Error Correction Coding: Weeks 9-10

Reed-Solomon Codes

Eric DeFelice
Department of Electrical Engineering
Rochester Institute of Technology
ebd9423@rit.edu

Abstract – **Reed-Solomon codes are block based cyclic error-correcting codes that are implemented by using many of the techniques and concepts that were introduced previously. These codes are used in a variety of applications, including storage device systems and communications systems. They are particularly good at correcting burst errors in a system. This is a desirable feature, as many other codes are adept at correcting random errors, so Reed-Solomon codes can be used in conjunction with these other codes to form a very reliable error-correction system.**

In this paper, an introduction of Reed-Solomon codes will be presented, and the properties of this code will be looked at. Since Reed-Solomon codes are linear block codes, we will not go over the specifics of the encoder and decoder, as we have already looked at linear block codes in depth previously. Matlab examples of Reed-Solomon codes will be looked at to show the codes performance. Puncturing and shortening will then be introduced, and simulations of these concepts will be looked at to see their effect on the communications system.

I. Introduction

Reed-Solomon codes are block-based error correction codes that are particularly well suited to correct bursts of errors. This is due to the fact that they correct errors in blocks, and if a block has a bit error, the whole block is treated as an erasure. Therefore, if there is one bit error in a block, or several, the code treats the correction in a similar fashion. This property will be looked at more closely later on.

The Reed-Solomon coding system works the same way as other linear block codecs. The encoder will take a "block" of data and add redundant bits, and the decoder processes each block and attempts to correct the errors within the block. The number and type of errors that can be corrected depends on the characteristics of the particular Reed-Solomon code.

II. Properties of Reed-Solomon Codes

Reed-Solomon codes are a subset of BCH codes. They can be specified in the form of *RS(n,k)* with *s-bit* symbols. This means that the encoder takes *k* data symbols, each of *s* bits, and adds parity symbols so that there are a total of *n* symbols in the codeword. Therefore, there are *n-k* parity symbols, and each symbol is *s* bits. The Reed-Solomon decoder can correct up to *t* symbols that contain errors, where *t* is defined using the equation $2t=n-k$.

For example, an *RS(255,223)* code with 8-bit symbols contains 255 bytes total, where 223 are data bytes and 32 are parity bytes. In this example code, *n* is 255, *k* is 223, and *s* is 8. This code can correct 16 symbols in a codeword, so in the worst case, there are 8 single bit errors in 16 symbols, so the code will correct only 8 bit errors. In the best case, there are 16 full symbols in error, or 8 bit errors per symbol. In this case the code would correct 16x8, or 128, bit errors, out of the 2040 total bits. This case shows why the Reed-Solomon code is good at correcting bursts of errors.

Reed-Solomon codes can be augmented in two different ways, by *puncturing* and by *shortening*. Puncturing is done by removing parity symbols at the encoder output. This reduces the amount of error correction that can be done by the code, but allows for a more flexible coding rate. For this process to work correctly, both the encoder and the decoder need to know the puncture pattern, and the decoder can add in zero symbols where the punctured parity symbols were. The decoder treats these symbols as erasures, effectively taking away the ability to correct an actual erasure that may have happened elsewhere in the codeword. Puncturing helps to improve the data rate slightly, because the encoder has added fewer redundant bits to the transmitted codeword.

Shortening is similar to puncturing, except that data symbols are removed in the shortening process. This also allows the coderate to be adjusted, but now the error correction capabilities of the code have not been affected. The data rate does drop when shortening is performed, because now there are more parity symbols per data symbols since some of the data symbols that were previously present are no longer there. The entire block size is reduced when shortening is performed, but the number of parity symbols remains the same. Puncturing and shortening are very helpful procedures, because they allow the system designer to change the code rate and data rate of the system without defining a whole new Reed-Solomon coding scheme. These procedures also allow the system to be adaptive by just removing parity symbols in a favorable environment or removing data symbols in an environment where the channel is harsh.

III. Matlab Simulations

To understand the effects of Reed-Solomon coding in a communications system, Matlab simulations were done to compare the bit error rates of coded systems vs uncoded systems. In the simulations, a 64-QAM coding scheme was used, so each symbol is 6-bits. A *RS(63,53)* code was used during the simulations, so there are 63 total symbols in the codeword, with 53 of them being data symbols and 10 parity symbols. Since there are 10 parity symbols, this code can correct up to 5 symbol errors, or 10 erasure symbols. The decoder will treat the least reliable symbols as error symbols and will just replace them with zeros, effectively making them act as erasures caused by the channel.

For the first simulation, the Reed-Solomon code performance was compared to an un-coded system where each un-coded block is 63 symbols and 64-QAM modulation was also used. The first step in the simulation is to define the system parameters. The modulator and demodulator are defined as a 64-QAM modem using gray coding, and the channel is defined as an AWGN channel. The signal to noise ratio (SNR) of the channel will be varied from 4 dB up to 15 dB during the simulations. This range was chosen because it gives a good picture of where the RS-coded system and the un-coded system performance begin to diverge.

The next step in the simulation is to define the Reed-Solomon encoder using 63 total symbols and 53 data symbols. The decoder is also defined, and the number of erasures that we will correct in the system is set to 6. More erasures could be corrected, but doing so would increase the latency at the decoder, so to make the system more representative of a real-world solution, only 6 erasures were chosen to be corrected.

The data is randomly generated and then encoded and sent through the AWGN channel. At the channel output the data then goes to a function that finds the least reliable symbols, in our case the 6 least reliable because that is what we chose for erasure correction. The function finds the amplitude of the real and imaginary components of the received data and divides by 8. Then the function chooses the 6 largest values, deeming those as the least reliable because they are most likely to have a large amount of noise or error. Once the erasure locations are known, the decoder can correct them, and the BER is calculated for the channel and for the coded system. The results from the un-coded system and the RS(63,53) coded system are seen in figure 1.

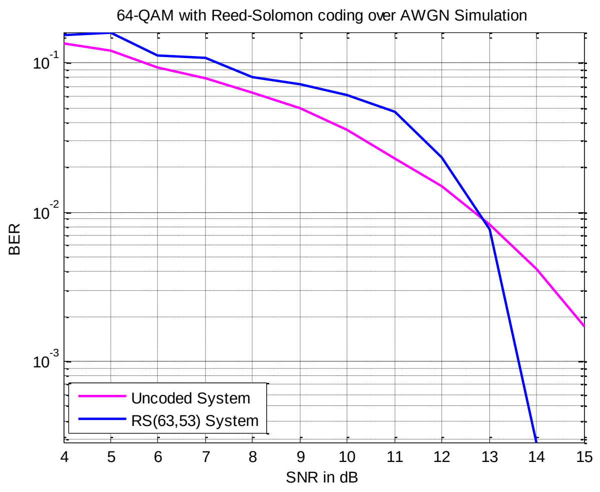

Figure 1: Un-coded system vs RS(63,53) system performance

This figure clearly shows that the Reed-Solomon code causes the BER at low SNRs to decrease slightly. This is because the code adds redundant bits. If the number of errors is similar between the systems, there are less data bits in the Reed-Solomon system, so the BER is higher. The Reed-Solomon coded system does have a lower BER when the SNR is higher. This is because there are less bit errors, which the code can correct successfully, allowing the code to be beneficial. The main reason why the code doesn't have better BER performance at low SNRs than the un-coded system is because Reed-Solomon coding is better suited for burst errors. If there is a single bit error in each symbol, the code can only correct six of them, leaving the rest in error. Perhaps a convolutional code would perform better for this system. We will look into this in the upcoming weeks.

The next simulation investigates the effects of puncturing on the code. In the simulation we will set the puncture pattern to the last two parity symbols, meaning that the last two parity symbols will be zeroed out, and the decoder will treat them as erasures. This should increase the data rate slightly, but have a worse bit error rate because we are reducing the amount of redundant symbols in the code. Figure 2 shows the results of that simulation.

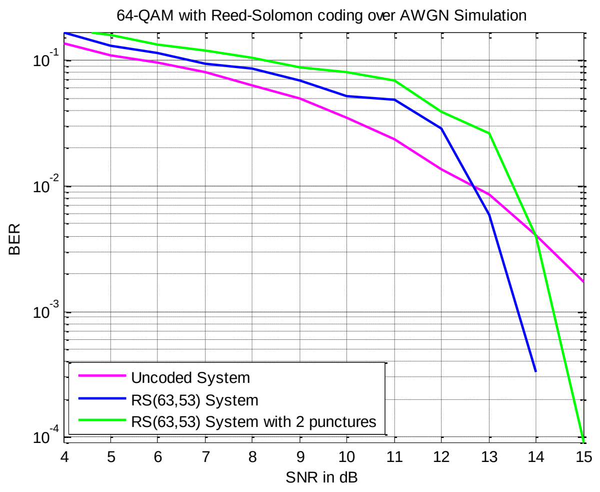

Figure 2: RS(63,53) system performance with two punctures

This figure shows that the BER does increase a little bit when two of the parity symbols are punctured. This is expected as we now have less error correction as well as fewer data bits per block when compared to the un-coded system.

The final simulation was done to look at the effects of shortening the RS code. For this simulation, the *RS(63,53)* code was shortened by 35 symbols, so to make it similar to an *RS(28,18)* code. In theory this code should provide good error correction, but would have worse data rate, as we have many more parity symbols per data symbols now. Also to make this work, the encoder must encode the symbols still using a generator in the GF(2^6) field, and the random symbols must be selected from that field as well. The results from this simulation are seen in figure 3.

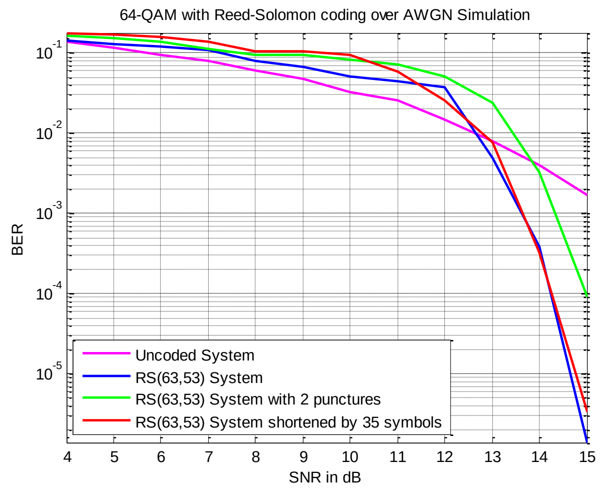

Figure 3: RS(63,53) system performance shortened by 35 symbols

This figure shows that the BER is very similar to the original RS coded system, except that the BER is worse at low SNRs. This is because our shortened code now has much fewer data symbols, so the data rate is already reduced, cause the BER to be much higher if any of those data symbols are in error.

IV. CONCLUSION

In this paper, Reed-Solomon codes were introduced and their operation was described. A brief overview of their properties was presented, and the concepts of puncturing and shortening were introduced. Finally, simulations were run to explore the effects of a Reed-Solomon code on a practical communications system. The effects of puncturing and shortening were also looked at through simulations. It was seen that the Reed-Solomon code does improve the bit error rate over an AWGN communication channel when the SNR is reasonably high, but the code does decrease the data rate at these SNRs. The Reed-Solomon code also decreases the BER slightly when compared to an un-coded transmission at low SNRs, because the construction of the code is better suited for bursts of errors, as opposed to random bit errors. These random bit errors create worse conditions in the RS coded systems because there are fewer data symbols to begin with, so any symbols in error effect the BER more so than the un-coded symbol errors.

The next couple of weeks will be spent by looking into convolutional and turbo codes, which are not block codes. These codes have much less latency than block codes because they do not need to wait for a whole block of data before transmission. The investigation into these two specific coding schemes will again be more Matlab based. The theory behind these codes will be introduced, but the math behind convolutional codes is less algebraic than block codes, and I am already somewhat familiar with it, therefore I will only review the background theory.

REFERENCES

[1] San Ling, Chaoping Xing, *Coding Theory: A First Course*. Cambridge University Press, 2004

[2] Andre Neubauer, Jurgen Freudenberger, Volker Kuhn, *Coding Theory: Algorithms, Architectures, and Applications*. John Wiley & Sons Ltd., 2007

[3] Scott A. Vanstone, Paul C. van Oorschot, *An Introduction to Error Correcting Codes with Applications*. Kluwer Academic Publishers., 1989

[4] http://www.mathworks.com/help/comm/examples/reed-solomon-coding-part-i-erasures.html

APPENDIX

```
%% Error correction coding : Weeks 9-10
%% Reed-Solomon coding
close all;
clear all;
clc;

%% Initialize variables
% Set the simulation stop criteria by defining the target number of errors
% and maximum number of transmissions.
targetErrors = 500;
maxNumTransmissions = 5e6;
% Simulate the encoder over a range of SNRs
EbNoUncoded = 4:15;

% Define variables for coded and uncoded BER
codedBER = zeros(1,length(EbNoUncoded));
uncodedBER = zeros(1,length(EbNoUncoded));

% Perform the simulations over the range of SNRs
for i=1:length(EbNoUncoded)
%64-QAM Modulator variables
M = 64; % Modulation order
% Setup modem to 6-bit, gray coded, with integer input/output
hMod   = comm.RectangularQAMModulator(M, 'SymbolMapping', 'Gray',...
         'BitInput', false);
hDemod = comm.RectangularQAMDemodulator(M, 'SymbolMapping', 'Gray',...
         'BitOutput', false);

% AWGN Channel variables
% Channel model
hChan = comm.AWGNChannel('NoiseMethod', 'Signal to noise ratio (Eb/No)');
% There is no up/down-sampling, and set signal power to 42 Watts.
hChan.SamplesPerSymbol = 1;
hChan.SignalPower = 42;
% Set the bits per symbol to log2(M) or 6 bps for 64-QAM
hChan.BitsPerSymbol = log2(M);

% Error rate measurement variables
% Objects for channel BER and coded BER
hChanBERCalc = comm.ErrorRate;
hCodedBERCalc = comm.ErrorRate;
% Integer to bit converters for 64-QAM
hIntToBit1 = comm.IntegerToBit('BitsPerInteger', log2(M));
hIntToBit2 = comm.IntegerToBit('BitsPerInteger', log2(M));
% Cumulative sum object to count the number of error symbols corrected by
% Reed-Solomon decoder
hCumSum = dsp.CumulativeSum;

% Reed-Solomon encoder
% Use a code with 53 message symbols and 10 parity symbols, so the total codeword
% is 63 symbols
N = 63;
K = 53;
hEnc = comm.RSEncoder(N,K, 'BitInput', false);

% Reed-Solomon Decoder
hDec = comm.RSDecoder(N,K, 'BitInput', false, 'ErasuresInputPort', true);
hDec.NumCorrectedErrorsOutputPort = true;
% We will correct symbols by picking out the least reliable ones and
% marking them as erasures.  This variable sets how many erasures we will
% declare.
numErasures = 6;
```

```
% Set the coded SNR to the uncoded SNR minus the noise from the code bits
% This is because there are extra redundant bits that only increase the
% noise power and not the signal power, as they don't contain data
EbNoCoded = EbNoUncoded(i) + 10*log10(K/N);
hChan.EbNo = EbNoCoded;

% Perform coding and decoding until the target number of errors has occurred
% or until the maximum number of transmissions has occurred
chanErrorStats = zeros(3,1);
codedErrorStats = zeros(3,1);
correctedErrors = 0;
while (codedErrorStats(2) < targetErrors) && ...
  (codedErrorStats(3) < maxNumTransmissions)
  % Data symbols - transmit 1 message word at a time. Each message word has
  % K symbols in the [0 N] range.
  data = randi([0 N],K,1);
  % Encode the message word. The encoded word encData is N symbols long.
  encData = step(hEnc, data);
  % Modulate encoded data.
  modData = step(hMod, encData);
  % Send the modulated data through the AWGN channel model.
  chanOutput = step(hChan, modData);
  % Demodulate channel output.
  demodData = step(hDemod, chanOutput);
  % Find the 6 least reliable symbols and generate an erasures vector using
  % the getErasuresRSCodingDemo function. A one in the ith element of the
  % vector erases the ith symbol in the codeword.
  % Zeros in the vector indicate no erasures.
  erasuresVec = getErasuresRSCodingDemo(chanOutput,numErasures);
  % Decode data using the decoder, the demodulated data and the erasures
  % vector
  [estData errs] = step(hDec, demodData, erasuresVec);
  % Accumulate the number of corrected errors using the cumulative sum object.
  if errs >= 0
    correctedErrors = step(hCumSum, errs);
  end
  % Convert integers to bits and compute the channel BER.
  chanErrorStats(:,1) = ...
    step(hChanBERCalc,step(hIntToBit1,encData),step(hIntToBit1,demodData));
  % Convert integers to bits and compute the coded BER.
  codedErrorStats(:,1) = ...
    step(hCodedBERCalc,step(hIntToBit2,data),step(hIntToBit2,estData));
end
% Set the BER for the current SNR
codedBER(i) = codedErrorStats(1);
uncodedBER(i) = chanErrorStats(1);
end

% Plot the BER vs SNR
semilogy(EbNoUncoded,uncodedBER,'m-','linewidth',2.0);
hold on
semilogy(EbNoUncoded,codedBER,'b-','linewidth',2.0);
title('64-QAM with Reed-Solomon coding over AWGN Simulation');xlabel('SNR in dB');ylabel('BER');
legend('Uncoded System','RS(63,53) System');
%axis tight
%grid


%% Perform simulations with puncturing
% Define variables for coded and uncoded BER
codedBER_w_punc = zeros(1,length(EbNoUncoded));
% Perform the simulations over the range of SNRs
for i=1:length(EbNoUncoded)
%64-QAM Modulator variables
```

```
M = 64; % Modulation order
% Setup modem to 6-bit, gray coded, with integer input/output
hMod   = comm.RectangularQAMModulator(M, 'SymbolMapping', 'Gray',...
         'BitInput', false);
hDemod = comm.RectangularQAMDemodulator(M, 'SymbolMapping', 'Gray',...
         'BitOutput', false);

% AWGN Channel variables
% Channel model
hChan = comm.AWGNChannel('NoiseMethod', 'Signal to noise ratio (Eb/No)');
% There is no up/down-sampling, and set signal power to 42 Watts.
hChan.SamplesPerSymbol = 1;
hChan.SignalPower = 42;
% Set the bits per symbol to log2(M) or 6 bps for 64-QAM
hChan.BitsPerSymbol = log2(M);

% Error rate measurement variables
% Objects for channel BER and coded BER
hChanBERCalc = comm.ErrorRate;
hCodedBERCalc = comm.ErrorRate;
% Integer to bit converters for 64-QAM
hIntToBit1 = comm.IntegerToBit('BitsPerInteger', log2(M));
hIntToBit2 = comm.IntegerToBit('BitsPerInteger', log2(M));
% Cumulative sum object to count the number of error symbols corrected by
% Reed-Solomon decoder
hCumSum = dsp.CumulativeSum;

% Reed-Solomon encoder
% Use a code with 53 message bits and 10 parity bits, so the total codeword
% is 63 bits
N = 63;
K = 53;
hEnc = comm.RSEncoder(N,K, 'BitInput', false);

% Reed-Solomon Decoder
hDec = comm.RSDecoder(N,K, 'BitInput', false, 'ErasuresInputPort', true);
hDec.NumCorrectedErrorsOutputPort = true;
% We will correct symbols by picking out the least reliable ones and
% marking them as erasures.  This variable sets how many erasures we will
% declare.
numErasures = 6;

% Specify the puncture pattern vector of length N-K. Set the last two
% elements of the puncture pattern vector to zero to puncture the last two
% parity bits of each codeword.
numPuncs = 2;
hEnc.PuncturePatternSource = 'Property';
hEnc.PuncturePattern = [ones(N-K-numPuncs,1); zeros(numPuncs,1)];
hDec.PuncturePatternSource = 'Property';
hDec.PuncturePattern = hEnc.PuncturePattern;

% Set the coded SNR to the uncoded SNR minus the noise from the code bits
% This is because there are extra redundant bits that only increase the
% noise power and not the signal power, as they don't contain data
EbNoCoded = EbNoUncoded(i) + 10*log10(K/N);
hChan.EbNo = EbNoCoded;

% Perform coding and decoding until the target number of errors has occurred
% or until the maximum number of transmissions has occurred
chanErrorStats = zeros(3,1);
codedErrorStats = zeros(3,1);
correctedErrors = 0;
while (codedErrorStats(2) < targetErrors) && ...
  (codedErrorStats(3) < maxNumTransmissions)
  % Data symbols - transmit 1 message word at a time. Each message word has
  % K symbols in the [0 N] range.
  data = randi([0 N],K,1);
  % Encode the message word. The encoded word encData is N symbols long.
  encData = step(hEnc, data);
  % Modulate encoded data.
  modData = step(hMod, encData);
  % Send the modulated data through the AWGN channel model.
  chanOutput = step(hChan, modData);
  % Demodulate channel output.
  demodData = step(hDemod, chanOutput);
  % Find the 6 least reliable symbols and generate an erasures vector using
  % the getErasuresRSCodingDemo function. A one in the ith element of the
  % vector erases the ith symbol in the codeword.
  % Zeros in the vector indicate no erasures.
  erasuresVec = getErasuresRSCodingDemo(chanOutput,numErasures);
  % Decode data using the decoder, the demodulated data and the erasures
  % vector
  [estData errs] = step(hDec, demodData, erasuresVec);
  % Accumulate the number of corrected errors using the cumulative sum object.
  if errs >= 0
    correctedErrors = step(hCumSum, errs);
  end
  % Convert integers to bits and compute the channel BER.
  chanErrorStats(:,1) = ...
    step(hChanBERCalc,step(hIntToBit1,encData),step(hIntToBit1,demodData));
  % Convert integers to bits and compute the coded BER.
  codedErrorStats(:,1) = ...
    step(hCodedBERCalc,step(hIntToBit2,data),step(hIntToBit2,estData));
end
% Set the BER for the current SNR for the puncturing case
codedBER_w_punc(i) = codedErrorStats(1);
end

% Plot the BER vs SNR
hold on;
semilogy(EbNoUncoded,codedBER_w_punc,'g-','linewidth',2.0);
legend('Uncoded System','RS(63,53) System','RS(63,53) System with 2 punctures');
axis tight
grid

%% Perform simulations with shortening
% Define variables for coded and uncoded BER
codedBER_w_short = zeros(1,length(EbNoUncoded));
% Perform the simulations over the range of SNRs
for i=1:length(EbNoUncoded)
%64-QAM Modulator variables
M = 64; % Modulation order
% Setup modem to 6-bit, gray coded, with integer input/output
hMod   = comm.RectangularQAMModulator(M, 'SymbolMapping', 'Gray',...
         'BitInput', false);
hDemod = comm.RectangularQAMDemodulator(M, 'SymbolMapping', 'Gray',...
         'BitOutput', false);

% AWGN Channel variables
% Channel model
hChan = comm.AWGNChannel('NoiseMethod', 'Signal to noise ratio (Eb/No)');
% There is no up/down-sampling, and set signal power to 42 Watts.
hChan.SamplesPerSymbol = 1;
hChan.SignalPower = 42;
% Set the bits per symbol to log2(M) or 6 bps for 64-QAM
hChan.BitsPerSymbol = log2(M);

% Error rate measurement variables
% Objects for channel BER and coded BER
hChanBERCalc = comm.ErrorRate;
```

```
hCodedBERCalc = comm.ErrorRate;
% Integer to bit converters for 64-QAM
hIntToBit1 = comm.IntegerToBit('BitsPerInteger', log2(M));
hIntToBit2 = comm.IntegerToBit('BitsPerInteger', log2(M));
% Cumulative sum object to count the number of error symbols corrected by
% Reed-Solomon decoder
hCumSum = dsp.CumulativeSum;

% Reed-Solomon encoder
% Use a code with 53 message bits and 10 parity bits, so the total codeword
% is 63 bits
N = 63;
K = 53;
hEnc = comm.RSEncoder(N,K, 'BitInput', false);

% Reed-Solomon Decoder
hDec = comm.RSDecoder(N,K, 'BitInput', false, 'ErasuresInputPort', true);
hDec.NumCorrectedErrorsOutputPort = true;
% We will correct symbols by picking out the least reliable ones and
% marking them as erasures.  This variable sets how many erasures we will
% declare.
numErasures = 6;

% Shorten the code by 35 symbols
% This would give us a RS(28,18) code, so much more parity symbols per data
% symbols
shortenLength = 35;

% Set the shortened codeword length and message length values
hEnc.CodewordLength = N - shortenLength;
hEnc.MessageLength = K - shortenLength;

hDec.CodewordLength = N - shortenLength;
hDec.MessageLength = K - shortenLength;

% We need to updated the primitive polynomial at the encoder so that it
% knows that the original code is an RS(63,53) code.  We set the primitive
% polynomial to be in the GF(2^6) field to show this.
hEnc.PrimitivePolynomialSource = 'Property';
hEnc.PrimitivePolynomial = de2bi(primpoly(6, 'nodisplay'), 'left-msb');

hDec.PrimitivePolynomialSource = 'Property';
hDec.PrimitivePolynomial = de2bi(primpoly(6, 'nodisplay'), 'left-msb');

% Set the coded SNR to the uncoded SNR minus the noise from the code bits
% This is because there are extra redundant bits that only increase the
% noise power and not the signal power, as they don't contain data
EbNoCoded = EbNoUncoded(i) + 10*log10((K-shortenLength)/(N-shortenLength));
hChan.EbNo = EbNoCoded;

% Perform coding and decoding until the target number of errors has occurred
% or until the maximum number of transmissions has occurred
chanErrorStats = zeros(3,1);
codedErrorStats = zeros(3,1);
correctedErrors = 0;
while (codedErrorStats(2) < targetErrors) && ...
  (codedErrorStats(3) < maxNumTransmissions)
  % Data symbols - transmit 1 message word at a time. Each message word has
  % K symbols in the [0 N] range. We now need to specify N to be 2^6-1
  % because we have changed the code length.
  data = randi([0 2^6-1],K-shortenLength,1);
  % Encode the message word. The encoded word encData is N symbols long.
  encData = step(hEnc, data);
  % Modulate encoded data.
  modData = step(hMod, encData);
```

```
    % Send the modulated data through the AWGN channel model.
    chanOutput = step(hChan, modData);
    % Demodulate channel output.
    demodData = step(hDemod, chanOutput);
    % Find the 6 least reliable symbols and generate an erasures vector using
    % the getErasuresRSCodingDemo function. A one in the ith element of the
    % vector erases the ith symbol in the codeword.
    % Zeros in the vector indicate no erasures.
    erasuresVec = getErasuresRSCodingDemo(chanOutput,numErasures);
    % Decode data using the decoder, the demodulated data and the erasures
    % vector
    [estData errs] = step(hDec, demodData, erasuresVec);
    % Accumulate the number of corrected errors using the cumulative sum object.
    if errs >= 0
      correctedErrors = step(hCumSum, errs);
    end
    % Convert integers to bits and compute the channel BER.
    chanErrorStats(:,1) = ...
      step(hChanBERCalc,step(hIntToBit1,encData),step(hIntToBit1,demodData));
    % Convert integers to bits and compute the coded BER.
    codedErrorStats(:,1) = ...
      step(hCodedBERCalc,step(hIntToBit2,data),step(hIntToBit2,estData));
  end
  % Set the BER for the current SNR for the puncturing case
  codedBER_w_short(i) = codedErrorStats(1);
  end

  % Plot the BER vs SNR
  hold on;
  semilogy(EbNoUncoded,codedBER_w_short,'r-','linewidth',2.0);
  legend('Uncoded System','RS(63,53) System','RS(63,53) System with 2 punctures', ...
         'RS(63,53) System shortened by 35 symbols');
  axis tight
  grid



  function erasuresVector = getErasuresRSCodingDemo(inputData, numErasures)
  %getErasuresRSCodingDemo Find erasures vector for commRSCodingErasures.m,
  %commRSCodingPunctures.m, commRSCodingShortening.m examples.
  %   erasuresVector = getErasuresRSCodingDemo(inputData, numErasures) finds a
  %   vector of indices, that point to the numErasures worst performance
  %   symbols in inputData. Then it creates an output vector, erasuresVector,
  %   of length length(inputData) with the elements corresponding to the worst
  %   case symbol indices set to one and all other elements set to zero.

  %   Copyright 2010-2011 The MathWorks, Inc.
  %   $Revision: 1.1.6.2 $  $Date: 2011/12/11 07:38:35 $

  collapsedData = abs([real(inputData) imag(inputData)]);

  collapsedData = abs(collapsedData - 4);
  collapsedData = abs(collapsedData - 2);
  collapsedData = abs(collapsedData - 1);
  collapsedData = max(collapsedData,[],2);
  [~, idx] = sort(collapsedData);
  erasuresIdx = idx(end-numErasures+1: end);

  erasuresVector = zeros(length(inputData),1);
  erasuresVector(erasuresIdx) = 1;
```