Afzal Khan

**Writeup**

With the completion of the Hamming code algorithm, there are many characteristics to touch upon. The implementation of the hamming code algorithm follows the Hamming(8, 4) code. In this format, the lower four bits of the byte are assigned to the *msg* bits whereas the upper four bits are designated as the parity bits. Instead of using the parity equations to calculate the parity bits one by one, we employ the use of the *BitMatrix* ADT.

This structure is strongly dependent on another ADT known as the *BitVector* ADT. The bitvector structure essentially converts bytes into a vector of either 4 bits in length or 8 bits in length. The vectors than allow for operations to be called on them to allow the manipulation if bits. For instance, flipping bits, logical shifting bits, and clearing bits are all function made capable by the bitvector ADT. The bitmatrix structure takes this vector characteristics and converts them into a matrix representation. This allows for matrix operations to be conducted on the bytes.

Since individual calculating parity bits can be hard to implement, we use a generator matrix *G* to encode our messages. The generator matrix allows for a message *m* to be converted into a hamming code, *c*. The 4 x 8 generator matrix can use matrix multiplication with the matrix representation of *m*, a 1 x 4 matrix. The formula is as follows, $\vec{c} = \vec{m}G$.

Decoding messages follows a similar idea in reverse. Instead of using *G*, the matrix is used is know a parity-checker matrix. The transpose of the parity-checker matrix, $H^T$, is what will be multiplied to the encoded code, *c*. This will result in a matrix known as the *error-syndrome*. The formula is as follows, $\vec{e} = \vec{c}H^T$.

*Encoding Lookup Table*

The simplest way to decode a hamming code is by using a look up table. Essentially all possible values for the *error-syndrome* are added to an array. When the nibble is retrieved, it is checked against this table to determine if an error exists in the *msg*, and if so, on which bit. However, in order to make this code effective, a lookup table for encoding should also be enables.

Essentially, the encoding lookup table is a form of *memorization*. This means that as values are calculated they are *cached* to be quickly accessed again later. The encoding function utilizes the *bm_multiply* function, which allows to matrix to be multiplied and returns the resulting matrix. However, this function runs in $O(n^3)$ time complexity. On extremely large files this can quickly get taxing for the system. In order to combat this issue, I *cached* my encoding results. Since we are only encoding a nibble at a time or (0 0 0 0), there are only 16 possible values that these bits can hold (0000, 0001, 0010 … etc.) I created an array of 16 values where each index corresponded to the possible nibble value. Then each index was assigned its respecting encoded hamming code. As a result while encoding a file, when a nibble was split, it was a matter of looking it up in the encoding look up table. Having to call *ham_encode* twice in every iteration of a file character would have take an extremely large amount of time for files such as *bible.txt*.

*Entropy*

Entropy simply put is the disorder or chaos in an environment. It is comparable to the idea of chaos or randomness. When something is said to have a very high entropy, you can expect for it to be unpredictable and unordered. In order to investigate this concept, I decided toe feed error to my encoding program. Using the provided *entropy.c* program I was able to measure the entropy of the infile. In order to best investigate this are, I decided to take two different files. The first was the *grammar.lsp* file in the *corpora* folder. This Lisp source file is a moderately sized file with numerous characters. The second file is the *bible.txt*. This is a text file version of the Bible and is extremely. I wanted to investigate if the length and type of character in the file had any effect of the measure entropy. The following is my results.

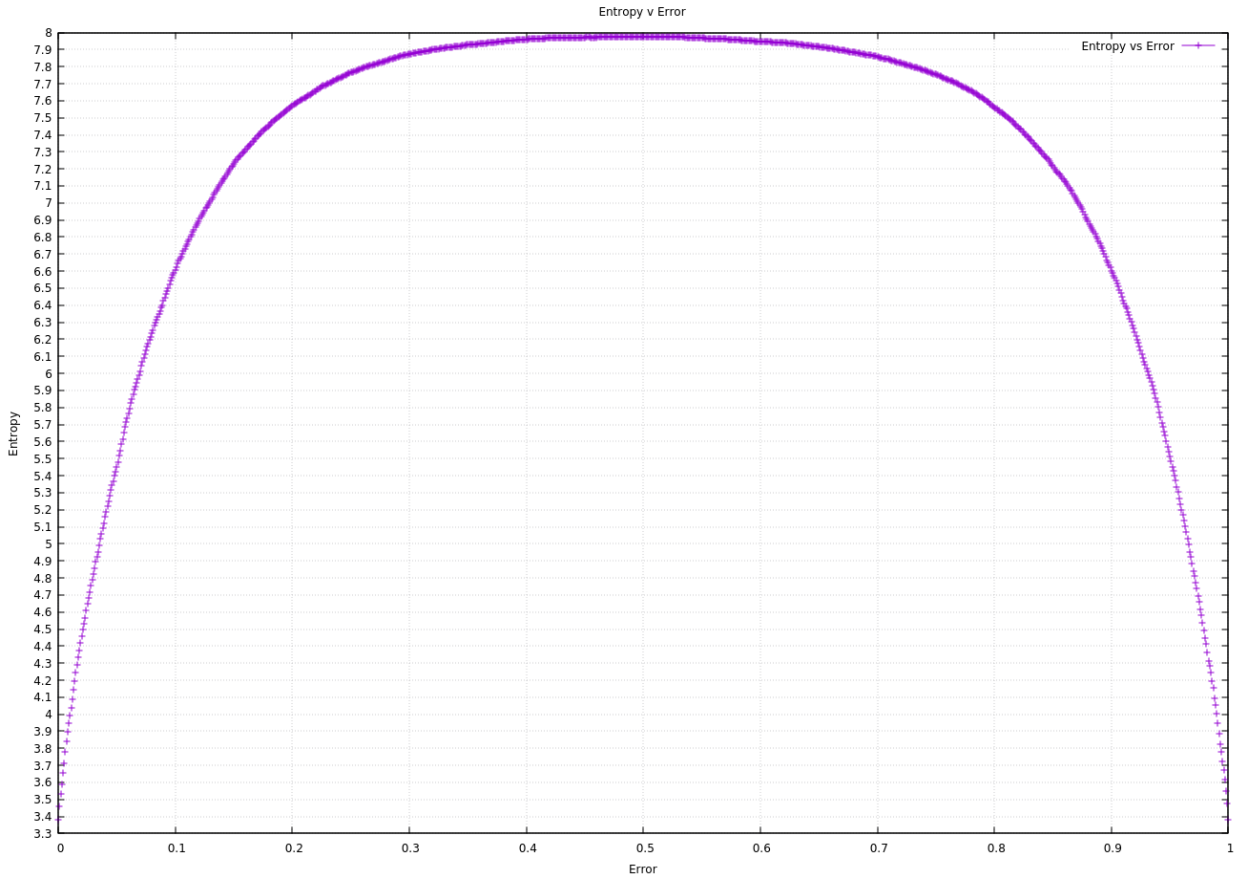



Figure 1. *grammar.lsp*

Using error steps of 0.001, I managed to record the change in entropy as the error inputted reached 1 for the *grammar.lsp* file. Curiously enough, after 50 percent error, the entropy of the system began to decrease. I decided to try the larger, Bible text file to see if this anomaly persisted.

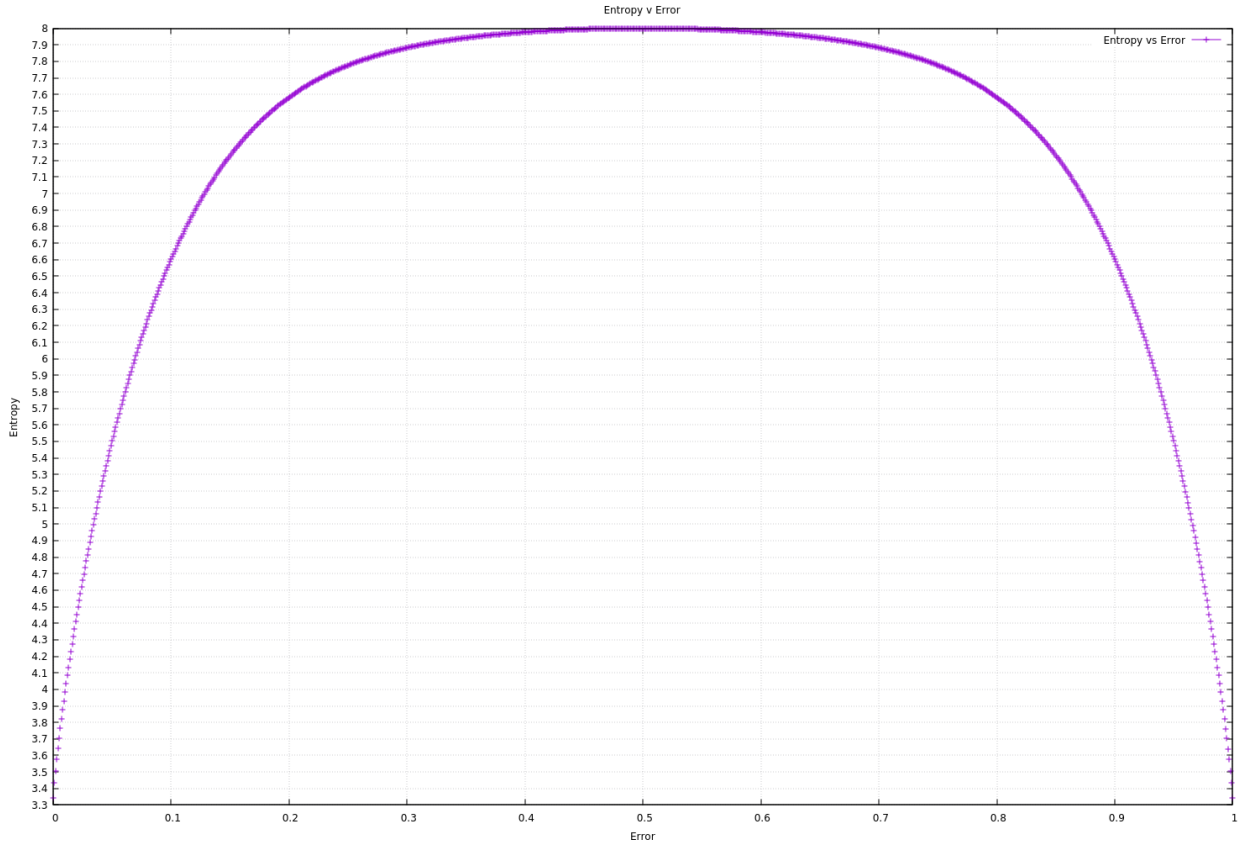



Figure 2. *bible.txt*

The parabola like curve existed in the *bible.txt* file as well. In both graphs, in the range 0 to 1 the line does not exist when entropy is equal to 0. This makes sense since even an un-encoded file is going to have some form of randomness regardless of the length or type of the file. However, there are some small differences between the two.

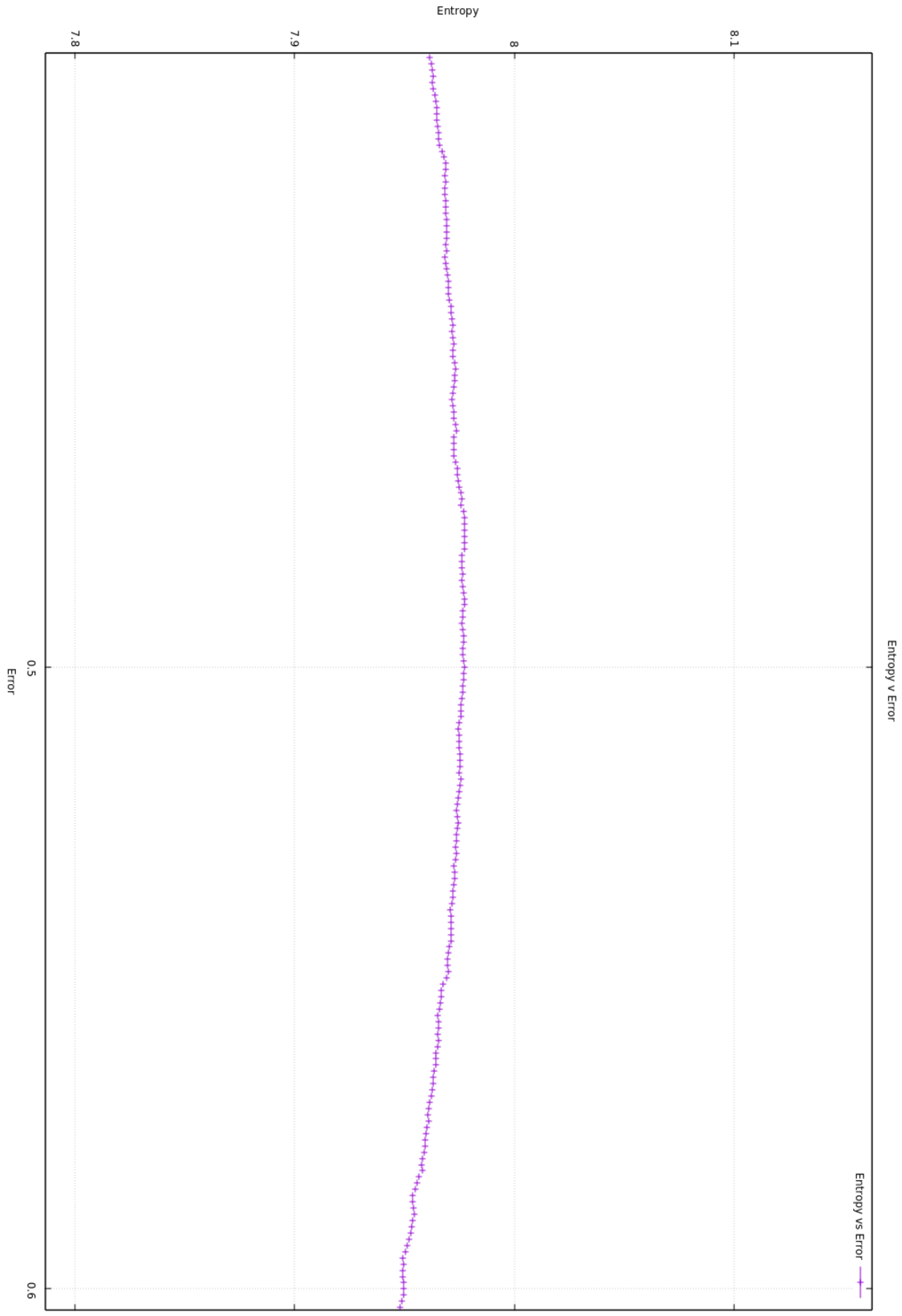



Figure 3. *grammar.txt* zoomed

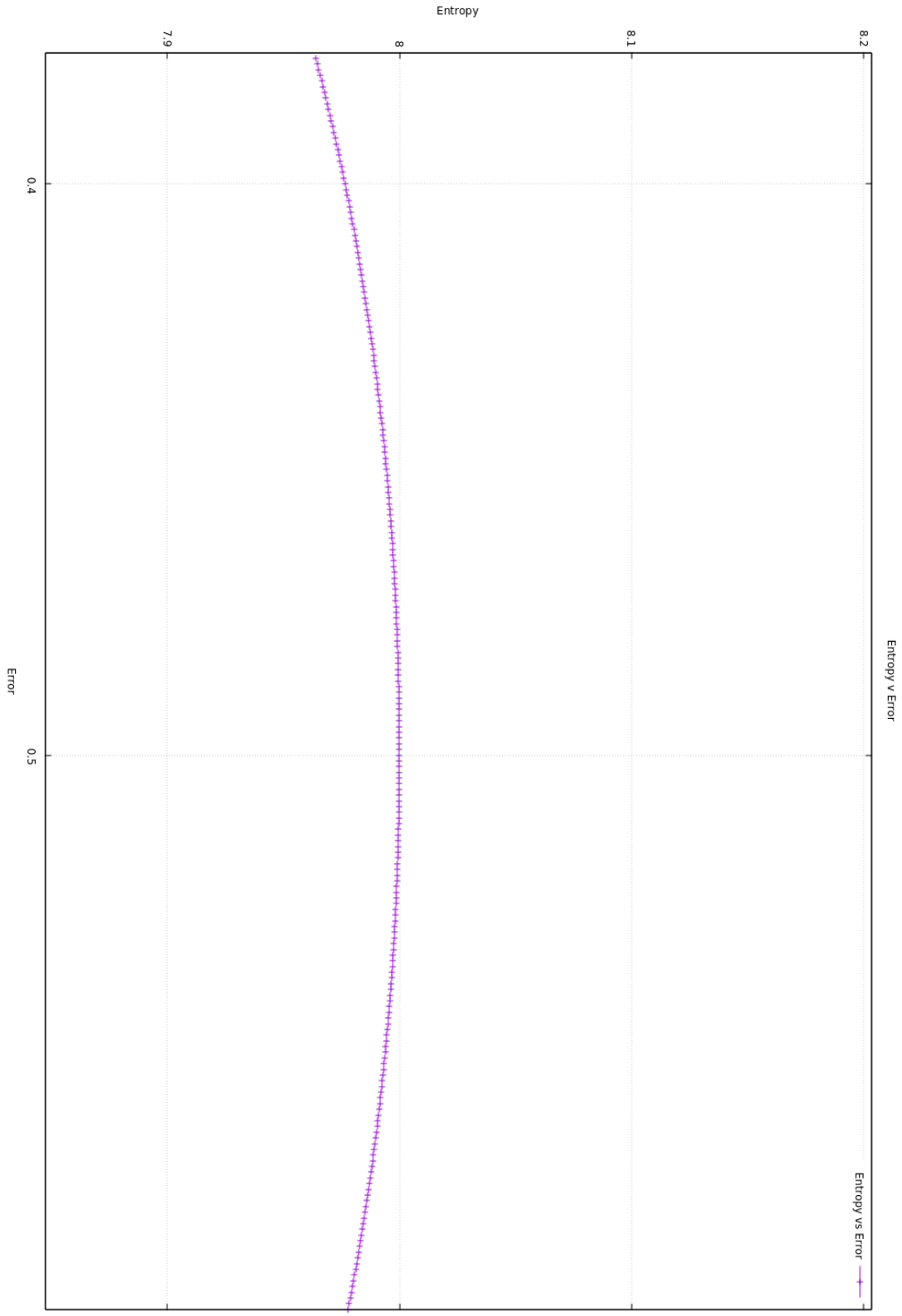
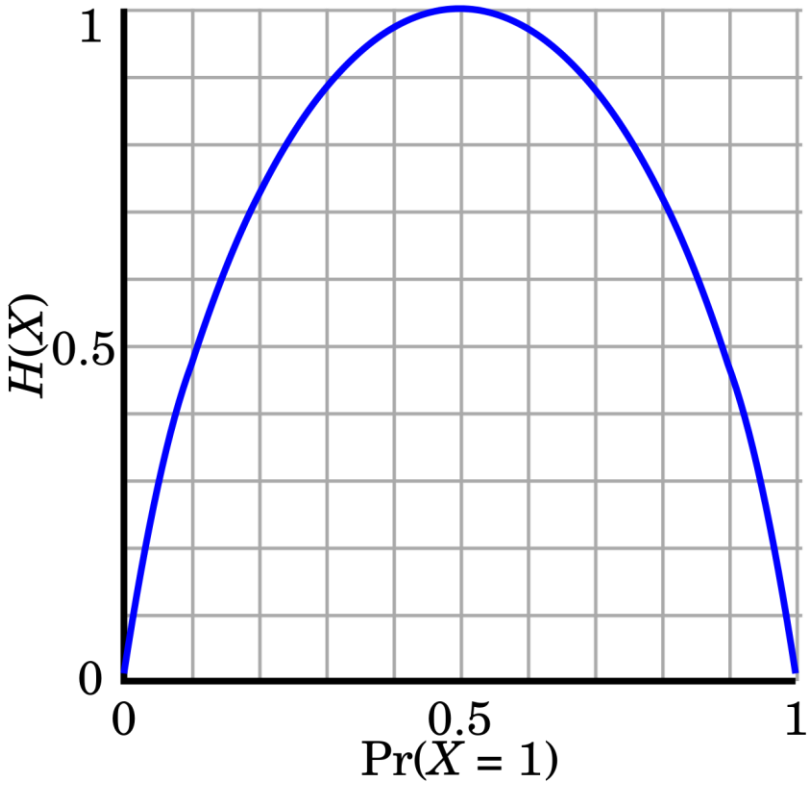



Figure 4. *bible.txt* zoomed

The entropy for *grammar.lsp* peaked at a little less than 8.0. However, in the *bible.txt* the entropy managed to reach a max entropy of 8. This is most likely due to the differences in the length of the files. It is possible that as the size of the file exists, the threshold for the max amount of entropy increases as well.

However, none of this explain the shape of the line. Upon investigating the calculation of entropy and the source code file for *entropy.c*, I came across the formula for *Shannon Entropy* (Wikipedia Entropy). Named after mathematician Claude Shannon, the formula illustrates the calculation of entropy using the summation of probabilities, *P*, for some discrete variable *X*. Having remembered some of my AP Statistics class in high school I understood the modeling of the equation using Bernoulli's principle. According to Wikipedia, a coin flip can model the change in entropy. They say, "… entropy of the unknown result of the next toss of the coin is maximized if the coin is fair (that is, if heads and tails both have equal probability 1/2)." In other words when the probability is exactly fifty-fifty, the entropy for the system is maxed. This perfectly explains why the when the probability, or in this case error, was at 0.5, the program recorded its peak entropy.




*-Wikipedia on Entropy*