Vansh Kumar
EE/Ma/CS 127

Final Project Report

I chose to implement narrow-sense ($b = 1$) Reed-Solomon codes of field size q, where q is a prime number. The specific codes that I chose to analyze were:

- $q = 23$ and $t = 2$, a (22, 16)-code that corrects any 2 errors (primitive element 2)
- $q = 47$ and $t = 4$, a (46, 38)-code that corrects any 4 errors (primitive element 5)
- $q = 73$ and $t = 8$, a (72, 56)-code that corrects any 8 errors (primitive element 5)

Finite Field Arithmetic:

For overall finite field arithmetic that I knew I would have to do throughout, I created a Python class to handle it. It is called *GFInt* and extends the built-in *int* class - the only internal value it stores is called *self.integer* and is just an integer value, as every element of the field of Z mod q can be mapped to the integers. Features of this class:

- Addition (and by extension subtraction) are just adding two integers and modding by q.
- For multiplication and division, I used the fact that a primitive element *alpha* generates the field and created exponential and log tables that could be used to simplify these operations. Multiplication then just became exponentiating the sum of the logs of the input arguments mod q, and division was similarly simplified.
- Exponentiation was implemented in the same way, with the exponentiation table value of the log of the integer times the exponent mod q-1.
- Taking an inverse also became a table lookup, being the exponentiation of q - 1 - log[integer]
- Comparisons (==, !=, <, and >) were also implemented, based solely on the normal comparisons for integers

Polynomial (lists of *GFInt* values) Arithmetic:

- For polynomial addition (and by extension subtraction), I implemented a separate function called polyAdd that simply added two lists element-wise (the fact that each element was a *GFInt* abstracted away the finite field arithmetic)
- For polynomial multiplication, I used a very simple implementation that just multiplied every term of one of the polynomials by the other polynomial and summed the results of each step
- For polynomial division, I did long division and returned the quotient and remainder

Generator Polynomial:

To compute the generator polynomial, I used my polynomial multiplication function to essentially multiply out $g(x) = (1\text{-}alpha)(1\text{-}alpha^{2})\cdots(1\text{-}alpha^{2t})$ and just return the result.

Encoding:
I used a systematic encoder, the one that we implemented in homework 4. Given a message polynomial *m(x)*, it simply returns $c(x) = x^{2t}m(x) - R_{g(x)}[x^{2t}m(x)]$, where $R_{g(x)}[x^{2t}m(x)]$ denotes the remainder when $x^{2t}m(x)$ is divided by *g(x)*, the generator polynomial.

Channel:
My channel was fully described by P_s, a probability such that:
$P(e_i = a) = \{1 - P_s$ if $a = 0$; $P_s / (q - 1)$ otherwise$\}$, where we are given c_i, a codeword, and are computing $y_i = c_i + e_i$. The way this was actually implemented was by computing a table of probabilities for each possible error value *a* such that the value at index *a* corresponded to the probability that $e_i = a$. Then, a random number between 0 and 1 was computed and whatever "bin" it fell into in this probability table defined what the error value was for that error location.

For example, if we had probability table [0.4, 0.55, 0.7, 0.85, 1] and got random number 0.6, it would fall between 0.55 and 0.7 in the table and so would have an error value of 2, as that is the index of 0.7 in the table.

Decoding:
For decoding, first I computed the syndrome of the received vector *y*. This simply consists of evaluating the received vector and having S_j as the evaluated value of *y* at $x = alpha^j$.

Then, I calculated the error locator and error evaluator using the resultant syndrome and the extended Euclidean algorithm. I implemented the Euclidean algorithm by doing polynomial division at every step and keeping track of coefficients at each step. I then calculated the error locator polynomial by scaling the coefficient polynomial *A* of *S*, the syndrome, to get the leading constant to be 1. The error evaluator polynomial was then calculated by computing $S(x)\Delta(x)$ *mod* x^t. If a divide by zero error is encountered because the scale factor, the leading constant of *A(x)*, ends up being zero, the error locator and error evaluator are both set to *None* to indicate decoder failure.

Once the error locator and evaluator have been computed, a Chien Search is performed, essentially looking through all field elements to find which ones are roots of the error locator polynomial. If less roots are found than the degree of the error locator polynomial, then decode failure is declared.

Then, Forney's algorithm is run to actually compute error locations and values. Finding error locations is easy, it is just inverting the roots of the error locator polynomial. Finding error values requires taking the formal derivative of the error locator polynomial, and the computing $-\Omega(x_i)/\Delta'(x_i)$, where x_i denotes the ith root of the error locator. If the derivative of $\Delta(x)$ evaluated at x_i ends up being zero, decoder failure is declared and *None* is returned for both the *locations* and *evals* lists to indicate this.

Finally, the received vector *y* was actually fixed by looping through the error locations and error values and subtracting e_i from y_i at the error locations *i*. One final chance for decoder failure was if an error locations happened to be out of bounds of the received polynomial *y*.

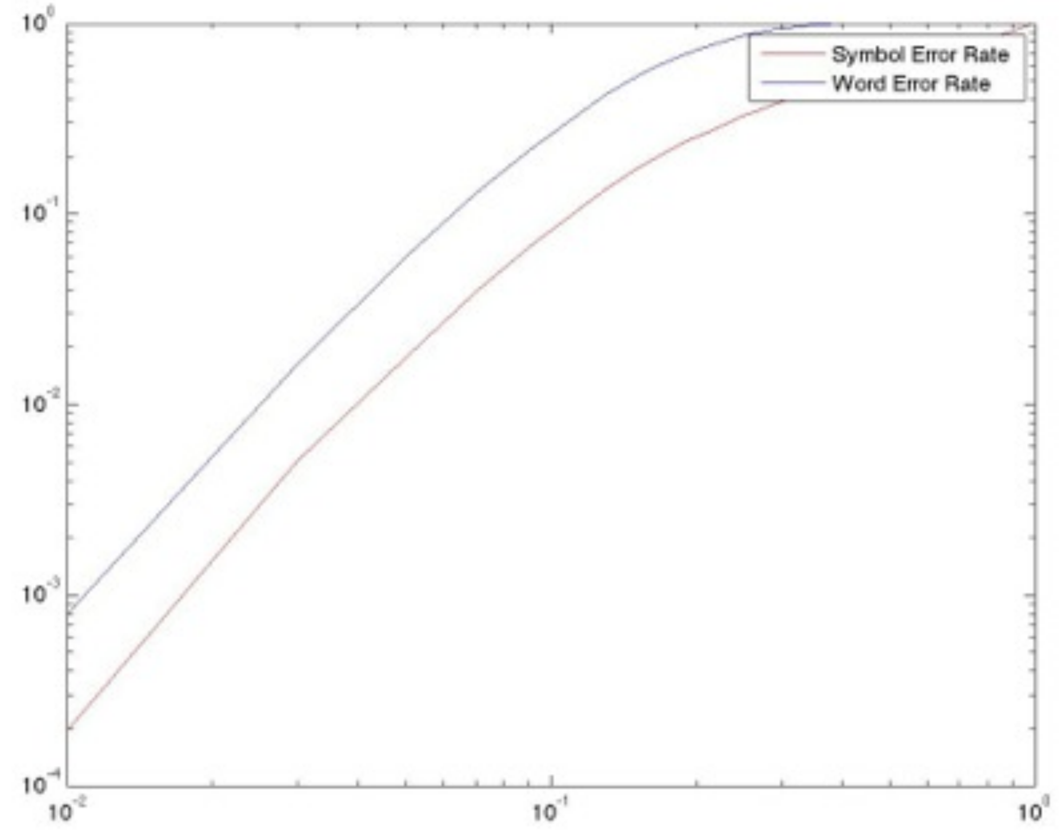
Simulation:

For simulation, random messages of length *k* were encoded, sent through the channel, and then decoded. Probabilities from 0.01 to 1 with a spacing of 0.02 were used. Each probability was tested with 20000 random messages to calculate the symbol error rate and word error rate for each probability. For a given message and encoding/channel/decoding process, the number of symbol errors is the number of places where the given message and the decoded message differ and the number of word errors is 1 if the decoded and encoded codewords differ and 0 otherwise. Then, the number of symbol errors is divided by $n * k$, where *n* is the number of points tested for a given probability, to give an error rate and the number of word errors is divided by *k* to give an error rate.

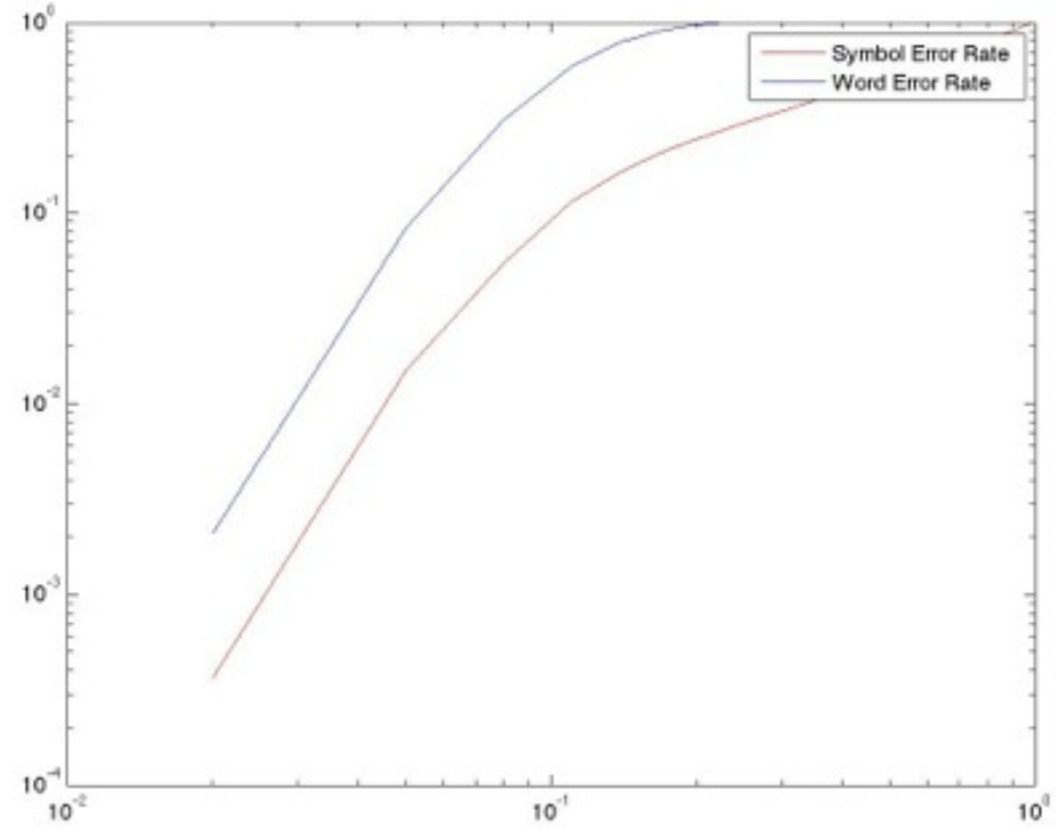
Results:

Note that all codes have ≈ the same rate so they are theoretically comparable in that sense.
The x-axis of all of these graphs is probability of error (P_s) and the y-axis is error rate.

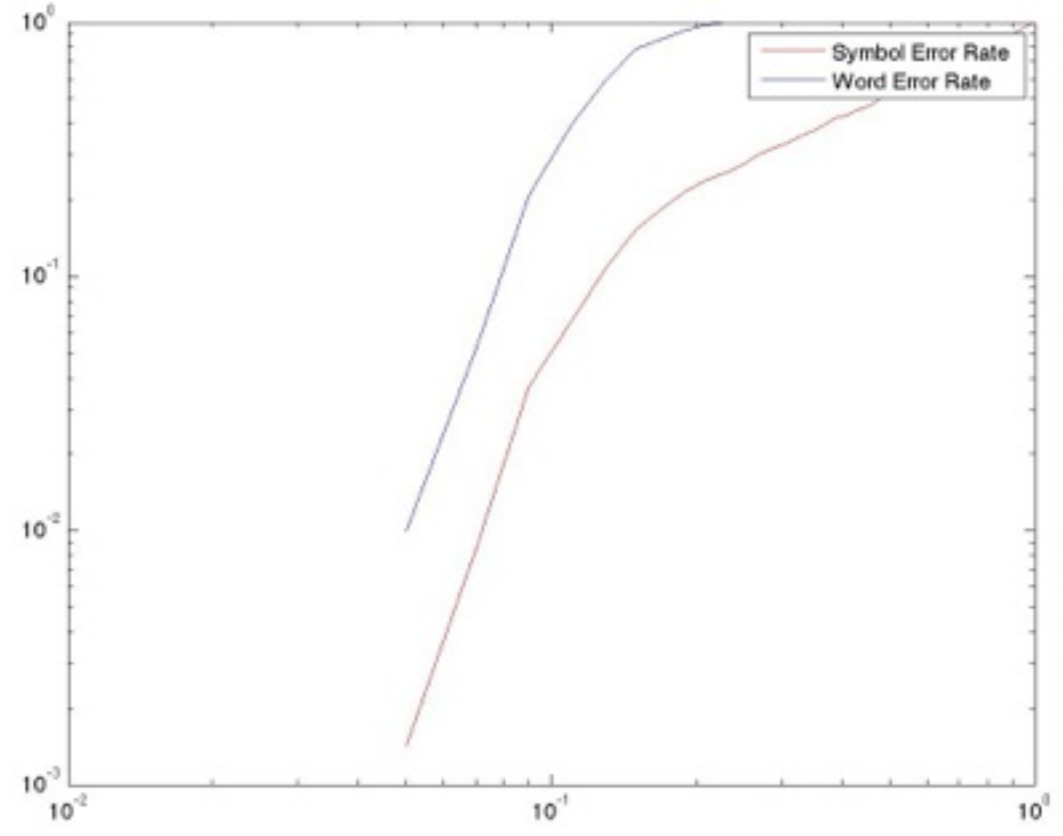
(22, 16)-code:



(46, 38)-code:


Symbol Error Rate
Word Error Rate
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-2}
10^{-1}
10^{0}

(72, 56)-code:


Symbol Error Rate
Word Error Rate
10^0
10^{-1}
10^{-2}
10^{-3}
10^{-2}
10^{-1}
10^0