hamming code is any one-to-one code with a check matrix whose columns are the numbers 1 through $2^r - 1$ written in binary.

✎ Show that every r-digit Hamming code is error-correcting.

✎ List all the 2-digit Hamming codes. Do they look familiar?

5.3 Reed-Muller codes

Block matrices give us an elegant way of building new codes from old ones.

✎ Let's say A and B are matrices with the same number of rows. Show that if the codes with generators A and B are both error-correcting, the code whose generator is the block matrix

$$G = \left[\begin{array}{c|c} A & B \\ \hline 0 & B \end{array}\right]$$

is also error-correcting.

✎ Suppose H_A is a check matrix for the code with generator A, and H_B is a check matrix for the code with generator B. Show that

$$\left[\begin{array}{c|c} H_A & H_A \\ \hline 0 & H_B \end{array}\right]$$

is a check matrix for the code with generator G.

✎ Suppose the code with generator A has minimum distance d_A, and the code with generator B has minimum distance d_B. Show that the minimum distance of the code with generator G is the minimum of d_A and $2d_B$.

HINT: Apply the second exercise of Section 4 to the check matrix above.

That last exercise may sound rather dry, but it shows that the block matrix construction we've been thinking about has a remarkable property, best illustrated through an example.

The code with generator

$$A = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

is pretty boring. It's a repetition code that can detect one error in each two-bit output block, at the cost of doubling the length of your message. It's hardly worth pointing out that $d_A = 2$.

The code with generator

$$B = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

is totally useless: it "encodes" a block of two bits by not doing anything to it. It obviously can't correct, or even detect, errors. If you're a hopeless pedant, you can prove this by observing that $d_B = 1$.

Now that we know d_A and d_B, the last exercise tells us that the code with generator

$$\left[\begin{array}{c|cc} 1 & 1 & 0 \\ 1 & 0 & 1 \\ \hline 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}\right]$$

has a minimum distance of 2. This code is neither boring nor useless! It only increases the length of your message by a factor of 4/3, but it can still detect up to one error in each four-bit output block. By themselves, the codes with generators A and B are unimpressive: one uses a lot of extra space just to detect one error, and the other is very space-efficient but doesn't do anything. Through the block matrix construction, the two codes can pool their strengths and cover each other's weaknesses, becoming a more sophisticated code.

This idea can be taken much further, leading to a family of codes called Reed-Muller codes. Each member of the family is labeled by a pair of numbers. The $(0, m)$ Reed-Muller code is the 2^m repetition code, and the (m, m) Reed-Muller code is the 2^m identity code. Their generators are

$$F(0, m) = \left.\begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}\right\} 2^m \qquad F(m, m) = \left.\begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix}\right\} 2^m$$

The other Reed-Muller codes are built from these ones using the block matrix construction we've been studying. For positive numbers $r < m$, the (r, m) Reed-Muller code is given by the generator

$$F(r, m) = \left[\begin{array}{c|c} F(r-1, m-1) & F(r, m-1) \\ \hline 0 & F(r, m-1) \end{array}\right].$$

If you think of the $(\bullet, m)$ codes as the "mth generation," you can say that each code is a combination of two codes from the previous generation—except for the repetition and identity codes, of course, which were there from the start. The first few generations of Reed-Muller codes are shown in Figure 1, on the next page. You might find it helpful to refer to the figure while doing the exercises below.

- Write down formulas for the output block size and the minimum distance of the (r, m) Reed-Muller code.
- On the (r, m) grid, connect the dots that label Reed-Muller codes with the same minimum distance. Using a different color or line style, connect the dots that label codes with the same ratio of minimum distance to output block size.
- Calculate the input block sizes of the $(0, 1)$, $(1, 3)$, $(2, 5)$, and $(3, 7)$ codes.
- If you're into combinatorics, try to write down a formula for the input block size of the (r, m) code.

  HINT: Binomial coefficients obey the identity $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$.

Looking through the Reed-Muller family, we can find sequences of codes with steadily growing input block size and minimum distance, but more or less constant space efficiency. This can be very useful for certain applications, as illustrated by the case studies in Appendix A.2.

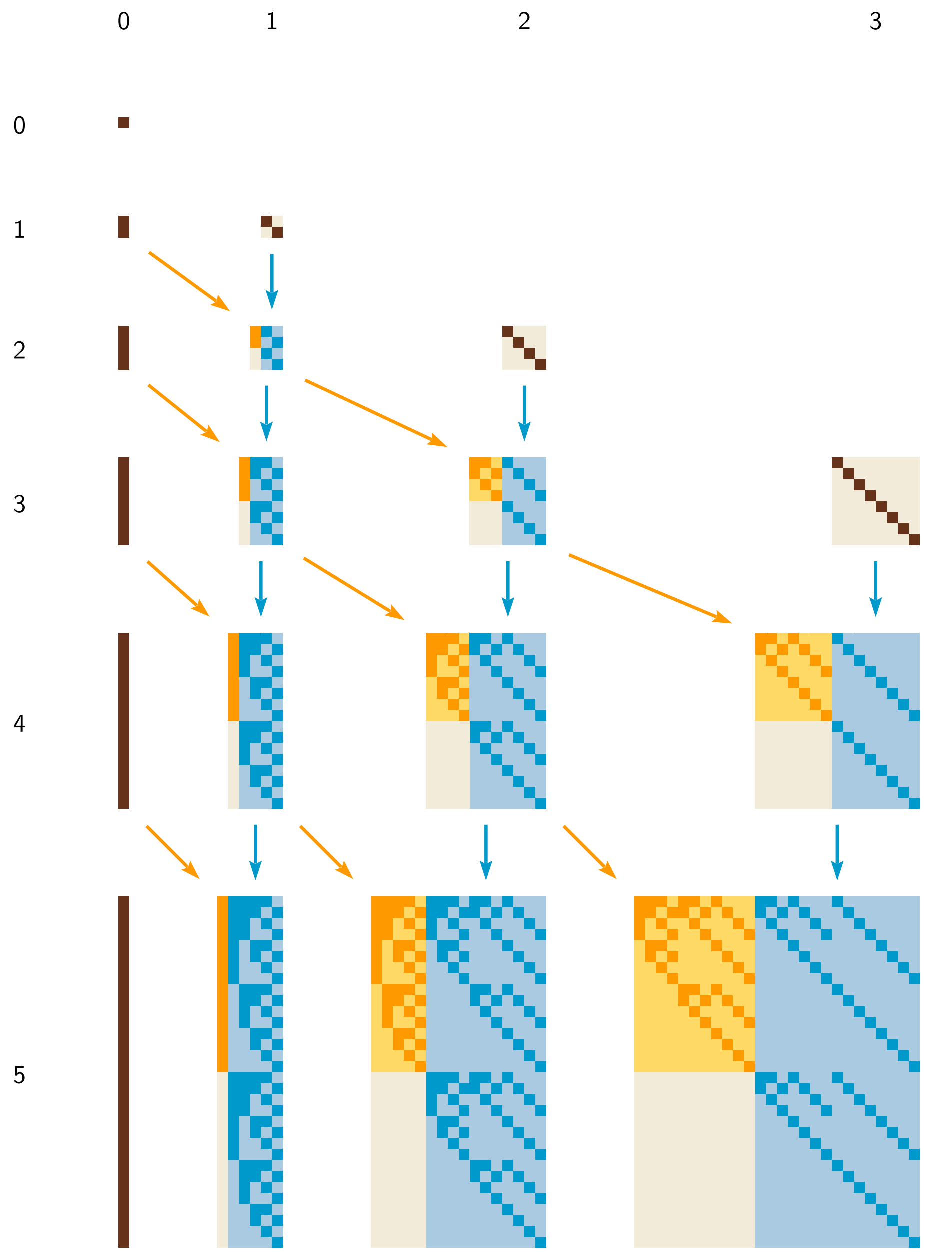

Figure 1: The generator matrices for the first few generations of Reed-Muller codes, with ones and zeros drawn as dark and light pixels. The colored blocks and arrows highlight the way each code is built from two codes in the previous generation.

5.4 Low-density parity check codes

Matrix multiplication is a lot easier when one of the matrices is mostly zeros—or *sparse*, if you like technical terms. In Section 3.2, the first step of our streamlined error-correction process was to multiply the received block by a check matrix, so you might wonder if having a sparse check matrix would let us speed up the process even more. It does, and codes with sparse check matrices are remarkable for more than just that. These codes, called low-density parity check codes, can approach the theoretical limit of error correction efficiency, but the principles of their design remain baffling, even after more than two decades of intense research.

There are lots of different kinds of low-density parity check codes. We'll focus on the ones called Gallager codes, which were among the first to be discovered, although their significance wasn't appreciated until much later. For $3 \leq j < k$, an (n, j, k) Gallager code is any one-to-one code with a check matrix that has n columns, j ones in each column, and k ones in each row.

✎ How many rows does an (n, j, k) check matrix have?

There are lots of (n, j, k) Gallager codes. Some are pretty good, some are pretty bad, and I don't know of any easy way to say for sure which is which. When Gallager studied these codes, however, he discovered a remarkable fact: when n is really big, picking (n, j, k) check matrices at random gives you codes with consistently large minimum distances. When n is large enough, for example, a random $(n, 3, 4)$ check matrix is almost sure to give you a code with a minimum distance of at least $0.122\,n$, and a random $(n, 5, 6)$ check matrix is almost sure to give a minimum distance of at least $0.255\,n$ [1, Figure 4]. Gallager's proof, which is not for the faint of heart, gives a practical way to calculate these floors. See Section 2.2 of [2] for details, if you dare; the key result is Theorem 2.4.

✎ Come up with a practical way to pick (n, j, k) check matrices at random so that each one has the same chance of being chosen.

HINT: There are lots of practical ways to put a bunch of things in a random order so that each order is equally likely. One good one is called the Knuth shuffle. Another involves a bag of Scrabble tiles.

✎ In the exercises from Sections 3.4 and 4, you learned that rearranging the rows and columns of a check matrix doesn't affect the minimum distance of the corresponding codes. Using this fact, show that if you want to take advantage of the minimum distance floor for (n, j, k) Gallager codes, you don't have to pick the whole check matrix at random: you can fix the first n/k rows and randomize the rest.

✎ Write a computer program that will spit out a random (n, j, k) check matrix and a generator to go with it. Write another program that can use the generator and check matrix to encode and decode messages.

6 A flight of fancy

The tools and ideas we've been using to learn about error-correcting codes come from matrix algebra. These methods, at heart, are all about pushing bits around. They tend to feel very concrete, and they translate quite naturally into computer programs. They make it easy, however, to get bogged down in implementation details, obscuring the big picture. Reading about a complicated construction in the language of matrix algebra, I often feel like I'm trying to learn how a computer chip works by cracking it open and looking at the wires.

If you take matrix algebra and boil off the details, leaving only the concepts behind, you end up with a subject called abstract linear algebra. All the tools of matrix algebra have parallels there, often appearing in a more visual and intuitive way. Vectors become points in space, matrices become geometric transformations, and bases become evenly spaced grids. The minimum distance of a code appears as a literal distance, the shortest walk from the zero vector to a codeword.

As a teaser, let's revisit the task of finding a check matrix for a given linear code. There's a matrix algebra procedure for doing this, outlined in Section 3.3. The procedure itself, described at the end of the section, is a mysterious dance of pivot-finding and column operations. The preceding discussion of how the procedure works is long and unwieldy, and it probably doesn't leave you feeling like you could have found the right moves on your own.

If you learn enough abstract linear algebra, you can boil all of Section 3.3 down to the sentence below. It gives both a conceptual explanation of how the procedure works and a fairly concrete recipe for carrying it out. If you're used to cokernels, whatever those are, it's probably also one of the the first things you'd try if you set out to find a check matrix.

> To get a check matrix for a linear code whose generator G is a matrix with n rows, just take the canonical projection of $\mathbb{B}^n$ onto the cokernel of $\mathbf{x} \mapsto G\mathbf{x}$ and write it as a matrix with respect to the standard basis for $\mathbb{B}^n$.

You don't have to learn a bunch of fancy math to solve practical engineering problems, but you can find a lot of shortcuts up in the clouds. And besides—one of the nice things about learning to walk is that someday, you can learn to fly.

A Reliability

Evaluating the reliability of error-detecting and error-correcting codes can be surprisingly tricky, especially if you want to understand how it changes as you move through a family of increasingly sophisticated codes. In Appendix A.1, we'll see how this kind of analysis plays out for repetition codes, which are simple enough to be studied as a family without bringing in too much heavy machinery.

Fortunately, when it comes to specific applications, it's usually enough to compare codes on a case-by-case basis. We'll work through an example in Appendix A.2.

A.1 A family analysis of repetition codes

The n repetition code, discussed in Section 5.1, is one of the simplest error-correcting codes. Let's find out how reliable it is, with an eye toward seeing how its reliability changes as n grows. I'll encode a single bit and send you the resulting n-bit block. If fewer than half of the bits get flipped on their way across the communication channel, you'll correctly reconstruct the bit I started with. If half or more of the bits get flipped, however, your reconstruction will be wrong. We'll discover that, under suitable conditions, your chances of incorrectly reconstructing my bit shrink exponentially as n grows.

Let's say our channel flips each bit with probability ε, and the bit flips are *independent*: knowing whether one bit got flipped won't help us guess whether another bit got flipped. Call the total number of bit flips R. We're uncertain about the value of R, although we can calculate the probabilities of its possible values, so R is an example of a *random variable*. We want to estimate the probability that $R \geq n/2$, which I'll abbreviate as $\Pr(R \geq n/2)$.

We can simplify our calculation using a sneaky trick. Pick a positive number λ, and consider the random variable λ^R. Although it looks more complicated than R, this new random variable will prove easier to work with, and it contains the same information.

✎ Explain why the probability that $R \geq n/2$ is equal to the probability that $\lambda^R \geq \lambda^{n/2}$.

Let's count errors bit by bit, defining R_i to be one if the ith bit got flipped, and zero otherwise. The numbers $R_1, \ldots, R_n$ are random variables too, and

$$R = R_1 + \ldots + R_n.$$

It follows that

$$\begin{aligned}\lambda^R &= \lambda^{R_1 + \ldots + R_n} \\ &= \lambda^{R_1} \cdots \lambda^{R_n}.\end{aligned}$$

The *expectation value* $\mathrm{E}(U)$ of a random variable U is, roughly speaking, the value you should guess it has if you want to minimize how wrong you are. If the possible values of U are u_1 through u_k, the expectation value of U is given by the formula

$$\mathrm{E}(U) = u_1 \Pr(U = u_1) + \ldots + u_k \Pr(U = u_k).$$

The random variable λ^{R_i} has the value λ with probability ε, and the value 1 with probability $1 - \varepsilon$, so

$$\mathrm{E}(\lambda^{R_i}) = \lambda\varepsilon + 1(1 - \varepsilon).$$

Let's pick $\lambda = \frac{1-\varepsilon}{\varepsilon}$, so this expression simplifies to

$$\mathrm{E}(\lambda^{R_i}) = 2(1-\varepsilon).$$

Because we're assuming the bit flips are independent, the values of the random variables $\lambda^{R_1}, \ldots, \lambda^{R_n}$ are independent too: knowing the value of one variable doesn't help us guess the values of the others. When phrased in a more mathematically precise way, this turns out to mean

$$\mathrm{E}(\lambda^{R_1} \cdots \lambda^{R_1}) = \mathrm{E}(\lambda^{R_1}) \cdots \mathrm{E}(\lambda^{R_n}).$$

✎ Using our previous calculations, deduce from the statement above that

$$\mathrm{E}(\lambda^{R}) = 2^n(1-\varepsilon)^n.$$

If U is a random variable that never takes negative values, *Markov's inequality* says that for any constant c,

$$\Pr(U \geq c) \leq \frac{1}{c}\,\mathrm{E}(U).$$

✎ Use Markov's inequality, together with our other calculations, to show that

$$\Pr(\lambda^R \geq \lambda^{n/2}) \leq \left[\frac{\varepsilon(1-\varepsilon)}{4}\right]^{n/2}.$$

The probability that $\lambda^R \geq \lambda^{n/2}$, as you argued earlier, is the same as the probability that $R \geq n/2$, so this bound is just the one we wanted:

$$\Pr(R \geq n/2) \leq \left[\frac{\varepsilon(1-\varepsilon)}{4}\right]^{n/2}.$$

This result is a special case of *Hoeffding's inequality*, and the reasoning we used to prove it is an example of a *Chernoff bound*.

✎ See if you can use the methods from this section to investigate the reliability of a more complicated family of codes.

A.2 A case study of Reed-Muller codes

Let's say someone's going to send us a 16-bit message, and we want to be really, really sure we received it correctly. It's an all-or-nothing game: one undetected error in the received message is as bad as eleven. We can afford to double the length of the message to add redundancy, and hopefully catch any errors. The $(0,1)$, $(1,3)$, and $(2,5)$ Reed-Muller codes from Section 5.3 sound like good tools for the job, but which one is the best?

Here are the details of the three codes under consideration.

Code	$(0,1)$	$(1,3)$	$(2,5)$
Input block size	1	4	16
Output block size	2	8	32
Minimum distance	2	4	8

It will take sixteen blocks of the $(0,1)$ code, four blocks of the $(1,3)$ code, or one block of the $(2,5)$ code to carry the message.

If you're lazy, you could try using the ratio of a code's minimum distance to its output block size as a rough measure of reliability. This ratio, you might figure, tells you how many errors per bit the transmission can sustain without overwhelming the code's error-detection capabilities, so bigger is better.

✎ Based on this reasoning, which of our candidates should be the most reliable code?

There's a subtle problem with this analysis, though. Suppose the message is encoded using sixteen blocks of the $(0,1)$ code. The transmission can accumulate as many as sixteen errors without overwhelming the code, but that's only in the best case, when the errors all fall in different blocks. In the worst case, the code is much more fragile: all it takes is two errors in the same block to corrupt the transmission beyond detection. The $(2,5)$ code, by comparison, can detect up to seven errors anywhere in the transmission.

It looks like the reliability contest will come down to a balance between best-case and worst-case performance. To find out which way it goes, we'll need to do a more detailed calculation. Let's say a block has been transmitted *safely* if the number of transmission errors is less than the code's minimum distance, making it impossible for the errors to slip through undetected. Our goal is to avoid missing even a single transmission error, so we'll call the whole transmission safe if every block is sent safely. For each candidate code (r,m), we want to know the probability $P_{\text{all}}(r,m)$ of a safe transmission.

As usual, let's say our channel flips each bit with probability ε, and errors in different bits are independent. Each probability P_{all} will turn out to be a polynomial function of ε. For a typical communication channel, ε is somewhat small, making P_{all} close to one. In this situation, it's convenient to look instead at the probability $Q_{\text{all}} = 1 - P_{\text{all}}$ of an unsafe transmission. Since ε is small, we can hope to get a reasonable approximation of Q_{all} by computing it "to leading order" in ε, ignoring all the terms except the one with the lowest power of ε. This technique paints a misleading picture in some cases,[3] but those cases tend to feel somewhat con-

[3]Suppose, for example that Q_{all} is the polynomial

$$1 - \frac{199}{100}(1-\varepsilon)^{50} + \frac{99}{100}(1-\varepsilon)^{100}.$$

There's nothing inherently wrong with imagining this; as ε varies from zero to one, Q_{all} does the same, as it should. Observing that $Q_{\text{all}} = \varepsilon/2 +$ [higher powers], we'd typically expect Q_{all} to be near $\varepsilon/2$ when ε is small. When $\varepsilon = 0.01$, for instance, we'd expect Q_{all} to be around 0.005—but in fact, it's over 0.15!

trived, so we can take our leading-order approximations of Q_{all} as good preliminary indications of how reliable our candidate codes are.

The probability $Q_{\text{all}}(0,1)$ is the most straightforward to approximate, because the probability $P_{\text{block}}(0,1)$ of safely sending a single block with the $(0,1)$ code is very simple. Let $Q_{\text{block}} = 1 - P_{\text{block}}$.

✎ Explain why $Q_{\text{block}}(0,1) = \varepsilon^2$.

✎ Explain why $P_{\text{all}}(0,1) = P_{\text{block}}(0,1)^{16}$.

✎ Deduce that $Q_{\text{all}}(0,1) = 16\,\varepsilon^2 + \text{[higher powers]}$.

The probability $Q_{\text{all}}(2,5)$ is the next most straightforward, because the $(2,5)$ code can fit the whole message in a single block. That means $Q_{\text{all}}(2,5) = Q_{\text{block}}(2,5)$.

✎ Explain why

$$Q_{\text{block}}(2,5) = \binom{32}{8}\varepsilon^8(1-\varepsilon)^{24} + \ldots + \binom{32}{32}\varepsilon^{32}(1-\varepsilon)^0,$$

and fill in a bit of the ... while you're at it.

✎ Deduce that $Q_{\text{all}}(2,5) \approx 10^7\,\varepsilon^8 + \text{[higher powers]}$.

To approximate $Q_{\text{all}}(1,3)$, we combine the techniques we used for the other two probabilities.

✎ Explain why

$$Q_{\text{block}}(1,3) = \binom{8}{4}\varepsilon^4(1-\varepsilon)^4 + \ldots + \binom{8}{8}\varepsilon^8(1-\varepsilon)^0.$$

✎ Explain why $P_{\text{all}}(1,3) = P_{\text{block}}(1,3)^4$.

✎ Deduce that $Q_{\text{all}}(1,3) \approx 300\,\varepsilon^4 + \text{[higher powers]}$.

Now that we've approximated the relevant probabilities, we can compare our candidate codes.

✎ Based on our leading-order approximations, decide which candidate has the best chance of safely transmitting the message when ε is small—around 0.01, say.

✎ To get some indication of how valid our leading-order approximation is, compute the probabilities Q_{all} out to the next-lowest powers of ε, and check whether the extra terms are really negligible compared to the lowest order terms.

✎ Use a computer algebra system like `SymPy` to compute the probabilities P_{all} numerically for various values of ε. Now you can say for sure which candidate is the best for the job when ε is around 0.01. Did our leading-order approximations lead us to the right conclusion?

References

[1] R. G. Gallager. Low-density parity-check codes. *IRE Transactions on Information Theory*, 8(1):21 – 28, 1962.

[2] Robert G. Gallager. *Low-Density Parity-Check Codes*. M.I.T. Press, 1963.

[3] Venkatesan Guruswami. Introduction to coding theory. Carnegie Mellon University, Spring 2010. `http://www.cs.cmu.edu/~venkatg/teaching/codingtheory/`.

[4] Nathan Kaplan and members of the tutorial. Coding theory lecture notes. Harvard, Summer 2011. `http://users.math.yale.edu/~nk354/teaching.html`.