Erick Fejta
Math 300A
April 18, 2002
Professor Bryan Smith

Error-Correcting Codes and Projective Planes

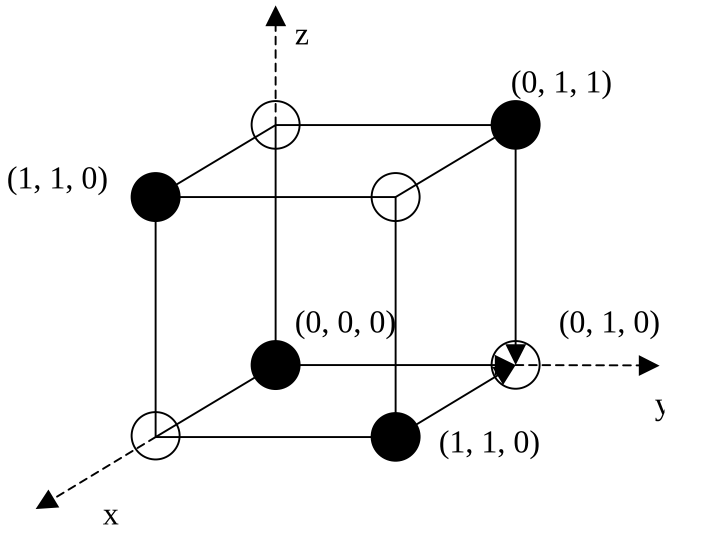
Computers were not always as stable as they are today. Our electronic technology had not yet reached a point where we could control so thoroughly the flow of data through electronic circuits, and as a result, data from one bit would on occasion leak out into another bit—changing the number two into a three for instance. Preventing this was extremely important. If the computer spends hours calculating some problem, for instance checking whether a number is a prime, one wants to have confidence in the result the computer returns. If it cannot be determined whether the computer had an error during the process and multiplied a number instead of divided then you have no way of knowing that 109433307*2^66452-1 actually is a prime. It could have just as easily turned a two into a three and returned four as a prime as well. Well by the late 1940s, it was common practice to use error-checking in calculations to make sure an error had not occurred. For instance, add a digit to the end that is calculated from the rest. This new digit that is computed from the other digits is called a check digit. There reason is because upon receiving the message, if the other digits calculate to a different check digit than what was sent with the message, you know you have an error. Thus it allows you to check whether the message contains an error. For example, the codeword for 45 could be 4+5 = 9 so you send the word 459. If the receiver gets 359, 3+5 = 8 < 9 so an error occurred. Similar binary methods were used with great success in the 1940s. Each message contained four binary digits and a fifth check digit that was added to make the number of 1s in the message even. Although this way was fast, computing was not as flawless as it could be because the computer would stop when it detected an error.

This was because these algorithms were only error-checking. In the above example, if we receive 359, we have no way of knowing what the correct message is. The intended message could have just as easily been 459 or 369, or for that matter 358 (and of course, if multiple errors occurred then it could have easily been 099, 189, 279, etc). Therefore when a computer detects an error, it has to either wait for an operator to correct the error or start over.

When compounded with the nature of computing at this time, it is easily seen why this is such a problem. Today for a few thousand dollars you can purchase a computer which can do billions of operations per second and can fit under your desk or on your lap. Back then, a computer “occupied about one thousand square feet of floor space and weighed some ten tons” (Thompson 16). They also cost millions of dollars (and as a side note, were less powerful than an early 21st century scientific calculator). As a result, access to the computer was shared among many people, and the computer was given a queue of problems to compute. Of course, a pecking order developed as to who got access to it when. People low on the pecking order had to run their programs at night, and during the night there were no operators employed to fix errors when they occur. Thus if you had to run your programs at night, when an error occurred it simply discarded your program and moved on to the next one. One person in such a situation was Richard W. Hamming.

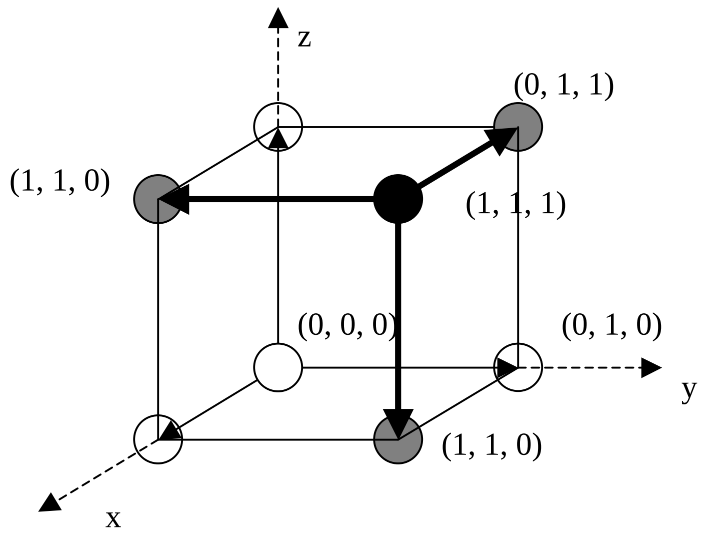
Hamming got frustrated at having to rerun the same programs, and set out looking for an efficient way to build a code where the computer could correct the errors and continue on. The difficulty with error-correcting codes is to keep the amount of redundancy as small as possible. In the above example, for every two numbers we send, we add a third for error checking purposes. Thus 2/3 = .66 so only 66% of the information sent corresponds to real information, which is called the information rate. We will also introduce the (n, r) notation, where n corresponds to the total number of digits, and r the number of message digits. The above example is a (3,2) code. Hamming eventually published a (7, 4) error-correcting code which has an information rate of 54% and gave the general formula $\frac{2^n}{n+1} \geq 2^r$.

Let us now consider how error-checking and correcting codes can be represented geometrically. If we are working in binary (all digits are either 1 or 0), then we can represent a (3, 2) single-error checking code by a three dimensional cube.



In this diagram the black dots represent valid codewords, and the other points are invalid ones (Thompson 9). It is (3, 2) because there are three digits (or bits as they are called in binary), one bit per axis, which make 2^3 = 8 potential codewords, but only 4 = 2^2 valid encodings. Also notice that if there is a single error on any codeword, it moves the location of the dot onto one of the white vertices. For instance, a single error on (0, 1, 1) will produce (1, 1, 1), (0, 0, 1), or (0, 1, 0). However, if a second error is added, then the location on the cube will go back to a valid codeword, ex. if given codeword (0, 1, 1) and the first and last bits are wrong upon reception, then the code goes to (1, 1, 1) and finally (1, 1, 0). Also, if there is a single error, (0, 1, 0), we cannot know where the error came from because (0, 0, 0), (0, 1, 1) and (1, 1, 0) are all valid candidates.

However, if we reduce the number of valid codewords to two in our cube example, we can produce an error-correcting code.



In this example, when an error is detected, it can correct the error by choosing the valid message that it is closest to by moving along the line segments. For example, (1, 1, 0) needs to only make one move to reach (1, 1, 1), but going to (0, 0, 0) takes two. Notice then that the distance between each valid point was one in the previous example, and two in this one. Therefore, if a single error occurred, the intended message was the valid point closest to the received point, which is (1, 1, 1) in this case. In order for a code to be error-detecting, the minimum distance between valid points must be at least 1, and to be error-correcting, this minimum distance must be at least 2. This minimum distance is called the Hamming distance. Although this example has a Hamming distance of two, on the down side, it is a (8, 2) code, which has an information rate of only 25%. This means that you transfer four times the normal amount of data information using this coding technique, which is too expensive. The (5, 4) error-recognizing code mentioned in the introduction was the most widely used encoding technique and had an information rate of 80%. Hamming knew that for error-correcting codes to be practical enough to implement, solutions needed higher information rates.

Hamming's first solution was an (8, 4) code with information rate 50%. He accomplished this using the idea of parity (Thompson 19). Parity is just a fancy name for binary check bits, similar to our first example where the check digit which is the sum of the previous two digits. However, parity is used in binary and thus parity bits are either 1 or 0, and they are chosen to make the rest of the word even. So 00 and 11 have a parity bit of 0, while 10 and 01 have an odd parity bit which makes the total even (011 = 0+1+1 = 2). Hamming took two of these two-bit words together and added parity checks on the rows and columns. The codewords 10 and 11 become

1	0	
1	1	

And we then add the parity bits to make each column and row even, yielding

1	0	1
1	1	0
0	1	

The final codeword is made by listing the contents of the rows, i.e. 101 110 01.

If there is an error in the transmission, and say 101 110 01 is received as 111 110 01, then using the tables we can discover where to make the correction.

1	1	1
1	1	0
0	1	

We see that the first row, 111, is odd, and the second column 111, is also odd. So the error is therefore in the first row, second column. Changing the 1 to a zero there, balances the table out.

Hamming was also able to construct a theorem as to the maximum potential information rate possible given a size n. Let k = n – r, which is the number of check bits. For an error-correcting code, the k bits must be able to describe n + 1 possibilities (Thompson 19). This is because the error can occur in any of the n bits, or there might be no error. Were this not the case, then we could not determine what bit of n the error occurred in. Since we are in base 2, we get $2^k >= n + 1$. Note that lg(X) is the standard computer science notation for $\log_{2(}X)$. Now we do some further manipulation, noting as you read that:

$2^k * 2^r >= (n+1)*2^r$ (multiply both sides by 2^r)

$2^{(k+r)} >= (n+1) * 2^r$

$2^n >= (n+1) * 2^r$ (k = n – r or n = r + k).

$(2^n)/(n+1) >= 2^r$ (divide both sides by n+1).

Finally n – lg (n+1) >= r (take the log base 2).

Using this, we can show that a (7,4) code can be constructed with information rate 53%, which is the highest of rate possible given only three check digits.

Let n = 7

Then 7 – lg 8 >= r

Thus 4 >= r, so there are 3 check digits.

Let n = 8

Then 8 – lg 9 >= r

Since r must be an integer, and 3 < lg 9 < 4, we can round up to 4.

Thus 8 – 4 >= 4, and there are 4 check digits.

Hamming then created a formula for constructing a code given n and r. He still used parity checks, but in a different way this time than rows and columns. The first parity check was on all bits whose decimal place (as in first, second third…) corresponded to a binary number that ended in 1, which are all odd numbers. The second parity check was on all bits whose decimal place corresponded to a binary number that had a 1 in the second place. For example 1**0**1 does not, but **1**0, **1**1, 1**1**0 and 1**1**1 all do.

The third parity check was on all bits whose decimal place corresponded to a binary number with a 1 in the third place (100, 101, 1100, etc). This process continues for the remaining number of k-bits. He also chose to place them in places that are a power of two (1, 2, 4, 8 …). So for our (7, 4) encoding, the check bits are at bits 1, 2 and 4 (Thompson 22-25). To encode the word 1101, we first put the digits into the encoded message, with blanks that we will fill with the parity checks. So we get:

1	2	3	4	5	6	7
_	_	1	_	1	0	1

Now we calculate the first parity bit, which include bits 3, 5 and 7. So we get 1+1+1 = 3, which is equivalent to 1 mod 2[A], so we put a 1 in the first bit to make the calculation even. The second bit includes 2, 3, 6 and 7. So we place 1+0+1 = 2 mod 2 = 0 in the second bit. The third parity bit includes 4, 5, 6 and 7, or 1 + 0 + 1 = 2 mod 2 = 0. This gives us 1010101 for the code word. Now suppose the second bit garbled and we receive 1110101 for this codeword. We can do the parity checks again to determine where the error occurred.

First, 1 + 1 + 1 + 1 = 4 mod 2 = 0 for the odd positions.

Second, 1 + 1 + 0 + 1 + 3 mod 2 = 1 for positions 2, 3, 6 and 7.

Third, 0 + 1 + 0 + 1 = 2 mod 2 = 0 for positions 4, 5, 6 and 7.

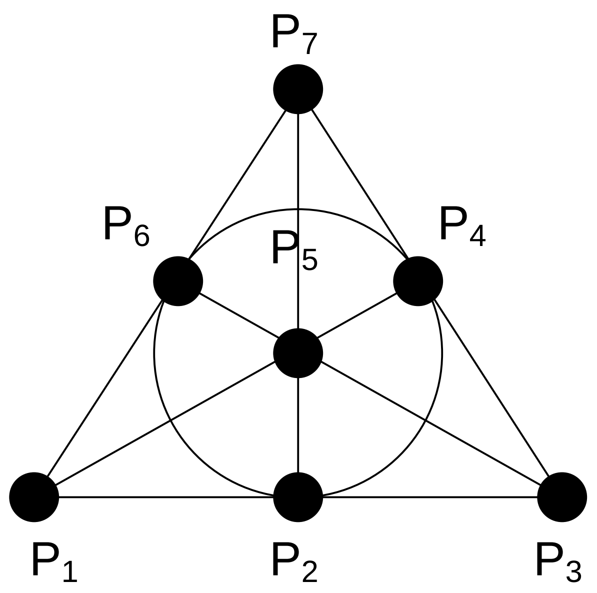
Cleverly, 010 corresponds to the bit that is in error (2 in decimal). Flipping the second bit gives us 1010101, our original message. Since we chose the parity bits to make the

[A] Mod is a common computer term that can be thought of most easily as a function that calculates the remainder. 3 mod 2 = 1 because 3/2 = 1 with a remainder of 1. Thus base B, the result will be 0 <= x < B. Another concrete example is how we handle degrees. 1124° ≡ 44° = 1124 mod 360.

addition calculations even (or 0 mod 2), if there is no error then the parity checking will return 000.

Armed with this discovery Hamming predicted that stops due to errors would become extremely rare. He gave a figure that they could run for 23,000 hours before stopping for an error, and an error will go undetected but every $1150 * 10^8$ hours. Thus people with low priority access to computers everywhere rejoiced in the 1950s as his discovery was incorporated into computers and they need not fear the machine spitting back your data unfinished because it ran across an error.

Although calculations such as these are precisely how a computer verifies and corrects messages, they are not very intuitive to the human eye. People prefer concrete models, such as geometric representations. Luckily the Hamming (7, 4) code is equivalent to a projective plane of order 2. Such a projective plane has seven points and seven lines, with each line having three points. Since it is a projective plane, we know that the incidence axioms hold and no lines are parallel. A model of this projective plane is given below:



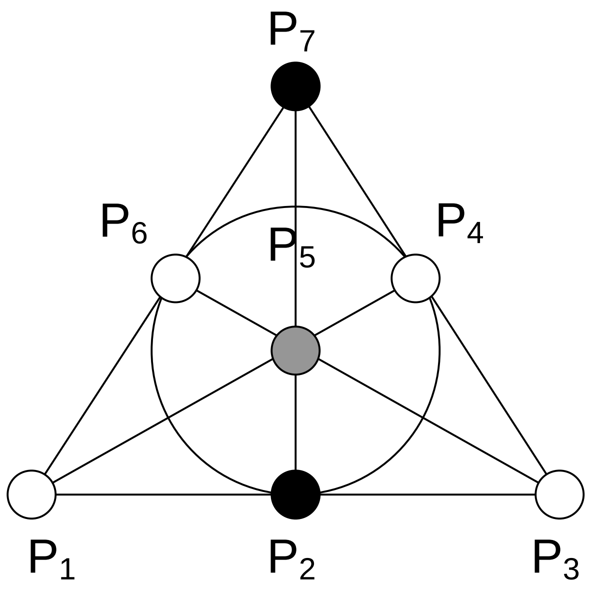
The way to convert a seven bit message into its equivalent geometric interpretation is easy. The bits describe a set of points in the projective plane. If a bit's value is 1 then that point is included in the set and 0 means it is not. So if the first bit is 1, then the set contains P_1 and the second bit determines whether P_2 is in the set, and so on. For example, the message 1010111 describes the set $\{P_1, P_3, P_5, P_6, P_7\}$ and 0000000 describes the empty set {}. Only certain sets are correct, error-free messages. These include the messages describing the empty set, the full set, lines and the complement of lines. Since there are seven lines, and their seven complements, this gives a total of sixteen valid messages when added to the empty and full sets. The Hamming (7, 4) code has only four message bits, which allows for a total of $2^4 = 16$ unique messages, so this is correct.

If the projective plane is equivalent to a (7, 4) code, it must be able to detect and correct any single error that occurs in the transmission of a valid message. Depending on the number of points in the set represented by the message, we will show in each case how to detect if the message is valid and how to use the geometric interpretation to correct invalid messages.

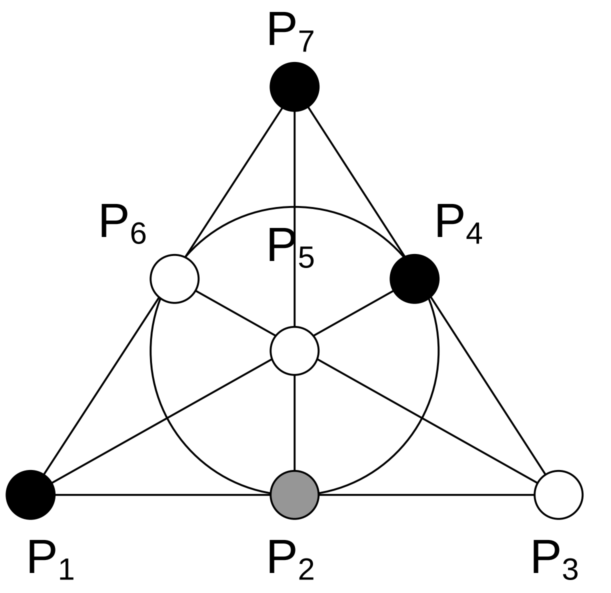
The empty set is a valid message, so no correction is necessary.

Sets of one point occur when an error occurs transmitting the empty set. However, since its bit equivalent is 0000000, if an error occurs, a 0 will have to change to a 1. Thus we will receive a set of a single point. Removing that single point, which changes the 1 back to a 0, fixes the error.

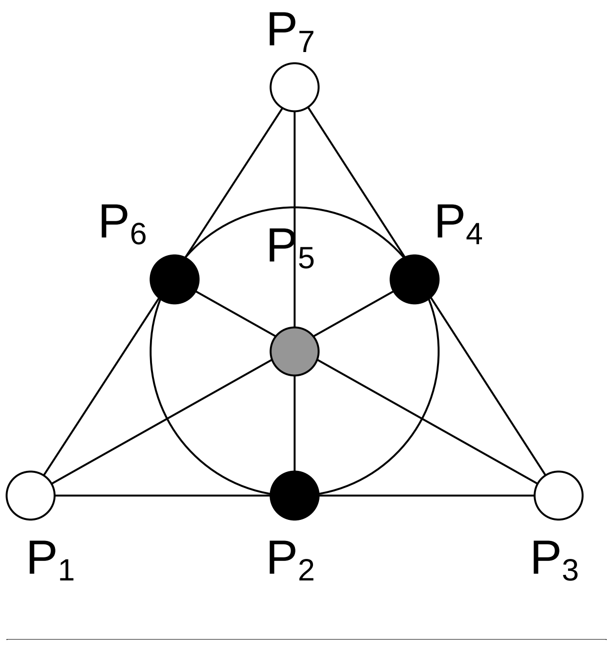
Sets of two points occur when the transmission of a line results in the removal of one of the lines. We are left with two points. However, recall from Incidence Axiom 1 that two distinct points describe a unique line (Greenberg 51). Since we know which line the two points belong two, we can look at the model to find the missing third point of the line and add it back to the set. For example the message 0100001 describes the set {P_2, P_7}. Looking at the model we can clearly see that the missing point is P_5. So the correct set is {P_2, P_5, P_7}, which is the message 0100101.



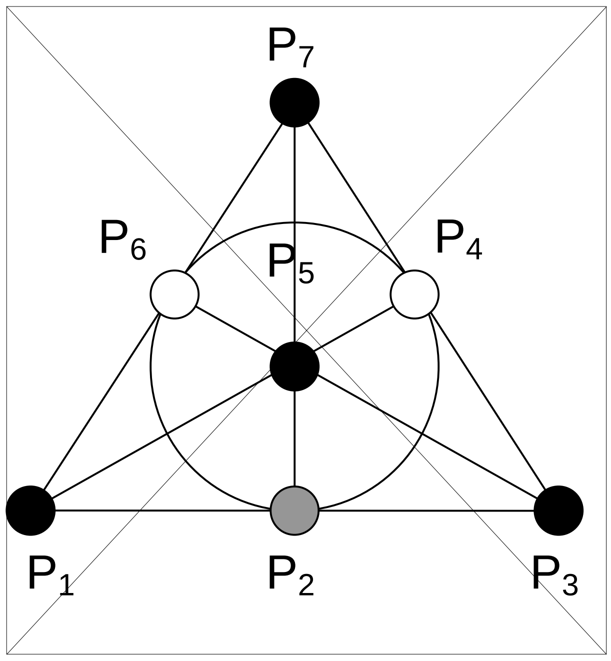
When there are three points in the set, either the set is a line, in which case the set is valid, or a point was removed from the complement of a line. In the latter case, we simply add the point back to make the complement. Because the three points describe three lines, three of the remaining points will make a complete line. Simply choose the only one remaining to get the correct point to add back. If the set $\{P_1, P_4, P_7\}$ is received, adding P_3, P_5 or P_6 would create the lines $\{P_3, P_4, P_7\}$, $\{P_1, P_4, P_5\}$ and $\{P_1, P_6, P_7\}$, respectively. So therefore the correct point to add is P_2.



Four point sets are similar to three point sets. Either the four points are the complement of a line and valid, or they are three collinear points making a line plus one other point. The latter case has an error and removing the extra point corrects it. Referring again to our model makes it obvious which to remove. The set $\{P_2, P_4, P_5, P_6\}$ contains the line $\{P_2, P_4, P_6\}$, so remove P_5.



Five point sets occur when a point is added to the set of a complement of a line. The five points will contain two lines. Remove the point that is incident to both. The set $\{P_1, P_2, P_3, P_5, P_7\}$ contains the lines $\{P_1, P_2, P_3\}$ and $\{P_2, P_5, P_7\}$. P_2 is incident with both, so remove it and we have four points which do not describe a line, meaning the remaining three do. Thus we have the complement of a line.



Sets of six points are similar to the single point set example, except that to correct the error you add the missing point back to get the full set. This is because the full set contains every point, so an error has to remove one of them.

The seven point set is the full set, which is a valid message.

Before we can conclude that we can use this model to correct single errors, one issue remains. We must be sure that an error in the transmission of the complement of line, which has four points, will not generate a line, which has three points. This will not happen because by definition the complement of a line does not contain all the points of any line. So in order to get all the points of a line we have to first add a new point to complete a line, and then remove the two points not in the line. This amounts to a total of three errors, whereas we only need to worry about catching and correcting a single error. Reversing this process it follows that we cannot get four points that are the complement of a line from a single error in a three point line; we need three. Also to get from a null set to a line, we need three errors to add three points, and we need three errors to remove three points from the full set to get four points. Therefore we can be sure that we can use the model to correct any single errors.

Bibliography

Canavan, Amy. "Hamming Codes." Fall 1999. University of Colorado at Denver. May 1, 2002. http://www-math.cudenver.edu/~wcherowi/courses/m6221/fall99/moons_over_my_hamming_code.pdf

Greenber, Marvin J. *Euclidean and non-Euclidean geometries: development and history*. 3rd Edition. New York: W. H. Freemand and Compnay, 1993.

Thompson, Thomas M. *From Error-Correcting Codes Through Sphere Packings to Simple Groups*. Walla Walla: The Mathematical Association of America, 1985.