CCS Discrete II — Professor: Padraic Bartlett

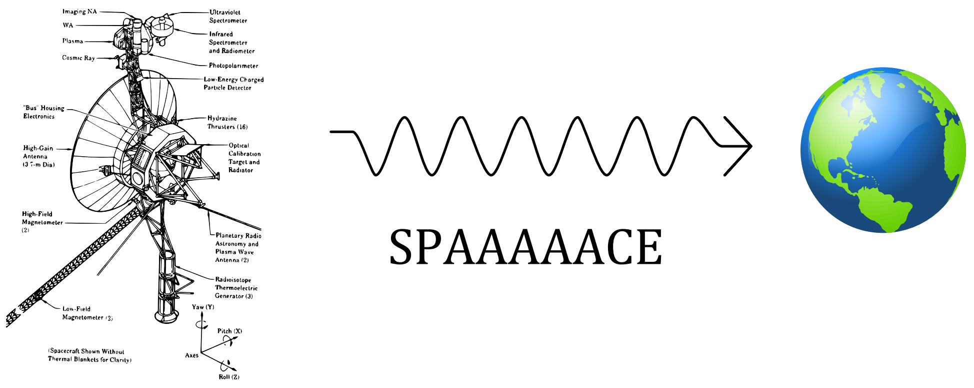
Lecture 1: Error-Correcting Codes; Vector Spaces

Week 1 — *UCSB 2015*

1 Error-Correcting Codes

To open our class, we're going to study the following problem:

Problem. Suppose that you are the Voyager 1 probe. You are currently on the outer limits of the solar system, and about to leave the solar system forever! Consequently, you want to call your parents. However, you are currently separated from your parents by the vast interstellar void of SPAAAAAAAAAACE.


Imaging NA
WA
Plasma
Cosmic Ray
Ultraviolet Spectrometer
Infrared Spectrometer and Radiometer
Photopolarimeter
Low-Energy Charged Particle Detector
"Bus" Housing Electronics
Hydrazine Thrusters (16)
High-Gain Antenna (3.7-m Dia)
Optical Calibration Target and Radiator
High-Field Magnetometer (2)
Planetary Radio Astronomy and Plasma Wave Antenna (2)
Radioisotope Thermoelectric Generator (3)
Low-Field Magnetometer (2)
Yaw (Y)
Pitch (X)
Axes
Roll (Z)
(Spacecraft Shown Without Thermal Blankets for Clarity)
SPAAAAACE

The vast interstellar void of space has an annoying habit of occasionally containing stray electromagnetic waves that interfere with your communications back home; when you send a signal back home (in binary, naturally), occasionally one of your 1's will be switched into a 0, or vice-versa. Assume that no more than one out of three consecutive bits in any message you send will be scrambled.

How can you call home?

One solution to this problem you might come up with is to simply "build redundancy" into the signals you send home, by sending (say) six 1's every time you went to send a 1, and six 0's every time you went to send a 0. For example, to send the message "0101," you'd send

$$000000111111000000111111.$$

Then, even if some of the bits are flipped, your parents back on earth could still decode your message. In particular, if at most one bit out of any consecutive three is flipped, each of your all-0's blocks will have at most two 1's in them after errors are introduced, while each of your all-1's blocks will have at most two 0's in them after errors. In either case, we would never confuse one of these blocks with the other: if your parents received the signal

$$010010001111100001101011,$$

they'd break it up into the four blocks

$$010010001111100001101011,$$

and "correct" the errors to

$$000000111111000000111111,$$

which is unambiguously the signal 0101.

This code can correct for the presence of one error out of any three consecutive blocks, but no better (i.e. if we could have more than two errors in a block of six, we might have three errors in a string of six: in this case it would be impossibe to tell what our string was intended to be. For example, the string 000111 could have resulted from three errors on the signal 000000, or three errors on the signal 111111.) It can accomplish this at the cost of sending $6k$ bits whenever it wants to transmit k bits of information.

Can we do better? In specific, can we make a code that is more efficient (i.e. needs less bits to transmit the same information,) or can correct for more errors? With a little thought, it's easy to improve our code above: if we instead simply replace each 0 with just 000 and each 1 with 111, our code can still correct for the presence of at most one error in any three consecutive blocks (for example, 101 is unambiguously the result of one error to 111,) and now needs to send just $3k$ bits to transmit k bits of information.

There are more interesting codes than just these repetition codes: consider for example the codeword table

word	signal to transmit
000	000000
100	100011
010	010101
001	001110
011	011011
101	101101
110	110110
111	111000

In this code, we encode messages by breaking them into groups of three, and then replacing each string of three with the corresponding group of six. For example, the message "010 101 111" would become

$$010101101101111000.$$

In this code, every word in the table above differs from any other word in at least three spots (check this!) Therefore, if we have at most 1 error in any six consecutive bits, we would never confuse a word here with any other word: changing at most one bit in any block of six would still make it completely unambiguous what word we started with.

Therefore, if we sent the string that we described above, and people on Earth received

$$010111101111110000,$$

they would first break it into groups of six

$$010111101111110000,$$

and then look through our codeword table for what words these strings of six could possibly be, if at most one error in every six consecutive bits could occur:

$$010101101101111000.$$

This then decodes to "010 101 111," the message we sent.

This code can correct for at most one error in any six consecutive bits (worse than our earlier code,) but does so much more efficiently: it only needs to send $2k$ bits to transmit a signal with k bits of information in it.

So: suppose we know ahead of time the maximum number of errors in any consecutive string of symbols. What is the most efficient code we can make to transmit our signals?

At this time, it makes sense to try to formalize these notions of "maximum number of errors" and "efficiency." Here are a series of definitions, that formalize the words and ideas we've been playing with in this talk:

Definition. A q-ary block code C of length n is a collection C of words of length n, written in base q. In other words, C is just a subset of $(\mathbb{Z}/q\mathbb{Z})^n$.

Example. The "repeat three times" code we described earlier is a 2-ary code of length 3, consisting of the two elements $\{(000), (111)\}$. We used it to encode a language with two symbols, specifically 0 and 1.

The second code we made is a 2-ary code of length 6, consisting of the 8 elements we wrote down in our table.

Definition. Given a q-ary code C of length n, we define its **information rate** as the quantity

$$\frac{\log_q(\text{\# of elements in C})}{n}$$

This, roughly speaking, captures the idea of how "efficient" a code is.

Example. The "repeat three times" code we described earlier contains two codewords of length 3; therefore, its information rate is

$$\frac{\log_2(2)}{3} = \frac{1}{3}.$$

This captures the idea that this code needed to transmit three bits to send any one bit of information.

Similarly, the second code we made contains 8 codewords of length six, and therefore has information rate

$$\frac{\log_2(8)}{6} = \frac{3}{6} = \frac{1}{2}.$$

Again, this captures the idea that this code needed to transmit two bits in order to send any one bit of information.

Definition. The **Hamming distance** $d_H(\mathbf{x}, \mathbf{y})$ between any two elements $\mathbf{x}, \mathbf{y}$ of $(\mathbb{Z}/q\mathbb{Z})^n$ is simply the number of places where these two elements disagree.

Given a code C, we say that the minimum distance of C, $d(C)$, is the smallest possible value of $d_H(\mathbf{x}, \mathbf{y})$ taken over all distinct $\mathbf{x}, \mathbf{y}$ within the code. If $d(C) \geq k$, we will call such a code a distance-k code.

Example. The Hamming distance between the two words

$$12213, 13211$$

is 2, because they disagree in precisely two places. Similarly, the Hamming distance between the two words

$$TOMATO, POTATO$$

is 2, because these two words again disagree in precisely two places.

The "repeat three times" code from earlier has minimum distance 3, because the Hamming distance between 000 and 111 is 3.

Similarly, the second code we described from earlier has minimum distance 3, because every two words in our list disagreed in at least 3 places.

The following theorem explains why we care about this concept of distance:

Theorem. A code C can detect up to s errors in any received codeword as long as $d(C) \geq s+1$. Similarly, a code C can correct up to t errors in any received codeword to the correct codeword as long as $d(C) \geq 2t+1$.

Proof. If $d(C) \geq s+1$, then making s changes to any codeword cannot change it into any other codeword, as every pair of codewords differ in at least $s+1$ places. Therefore, our code will detect an error as long as at most s changes occur in any codeword.

Similarly, if $d(C) \geq 2t+1$, then changing t entries in any codeword still means that it differs from any other codeword in at least $t+1$ many places; therefore, the codeword we started from is completely unambiguous, and we can correct these errors. □

Example. Using this theorem, we can see that both of our codewords can correct at most one error in any codeword, because their Hamming distances were both three.

Now that we've made this formal, we can now state our question rigorously:

Problem. Suppose that you are given a base q, a block length n for your codewords, and a minimum distance d that you want your codewords to be from each other (because you want to be able to correct up to $\lceil (d-1)/2 \rceil$ many errors in any codeword, for example.)

What is the maximum size of C — in other words, what is the maximum information rate you can get a code to have with these parameters?

Amazingly enough, this problem is wide open for tons of values! We really know very little about these maximum values: for example, when $n = 21, q = 2, d = 10$ this question is still open (the number of elements in C is between 42 and 47, according to http://www.win.tue.nl/ aeb/codes/binary-1.html. For general information on what we know and do not know, check out http://www.codetables.de/!)

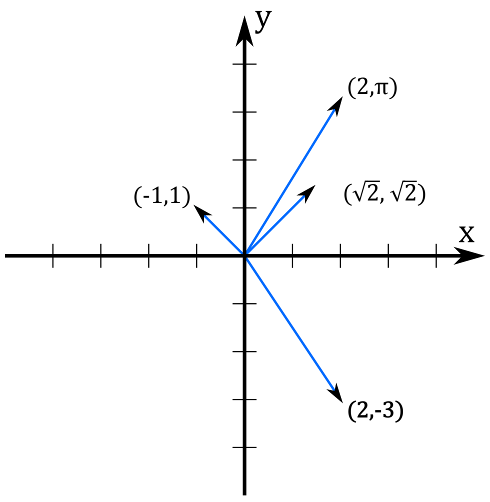
So: we have a mathematical object that on one hand seems incredibly practical (almost all modern electronical objects need some automated way to correct for errors) and also shockingly open (we don't know what the most efficient codes are for even very small block lengths n and distances d!) This, for me, is the sign of an exciting area of research: we both know nothing and want to be able to know everything!

To start our studies, though, we need to shore up our fundamentals. To study codes, we need to study the basic elements codes are made out of: codewords, i.e. elements of $(\mathbb{Z}/q\mathbb{Z})^n$! Perhaps unsurprisingly, these sets are full of rich structure in their own right; they form **vector spaces**, our topic of study for the next few classes!

2 Vector Spaces, Informally

(This section is duplicated from last quarter's Introduction to Proof notes; so, if you were OK with vector spaces back then, this should be OK here as well!) The two vector spaces you're probably the most used to working with, from either your previous linear algebra classes or even your earliest geometry/precalc classes, are the spaces $\mathbb{R}^2$ and $\mathbb{R}^3$. We briefly review how these two vector spaces work here:

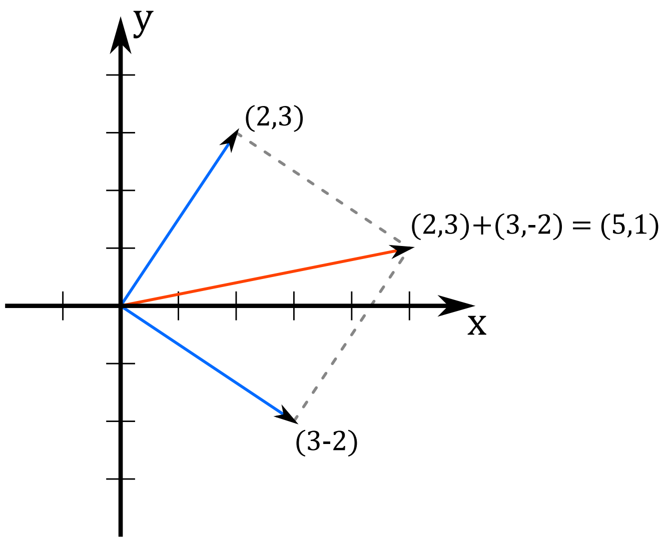
Definition. The **vector space** $\mathbb{R}^2$ consists of the collection of all pairs (a, b), where a, b are allowed to be any pair of real numbers. For example, $(2, -3), (2, \pi), (-1, 1)$, and $(\sqrt{2}, \sqrt{2})$ are all examples of vectors in $\mathbb{R}^2$. We typically visualize these vectors as arrows in the xy-plane, with the tail of the arrow starting at the origin[1] and the tip of the arrow drawn at the point in the plane with xy-coördinates given by the vector. We draw four such vectors here:



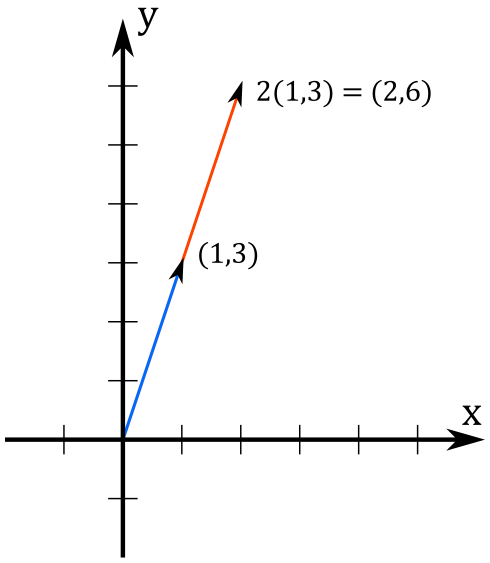
Given a pair of vectors in $\mathbb{R}^2$, we can **add** them together. We do this component-wise, i.e. if we have two vectors (a, b) and (c, d), their sum is the vector $(a+c, b+d)$. For example, the sum of the vectors $(3, -2)$ and $(2, 3)$ is the vector $(5, 1)$.

[1]The origin is the point $(0, 0)$ in the plane.

You can visualize this by taking the arrow corresponding to the first vector that we add, and "translating" this arrow over to the start of the second vector; if you travel along the first vector and then continue along this second translated vector, you arrive at some point in the plane. The arrow connecting the origin to this point is the vector given by the sum of these two vectors! If this seems hard to understand, the diagram below may help some:



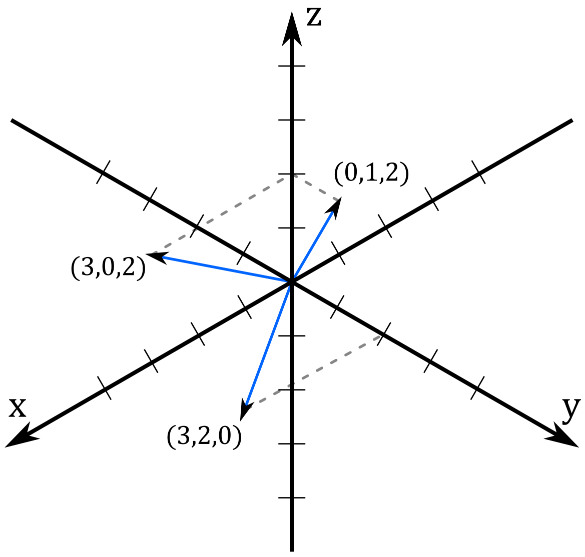
We can also **scale** a vector in $\mathbb{R}^2$ by any real number a. Intuitively, this corresponds to the concept of "stretching:" the vector (x, y) scaled by a, denoted $a(x, y)$, is the quantity (ax, ay). For example, $2(1, 3) = (2, 6)$, and is essentially what happens if we "double" the vector $(1, 3)$. We illustrate this below:



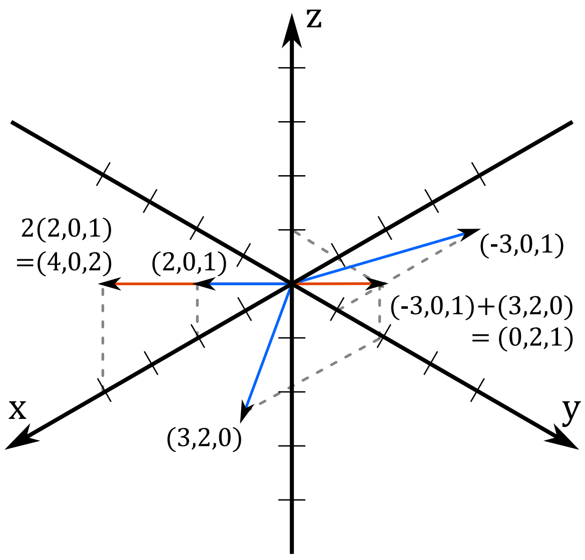
We can define $\mathbb{R}^3$ in a similar fashion:

Definition. The **vector space** $\mathbb{R}^3$ consists of the collection of all pairs (a, b, c), where a, b, c are allowed to be any triple of real numbers. For example, $(0, 1, 2), (3, 0, 2)$, and $(3, 2, 0)$

are all examples of vectors in $\mathbb{R}^3$. We typically visualize these vectors as arrows in three-dimensional xyz-space, with the tail of the arrow starting at the origin and the tip of the arrow drawn at the point in the plane with xyz-coördinates given by the vector. We draw three such vectors here:



Again, given a pair of vectors in $\mathbb{R}^3$, we can **add** them together. We do this component-wise, i.e. if we have two vectors (a, b, c) and (d, e, f), their sum is the vector $(a+d, b+e, c+f)$. For example, the sum of the vectors $(3, -2, 0)$ and $(2, 1, 2)$ is the vector $(5, -1, 2)$. We can also **scale** a vector in $\mathbb{R}^3$ by any real number a: the vector (x, y, z) scaled by a, denoted $a(x, y, z)$, is the quantity (ax, ay, az). These operations can be visualized in a similar fashion to the pictures we drew for $\mathbb{R}^2$:



You can generalize this discussion to $\mathbb{R}^n$, the vector space made out of n-tuples of real numbers: i.e. elements of $\mathbb{R}^4$ would be things like $(\pi, 2, 2, 1)$ or $(-1, 2, 1, -1)$. In fact, you can generalize this entire process to any arbitrary field F:

Definition. Take any set S. The **n-fold** Cartesian product of S with itself is the collection of all ordered n-tuples of elements of S: that is,

$$S^n = \{(s_1, s_2, \dots s_n) \mid s_1, s_2, \dots s_n \in S\}$$

Suppose specifically that S is actually some field F (examples of fields we studied last quarter: $\mathbb{R}, \mathbb{C}, \mathbb{Q}, \mathbb{Z}/p\mathbb{Z}$ whenever p is a prime, $\mathbb{F}_q$ for any prime power q via our equivalence-relation constructions.) We can define the operation of vector addition $+ : F^n \times F^n \to F^n$ on this set as follows: for any $\vec{f} = (f_1, \dots f_n), \vec{g} = (g_1, \dots g_n) \in F^n$, we can form

$$(f_1, f_2, \dots f_n) + (g_1, g_2, \dots g_n) := (f_1 + g_1, f_2 + g_2 + \dots f_n + g_n),$$

where the addition done on the right-hand side above is done using F's addition operation. Similarly We can also define the operation of scalar multiplication $\cdot : F \times F^n \to F^n$ as follows: for any $\vec{f} \in F^n, a \in F$, we can form the vector

$$a(f_1, f_2, \dots f_n) = (a \cdot f_1, a \cdot f_2, \dots a \cdot f_n),$$

where again the multiplication done on the right-hand side is done using F's multiplication operation.

3 Vector Spaces, Formally

In general, there are many other kinds of vector spaces — essentially, anything with the two operations "addition" and "scaling" is a vector space, provided that those operations are well-behaved in certain specific ways. Much like we did with $\mathbb{R}$ and the field axioms, we can generate a list of "properties" for a vector space that seem like characteristics that will insure this "well-behaved" nature. We list a collection of such properties and use them to define a vector space here:

Definition. A **vector space** V over a field F is a set V along with the two operations addition and scalar multiplication, such that the following properties hold:

- **Closure(+)**: $\forall \vec{v}, \vec{w} \in V$, we have $v + w \in V$.
- **Identity(+)**: $\exists \vec{0} \in V$ such that $\forall \vec{v} \in V$, $\vec{0} + \vec{v} = \vec{v}$.
- **Commutativity(+)**: $\forall \vec{v}, \vec{w} \in V, \vec{v} + \vec{w} = \vec{w} + \vec{v}$.
- **Associativity(+)**: $\forall \vec{u}, \vec{v}, \vec{w} \in V, (\vec{u} + \vec{v}) + \vec{w} = \vec{u} + (\vec{v} + \vec{w})$.
- **Inverses(+)**: $\forall \vec{v} \in V, \exists$ some $-\vec{v} \in V$ such that $\vec{v} + (-\vec{v}) = 0$.
- **Closure(·)**: $\forall a \in F, \vec{v} \in V$, we have $a\vec{v} \in V$.
- **Identity(·)**: $\forall \vec{v} \in V$, we have $1\vec{v} = \vec{v}$.
- **Compatibility(·)**: $\forall a, b \in F$, we have $a(b\vec{v}) = (a \cdot b)\vec{v}$.
- **Distributivity(+, ·)**: $\forall a \in F, \vec{v}, \vec{w} \in V, a\,(\vec{v} + \vec{w}) = a\vec{v} + a\vec{w}$.

As with fields, there are certainly properties that $\mathbb{R}^n$ satisfies that are not listed above. For example, consider the following property:

- **New property?(+)**: The additive identity, $\vec{0}$, is unique in any vector space. In other words, there cannot be two distinct vectors that are both the additive identity for a given vector space.

Just like before, this property turns out to be redundant: in other words, this property is implied by the definition of a vector space! We prove this here:

Claim. In any vector space, the additive identity is unique.

Proof. Take any two elements $\vec{0}, \vec{0'}$ that are both additive identities. Then, by definition, we know that because $\vec{0}$ is an additive identity, we have

$$\vec{0'} = \vec{0} + \vec{0'}.$$

Similarly, because $\vec{0'}$ is an additive identity, we have

$$\vec{0} = \vec{0'} + \vec{0}.$$

If we use commutativity to switch the $\vec{0}$ and $\vec{0'}$, we can combine these two equalities to get that

$$\vec{0} = \vec{0'} + \vec{0} = \vec{0} + \vec{0'} = \vec{0'}.$$

Therefore, we have shown that $\vec{0}$ and $\vec{0'}$ are equal. In other words, we've shown that all of the elements that are additive identities are all equal: i.e. that they're all the same element! Therefore, this additive identity element is **unique**: there is no other element that is somehow an additive identity that is different from $\vec{0}$. $\square$

As we did with fields, there are a number of other properties that $\mathbb{R}^n$ possesses that you can prove that any vector space must have: in your textbook, there are proofs that every vector has a unique additive inverse, that $0\vec{v}$ is always $\vec{0}$, that $-1\vec{v} = -\vec{v}$, and other such things.

Instead of focusing on more of these proofs, we shift our attention instead to actually describing some vector spaces!

A few of these are relatively simple to come up with:

- $\mathbb{R}^n$, the example we used to come up with these properties, is a vector space over the field $\mathbb{R}$.
- $\mathbb{C}^n$ is similar. Specifically: $\mathbb{C}^n$ is the set of all n-tuples of complex numbers: i.e.

  $$\mathbb{C}^n = \{(z_1, \ldots z_n) | z_1, \ldots z_n \in \mathbb{C}\}.$$

  Just like with $\mathbb{R}^n$, we can add these vectors together and scale them by arbitrary complex numbers, while satisfying all of the vector space properties. We leave the details for the reader to check, but this is a vector space over the complex numbers $\mathbb{C}$.

- Similarly, $\mathbb{Q}^n$, the set of all n-tuples of rational numbers

$$\mathbb{Q}^n = \{(q_1, \dots q_n) | q_1, \dots q_n \in \mathbb{Q}\},$$

  is a vector space over the field $\mathbb{Q}$.

- In general, given any field F, the set F^n along with the vector addition and scalar multiplication operations defined earlier, is a vector space!

  This is not hard to check:

  - **Closure(+)**: Immediate. Because F is a field and is closed under addition, the pairwise sums performed in vector addition must create another vector.
  - **Identity(+)**: Because F is a field, it has an additive identity, 0. The vector $\vec{0} = (0, 0, \dots 0)$ is consequently the additive identity for our vector space, as pairwise adding this vector to any other vector does not change any of the other vector's coördinates.
  - **Commutativity(+)**: Again, this is a consequence of F being a vector space. Because addition is commutative in F, the pairwise addition in our vector space is commutative.
  - **Associativity(+)**: Once more, this is a consequence of F being a vector space. Because addition is associative in F, the pairwise addition in our vector space is associative.
  - **Inverses(+)**: Take any $\vec{f} = (f_1, \dots f_n) \in F^n$. Because F is a field, we know that $(-f_1, \dots - f_n)$ is a vector in F^n as well. Furthermore, the pairwise addition of these two vectors clearly yields the additive identity $\vec{0}$; therefore, our vector space has inverses.
  - **Closure(·)**: This is a consequence of F being closed under multiplication.
  - **Identity(·)**: Because F is a field, it has a multiplicative identity 1. This 1, when used to scale a vector, does not change that vector at any coördinate because of this multiplicative identity property; therefore 1 is also the scalar multiplicative identity for our vector space.
  - **Compatibility(·)**: This is an immediate consequence from F's multiplication being associative, as for any $a, b \in F$, we have

$$\begin{aligned} a(b(f_1 \dots f_n) =& a(b \cdot f_1, \dots b \cdot f_n) = (a \cdot (b \cdot f_1), \dots a \cdot (b \cdot f_n)) \\ =& (a \cdot b) \cdot f_1, \dots (a \cdot b) \cdot f_n) = (a \cdot b)(f_1, \dots f_n). \end{aligned}$$

  - **Distributivity(+, ·)**: This is a consequence of F being a vector space. Because multiplication and addition are distributive in F, their combination in our vector space is distributive as well.

- A specific consequence of the above result is that something like $(\mathbb{Z}/5\mathbb{Z})^n$ is a vector space. This is a somewhat strange-looking beast: it's a vector space over a finite-sized field! In particular, it's a vector space with only finitely many elements, which is weird.

To understand this better, we look at some examples. Consider $(\mathbb{Z}/5\mathbb{Z})^2$. This is the vector space consisting of elements of the form

$$(a, b),$$

where $a, b \in \{0, 1, 2, 3, 4\}$. We add and scale elements in this vector space using mod-5 modular arithmetic: for example,

$$(2, 3) + (4, 4) = (1, 2),$$

because $2 + 4 \equiv 1 \mod 5$ and $3 + 4 \equiv 2 \mod 5$. Similarly,

$$2(3, 1) = (1, 2),$$

because $2 \cdot 3 \equiv 1 \mod 5$ and $2 \cdot 1 \equiv 2 \mod 5$.

Perhaps surprisingly, these odd-looking vector spaces are some of the most-commonly used spaces in the theoretical computer science/cryptographic settings. In particular, they come up very often in the field of elliptic curve cryptography, as you may remember from last quarter!

There are some odder examples of vector spaces:

- Polynomials! Specifically, let $\mathbb{R}[x]$ denote the collection of all finite-degree polynomials in one variable x with real-valued coefficients. In other words,

  $$\mathbb{R}[x] = \{a_0 + a_1 x + \ldots a_n x^n | a_0, \ldots a_n \in \mathbb{R}, n \in \mathbb{N}\}.$$

  Verifying that this is a vector space is not very difficult:

  - **Closure(+)**: Adding two polynomials together clearly gives us another polynomial.
  - **Identity(+)**: Adding 0 to any polynomial doesn't change it, and 0 is a polynomial itself (simply pick $a_0 = 0$ and $n = 0$.)
  - **Commutativity(+)**: We can add polynomials in any order that we want, and we'll always get the same answer. (This is because addition in $\mathbb{R}$ is commutative, and we just add polynomials by grouping common powers of x and adding their real-valued coefficients together!)
  - **Associativity(+)**: Holds for the precise same reason that commutativity holds.
  - **Inverses(+)**: Given any polynomial $a_0 + \ldots a_n x^n$, the polynomial $-a_0 + \ldots - a_n x^n$ is its additive inverse, as summing these two polynomials gives us 0.
  - **Closure(·)**: Multiplying a polynomial by a real number clearly gives us another polynomial.
  - **Identity(·)**: Multiplying a polynomial by 1 clearly gives us the same polynomial back.
  - **Distributivity(+, ·)**: Holds for the precise same reason that commutativity holds.

- Matrices! Specifically, let $M_{\mathbb{R}}(n,n)$ denote the set of $n \times n$ matrices with real-valued entries. For example

  $$M_{\mathbb{R}}(3,3) = \left\{ \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \;\middle|\; a,b,c,d,e,f,g,h,i \in \mathbb{R} \right\}.$$

  If we define matrix addition as simply entrywise addition: i.e.

  $$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \dots & b_{nn} \end{bmatrix} = \begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} & \dots & a_{1n}+b_{1n} \\ a_{21}+b_{21} & a_{22}+b_{22} & \dots & a_{2n}+b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1}+b_{n1} & a_{n2}+b_{n2} & \dots & a_{nn}+b_{nn} \end{bmatrix},$$

  and scalar multiplcation as simply entrywise multiplication, i.e.

  $$c \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} = \begin{bmatrix} ca_{11} & ca_{12} & \dots & ca_{1n} \\ ca_{21} & ca_{22} & \dots & ca_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ ca_{n1} & ca_{n2} & \dots & ca_{nn} \end{bmatrix},$$

  then this is a vector space! Specifically, it's a vector space for precisely the same reasons that $\mathbb{R}^n$ is a vector space: if you just think of a $n \times n$ matrix as a very oddly-written vector in $\mathbb{R}^{n^2}$, then every argument for why $\mathbb{R}^{n^2}$ is a vector space carries over to $M_{\mathbb{R}}(n,n)$.

  It might seem odd to think of matrices as a vector space, but if you go further in physics or pure mathematics, this is an incredibly useful and common construction. We leave the details of checking this is a vector space to the reader; it works just like everything else we've done thus far!

4 Using Vector Spaces to Study Codes

Let's return to error-correcting codes. Before, we mentioned that in general the following problem was open for most values of n, q, d:

Question 1. *Let $A_q(n,d)$ denote the maximum number of elements in a block-length n q-ary code C with $d(C) \geq d$. What is $A_q(n,d)$ for various values of q, n, d?*

As we stated before, this problem is open for most values of n, q, d: to give examples,

$$2720 \leq A_2(16,3) \leq 3276,$$

are the best bounds currently known for these parameters, and in general $A_2(n,k)$ is open for $n \geq 17, k \in \{3,4,5,6\}$.

Despite this, we can still find and determine lots of values for $A_q(n,d)$! Some, in fact, are downright trivial:

Theorem. $A_q(n, 1) = q^n$, for any q, n.

Proof. First, notice that any code C has distance at least 1; this is by definition, as

$$d(C) = \min_{c_1 \neq c_2 \in C} d(c_1, c_2),$$

and the smallest distance between any two nonequal words c_1, c_2 is at least 1 (because if they were distance 0, then they would be equal!)

Consequently, $A_q(n, 1)$ is just asking us for the maximum number of elements in a q-ary block-length n code, as the "distance 1" property is trivial to satisfy! Consequently, the code with the maximum number of elements is simply the code given by taking **all** of the possible code words: that is, it is the set $(\mathbb{Z}/q\mathbb{Z})^n$. This set has q^n elements, as claimed; so our proof is done! $\square$

It doesn't take much more work to prove the following:

Theorem. $A_q(n, n) = q$, for any q, n.

Proof. First, notice that if we take the code

$$\left\{ \overbrace{000\ldots0}^{n \text{ zeroes}}, \overbrace{111\ldots1}^{n \text{ ones}}, \ldots \overbrace{q-1\ldots q-1}^{n \text{ copies of } (q-1) \text{ symbols}}, \right\},$$

this code has q elements, all of which are distance n apart. So we have shown that $A_q(n, n) \geq q$ for any q, n.

So it suffices to prove that q is an upper bound for the size of any such code C, as well! To do this: take any q-ary block-length n code C with $d(C) \geq n$. Because C is a block-length n code, it is impossible for any two words to disagree in more than n places (as each word is length n); therefore, we can actually say that $d(C) = n$.

Suppose, for contradiction, that C has $q + 1$ or more words. Then, because there are q choices of symbol for the first symbol in each of the words of C, and there are $q+1$ words in total, there must be two words $c_1, c_2 \in C$ that start with the same symbol. But this means that $d(c_1, c_2) \leq n - 1$, because they can disagree in at most $n - 1$ places; in other words $d(C) \leq n - 1$, a contradiction! Therefore, we have shown that $|C| \leq q$, as claimed. $\square$

Working with other cases, however, gets harder in a hurry. Consider the following problem:

Question 2. *What is $A_2(n, 2)$ for arbitrary n?*

Answer. We start answering this problem by gathering some data first. For $n = 2$, this is a pretty simple problem; we've already shown above that $A_2(2, 2) = 2$.

Let's try $n = 3$ now. Assume without loss of generality[2] that the all-zero codeword 000 is in our binary block-length-3 distance 2 code C. What else can we put in this code?

[2] You'll prove that you can make this assumption in the homework!

Well, we could add in 111, to get the code

$$\{000, 111\};$$

by inspection, we cannot add any more words to this code, as any other word would either have one 1 (and thus be distance 1 from 000) or two 1's (and thus be distance 1 from 111.)

Alternately, instead of adding in 111, we could add in the three words 011, 101, 110, to get the code

$$\{000, 011, 101, 110\}.$$

This is a distance-2 code; all of the "two-1's" words are distance 2 from 000, and also distance 2 from each other (prove this if you don't see why!) We cannot add any more words to this code, as all remaining words have either one 1 or three 1's, and in either case are distance 1 from one of our "two 1's" words.

By exhaustion, then, we seem to have shown that $A_2(3, 2) = 4$. By using a similar technique, we can see that it looks likely that $A_2(4, 2) = 8$; if we start with the all-zero string 0000 and add in the "two-1's" words, we get

$$\{0000, 0011, 0101, 0110, 1001, 1010, 1100\},$$

to which we can also add the "four-1's" word 1111 to get

$$\{0000, 0011, 0101, 0110, 1001, 1010, 1100, 1111\}.$$

We clearly cannot add more words to this code, as any other word would have either one or three ones, and thus be within distance 1 of one of our existing codewords! Note that we haven't proven that this is the largest code possible; unlike our earlier work, we haven't considered other cases like adding in the "three-1's" words instead of the "two-1's" words. You can try (check it!) using those codewords, and verify that the resulting code is smaller than what we have here; however, it also seems likely that going through **all** of the other ways to grow this code would be tedious, and not like a great method if we want to generalize our results! Instead, we'll stick with intuition for now; it "looks like" 8 is the size of the largest code for $n = 4$, and hopefully we'll come up with a proof of this later.

Similarly, you could study $A_2(5, 2)$. To continue the pattern, you'd want the all-zero word 00000, all of the "two 1's" words, of which there are $\binom{5}{2} = 10$ (because you have five slots to put 1's in, and you're placing two 1's,) and all of the "four 1's" words, of which there are $\binom{5}{4} = 5$ (same reasoning.) This gives us $1 + 10 + 5 = 16$, which gives us an interesting pattern: we have

	$A_2(2,2)$	$A_2(3,2)$	$A_2(4,2)$	$A_2(5,2)$
Guesses	2	4	8	16

,

which might point to a conjecture of $A_2(n, 2) = 2^{n-1}$!

With this data collected, we transition from our guesses above to some actual proofs:

Theorem. $A_2(n, 2) = 2^{n-1}$.

Proof. As before, we prove this in two stages. First, we establish a lower bound of 2^{n-1} by giving a concrete example that shows this is possible. To do this, consider the binary block-length n code C formed by taking all of the code words with an even number of 1's. I claim that this code has distance 2. To see why, consider any two words w_1, w_2 with $d(w_1, w_2) = 1$. These two words disagree in exactly one place; so, outside of that one place, they have the same number of 1's, and at that place one is 1 and the other is 0. But this means that one of w_1, w_2 has exactly one more 1 than the other; in other words, it is impossible for both w_1, w_2 to have an even number of 1's (as an even number ± 1 is an odd number!)

So, we have shown that C is a distance-2 block-length n binary code. How many elements are in C? I claim that it is 2^{n-1}, or in other words that **half** of the total number of possible codewords[3] $(\mathbb{Z}/2\mathbb{Z})^n$ have an even number of 1's, and prove this by induction on n.

Our base case is immediate: if $n = 1$, then there is exactly one code word, 0, that has an even number of 1's. Inductively, suppose that half of the elements of $(\mathbb{Z}/2\mathbb{Z})^n$ have an even number of 1's, and the other half have an odd number of 1's. Look at the elements of $(\mathbb{Z}/2\mathbb{Z})^{n+1}$, and divide them into two groups: those whose last element is a 0, and those whose last element is a 1.

If you ignore the last element, then exactly half of the "last-element-zero" words have an even number of 1's, and half have an odd number of 1's; similarly, exactly half of the "last-element-one" words have an even number of 1's, and half have an odd number of 1's, by induction! Therefore, by taking the even-number-1's half from the last-element-zero words, and the odd-number-1's half from the "last-element-one" words, we have on one hand all of the "total number of 1's is even" numbers, and on the other hand half of the elements in total from $(\mathbb{Z}/2\mathbb{Z})^{n+1}$! So we've proven our claim by induction, and therefore proved that $|C|$ is precisely half of $|(\mathbb{Z}/2\mathbb{Z})^n| = 2^n$; i.e. $|C| = 2^{n-1}$, as claimed.

This does the first half of our proof; that 2^{n-1} is a lower bound. We now seek to prove that it is an upper bound, as well!

To see this, consider the concept of a **graph**:

Definition. A **graph** G with n vertices and m edges consists of the following two objects:

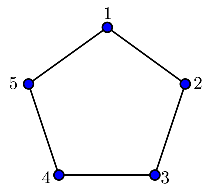
1. a set $V = \{v_1, \ldots v_n\}$, the members of which we call G's **vertices**, and
2. a set $E = \{e_1, \ldots e_m\}$, the members of which we call G's **edges**, where each edge e_i is an unordered pair of distinct elements in V, and no unordered pair is repeated. For a given edge $e = \{v, w\}$, we will often refer to the two vertices v, w contained by e as its endpoints.

Example. The following pair (V, E) defines a graph G on five vertices and five edges:

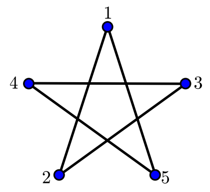
- $V = \{1, 2, 3, 4, 5\}$,
- $E = \{\{1, 2\}, \{2, 3\}, \{3, 4\}, \{4, 5\}, \{5, 1\}\}$.

[3] Because there are 2^n many words in $(\mathbb{Z}/2\mathbb{Z})^n$. Verify this if you forgot this proof from fall quarter!

Something mathematicians like to do to quickly represent graphs is **draw** them, which we can do by taking each vertex and assigining it a point in the plane, and taking each edge and drawing a curve between the two vertices represented by that edge. For example, one way to draw our graph G is the following:



However, this is not the only way to draw our graph! Another equally valid drawing is presented here:



In general, all we care about for our graphs is their vertices and their edges; we don't usually care about how they are drawn, so long as they consist of the same vertices connected via the same edges. Also, we usually will not care about how we "label" the vertices of a graph: i.e. we will usually skip the labelings on our graphs, and just draw them as vertices connected by edges.

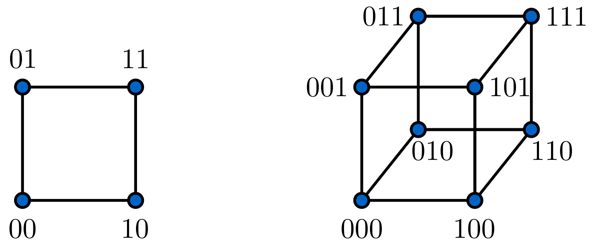
Graphs are one of the most versatile mathematical objects; it is hard to think of a concept of area of mathematics that cannot be described using the language of graph theory. When describing graphs, some additional terminology is handy: we list a few terms here.

- Two vertices are called **adjacent** if there is an edge connecting them. For example, vertices 1 and 2 are adjacent in the graph above, while vertices 1 and 3 are nonadjacent.
- Given a vertex v in a graph $G = (V, E)$, we define the **neighbors** of v, denoted $N(v)$, as the collection of all vertices that are adjacent to v. For example, $N(1) = \{2, 5\}$ in the graph above.
- In a graph $G = (V, E)$, we say that a vertex $v \in V$ has **degree** k if $N(v)$ contains k elements. For example, the vertex 1 in the example above has degree 2.

Returining to our problem, let's consider the following graph:

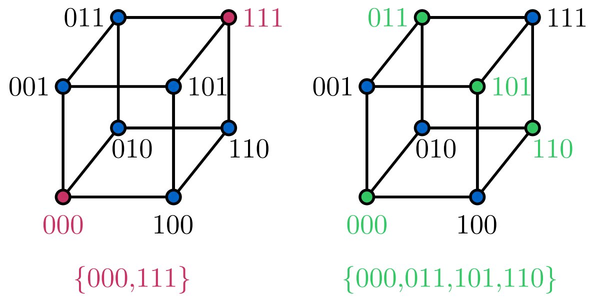
- **Vertices**: the set $(\mathbb{Z}/2\mathbb{Z})^n$.
- **Edges**: connect two words w_1, w_2 in our set with an edge if and only if $d(w_1, w_2) = 1$.

We draw a few examples of these graphs here:



Notice that these graphs are precisely just n-dimensional cubes! This shouldn't be too surprising — after all, the vertices of the unit cube in n dimensions are precisely all of the points in $\mathbb{R}^n$ whose coördinates are either 0 or 1 — but it's pretty nonetheless.

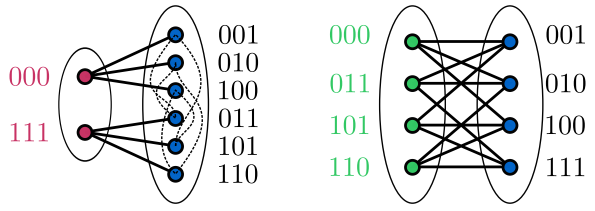
What I want to consider instead, here, is what **codes** correspond to on these graphs. In particular, suppose we take the graph above, and draw in the two distance-2 block-length-3 binary codes we came up with earlier:



Notice how in both cases, we never had any edges connecting two codewords! If you think about this for a moment, this makes sense: if two words are connected by an edge, then they are distance 1 from each other, which would cause a problem for any code in which all of our words are distance at least 2 from each other.

Also, notice that in the graph above, each vertex has degree n in the n-dimensional cube! This is again not hard to see: if we have any word w in $(\mathbb{Z}/2\mathbb{Z})^n$, there are exactly n words at distance 1 from w, as we get exactly one such word for each bit in w that we can change.

By combining these observations, we can prove our claim! To do this, take any block-length n distance-2 binary code C, and label it on the n-dimensional cube graphs above. Redraw our graphs by "clumping together" all of the codewords into one cluster, and the "non-codewords" into another cluster:



In the drawing above, the edges going from one cluster to the other are bolded, while the edges within a cluster (where they exist; they exist on the LHS, but not on the RHS) are dashed.

If you look at the "codeword" cluster, you can immediately see that the total number of edges connecting the codeword cluster to the non-codeword cluster is just

$$|C| \cdot n;$$

this is because there are $|C|$ vertices in the codeword cluster, each has n edges leaving it by our earlier observation, and all of those edges must go to non-codewords (again, by our earlier observations!)

Conversely, if you look at the "non-codeword" cluster and count edges there, you can see that the total number of edges from it to the codeword cluster is at most

$$|\{\text{non-codewords}\}| \cdot n,$$

because (again) each vertex has degree n! Notice that this time we don't know the exact number of edges going from non-codewords to codewords; not every edge from a non-codeword needs to go to a codeword, as our examples above demonstrate! However, we know that there are **at most** as many edges from non-codewords to codewords as our bound above,

Finally: we know that these two quantities are equal, because the edges that leave our codewords must go into the non-codewords, and vice-versa! Therefore, we have

$$|C| \cdot n \leq |\{\text{non-codewords}\}| \cdot n;$$

in other words,

$$|C| leq |\{\text{non-codewords}\}|.$$

But this means that there must be at least as many non-codewords as codewords; in other words, that C can contain at most half of the elements of $(\mathbb{Z}/2\mathbb{Z})^n$! In other words, we've proven that $A_2(n, 2) \leq 2^{n-1}$, which finishes the second part of our proof! $\square$

. . . so. You may have noticed that while we used the notation for vector spaces in places here, we didn't really use the structure of a vector space yet! We do this in the next week's notes; check them out when they go up!