Reed-Solomon Error Correcting Codes in C

Overview:

The purpose of this project was to implement the encoding and decoding of Reed-Solomon codes and perform simulations to measure the error correcting effectiveness of the codes under various random errors. I chose to implement this in C, using 32 bit integers to represent field elements and arrays of these integers to represent the coefficients of polynomials.

The guidelines of the project required that the code be a t-error-correcting code of length 31 over $GF(2^5)$, using the primitive polynomial $\alpha^5 + \alpha^3 + 1$, and having t as a variable input parameter.

Encoding:

The encoding algorithm implemented for this assignment is as follows:

$$c(x) = m(x)x^{n-k} - R_{g(x)}[m(x)x^{n-k}]$$

Where $c(x)$ is the encoded message, $m(x)$ is the message polynomial, and $R_{g(x)}[m(x)x^{n-k}]$ is the remainder of $m(x)x^{n-k}$ after division by $g(x)$, the generator polynomial.

This generator polynomial is obtained through the following formula:

$$g(x) = (x - \alpha)(x - \alpha^2)(x - \alpha^3) \dots (x - \alpha^{2t})$$

Where α is a primitive element in the field.

Decoding:

The Reed-Solomon decoding algorithm implemented for this assignment is based on the Euclidean algorithm and has four main steps: 1: Compute the syndrome. 2: Compute the error locator polynomial. 3: Find the roots of the error locator polynomial. 4: Calculate the error values. Once these steps are complete, the error values can be subtracted from the appropriate locations in the received code to correct it.

Since powers of α are roots of the generator polynomial, and thus roots of the encoded codeword, we can evaluate our received code at various powers of α to determine its "syndrome". This syndrome will be equal to 0 if the received code is a codeword, and will otherwise be used to determine the error locator polynomial. The elements of the syndrome can be computed as follows:

$$S_i = r(\alpha^i) = e(\alpha^i) = \Sigma e_k \alpha^{ik}$$

The error locator polynomial, $\Lambda(x)$, and error evaluator polynomial, $\Omega(x)$, can then be computed through use of the extended Euclidean algorithm. This can be set up as follows:

$$a(x) = x^{2t}, \qquad b(x) = S(x)$$

Where $S(x)$ is the polynomial form of the syndrome. To execute the algorithm, we then divide $a(x)$ by $b(x)$, storing the remainder and then setting $a(x)$ equal to $b(x)$. Then, setting $b(x)$ equal to our stored remainder, we can repeatedly divide until the degree of the remainder is less than t. When this point is reached, we have our results for $\Lambda(x)$ and $\Omega(x)$ in the form of the final remainder, and the extended component $t(x)$, which is computed along the way.

Next, we simply find the roots of the error locator polynomial. Since there were only 31 possible elements in the field, I implemented this through a brute-force search. The inverse of these roots

give the error locators X_i, the powers of which represent the indices of the errors in the received codeword.

Finally, using Forney's algorithm, we can compute the error values at the known indices:

$$e_{i_k} = \frac{\Omega(X_k^{-1})}{\Lambda'(X_k^{-1})}$$

If $\Lambda'(X_k^{-1}) = 0$, then we have a decoding failure.

As stated before, adding these values to the received code at the given indices returns the initial codeword. To retrieve our initial un-encoded message, since our encoder is systematic, we only have to take the last $k = 31 - 2t$ elements of the decoded code. We can then compare this decoded message to our initial message to determine if decoding was performed successfully.

Channel Model:

Our channel model used for simulation takes an integer probability P_s from 0 to 100, and for each element of the codeword that is transmitted, it adds a random error element a if a randomly generated number from 0 to 100 is less than P_s. This essentially means that each element of the codeword has a $\frac{P_s}{100}$ probability of having a random error element added to it.

Simulation Procedure:

Each simulation run takes a randomly created binary message vector of length $k = 31 - 2t$ and encodes it to a codeword. Next, this codeword is "sent through the channel" giving it random errors with some given probability. This new transmitted code is passed to the decoder which attempts to use the method described above to decode the message. If there is a decoding failure, or the decoding is incorrect, our simulation returns a decoding error.

Running many simulations for a given value of P_s and t allows us to compute the error rate:

$$ErrorRate(t, P_s) = \frac{\#\ of\ decoding\ errors}{\#\ of\ total\ messages\ sent}$$

In order to get a reasonable accuracy for the error rate, simulations are run for each value of P_s until 100 decoding errors occur.

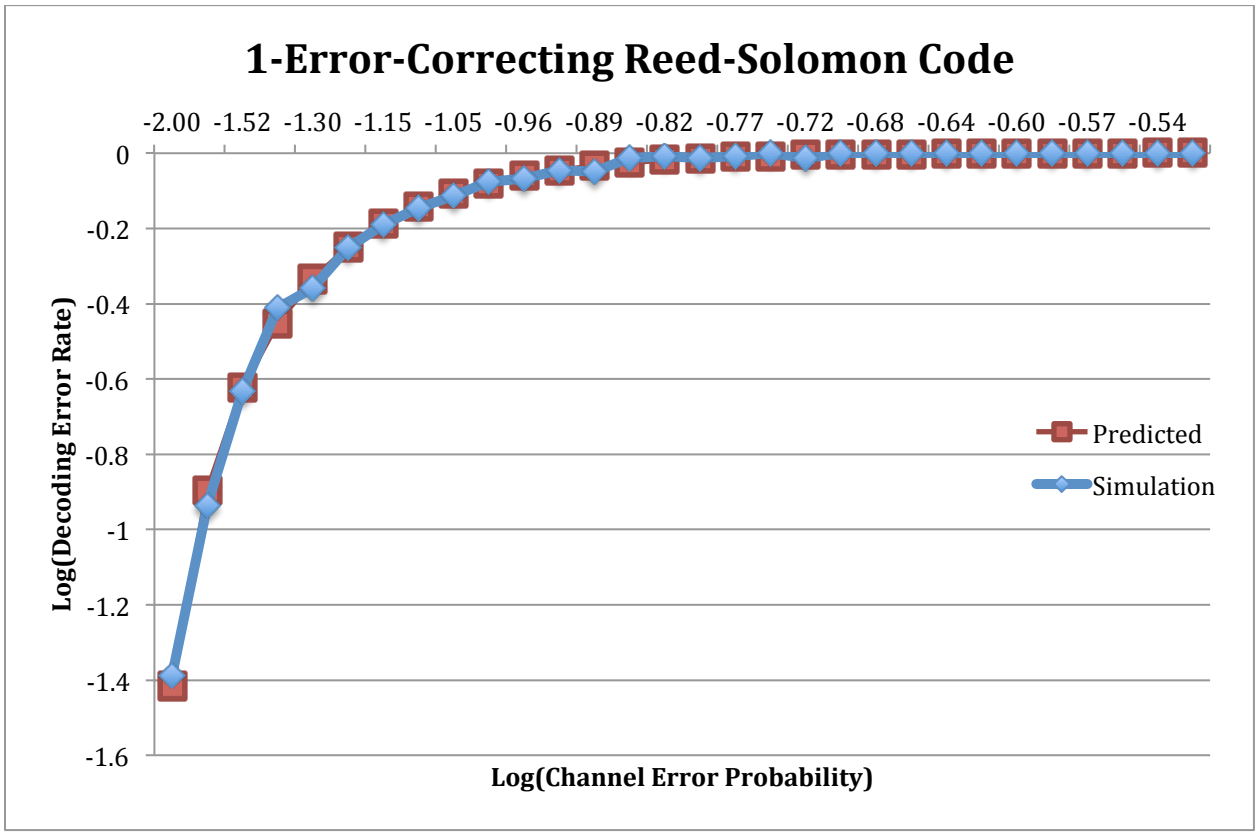
Theoretical Performance:

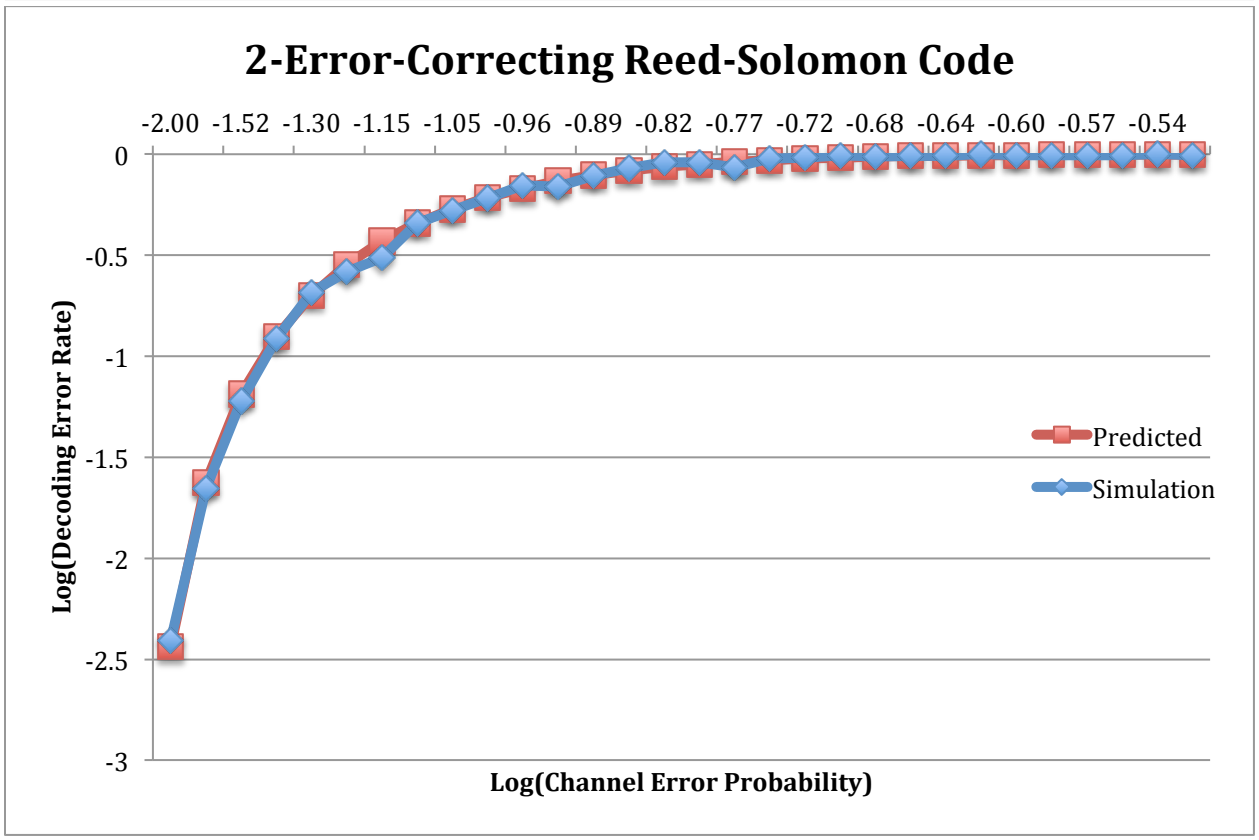
The theoretical error rate for a given t and P_s can be determined by treating each element passing through the channel as a Bernoulli trial where an error is a success, and then calculating the probability of having greater than t errors. This formula is given below:

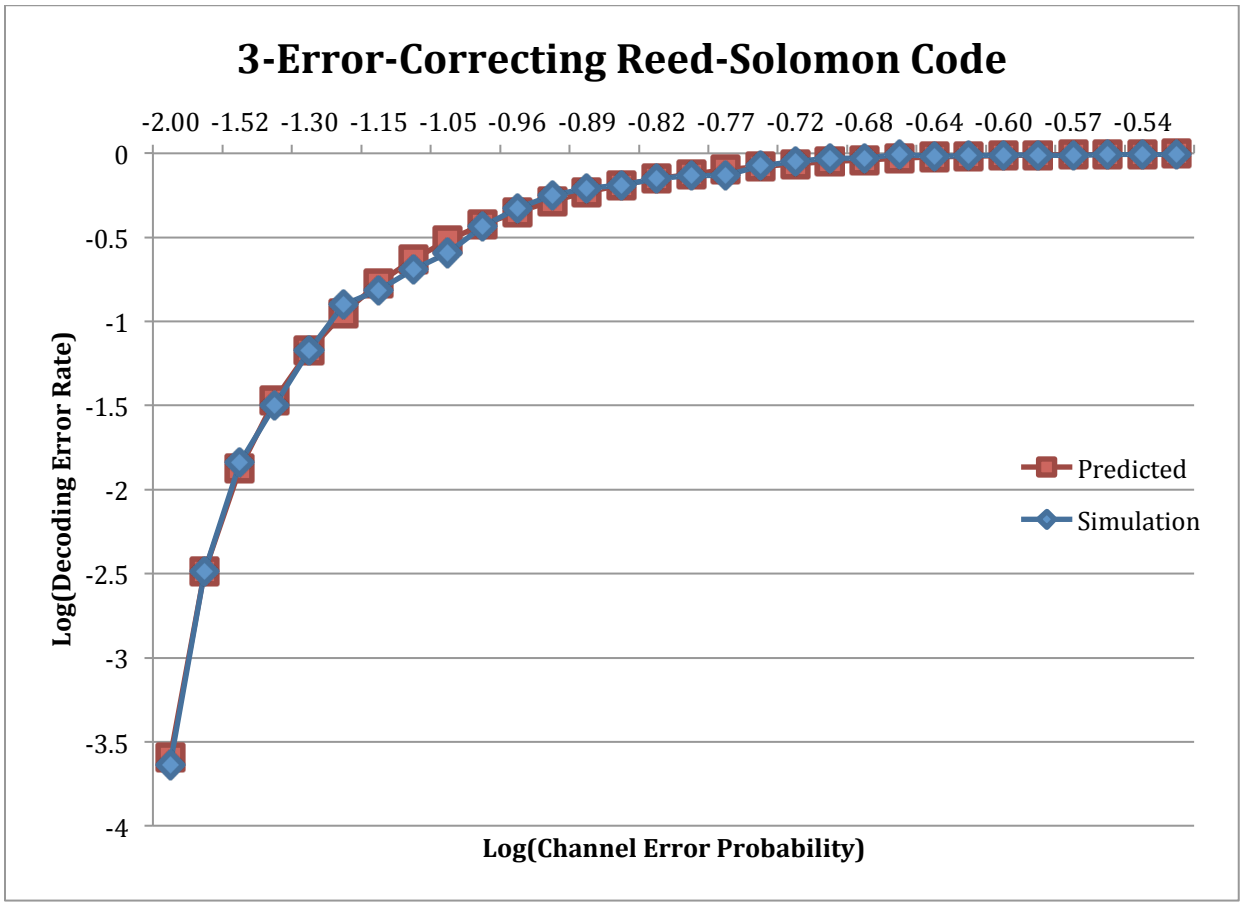
$$TheoreticalErrorRate(t, P_s) = \Sigma_{t+1}^{n}\ (n\ choose\ t) * P_s^t * (1 - P_s)^{n-t}$$

Where n is the length of the code.

Results:


1-Error-Correcting Reed-Solomon Code
-2.00 -1.52 -1.30 -1.15 -1.05 -0.96 -0.89 -0.82 -0.77 -0.72 -0.68 -0.64 -0.60 -0.57 -0.54
0
-0.2
-0.4
-0.6
-0.8
-1
-1.2
-1.4
-1.6
Log(Decoding Error Rate)
Log(Channel Error Probability)
Predicted
Simulation


2-Error-Correcting Reed-Solomon Code
-2.00 -1.52 -1.30 -1.15 -1.05 -0.96 -0.89 -0.82 -0.77 -0.72 -0.68 -0.64 -0.60 -0.57 -0.54
0
-0.5
-1
-1.5
-2
-2.5
-3
Log(Decoding Error Rate)
Log(Channel Error Probability)
Predicted
Simulation



Comparing the codes, we see that as expected, for larger values of t, we get lower Error Rates. Additionally, these lower error rates persist longer, allowing more codes to be decoded even as the channel error probability increases. However, for all three codes, we see that as we approach 30% error probability in the channel, there is essentially 100% decoding errors.

Additionally, our simulations appear to match the predicted error rates very closely, only differing at individual outlier data points. These differences could be fixed by running the simulations longer and waiting for more decoding errors to happen so as to give a more accurate decoding error rate.