Table of Contents


```
%
% Communication Theory Project
% Group: Shifra, Jonny, & Guy
%
% Part 2
```

Section 1: Reference Taps

```
% As mentioned in part 1, we began part 2 without a working
 implementation
% of reference taps. Although we wanted to move on to more important
 parts
% of the project (i.e. implement error correcting codes), this small
 part
% of the equalizer bugged us (pun intended) and so, as will be seen in
 the
% code to follow, we got the reference taps part of the equalizer
 working.

% Attached below, for the sake of completeness, is our code from part
 1
% updated with reference taps (there isn't so much to see here, the
 only
% difference is in the lines 59, 80-82, and 143).
%
% To shorten our code, we only considered 16-ary QAM modulation with
 an
% rsl-dfe equalizer, printing out the BER rate at 12 SNR.


% Parameters:
% We made the number of iterations large so that we could see the ber
 at
% 12 SNR and confirm that it meets the specifications.
numIter = 10;
n_sym = 1000;    % The number of symbols per packet
SNR_Val = 12;

% We are showing 16-ary QAM modulation
M = 16;
```

```
% Channel to use
chan = [1 .2 .4]; % Somewhat invertible channel impulse response,
 Moderate ISI

% Number of training symbols (max=len(msg)=1000)
num_train = 350;

% equalizer hyperparameters
n_weights = 6;
n_weights_feedback = 7;

numRefTap = 3;

% Building the equalizer:
% adaptive filter algorithm
adaptive_algo = rls(1, 0.1);

% equalizer object
eqobj = dfe(n_weights, n_weights_feedback, adaptive_algo); % like IIR
eqobj.ResetBeforeFiltering = 0;
eqobj.RefTap = numRefTap;
delay = (numRefTap-1)/eqobj.nSampPerSym;

% Create a vector to store the BER computed during each iteration
berVec_no_eq = zeros(numIter, 1);
berVec_eq = zeros(numIter, 1);

% Running the simulation:
for i = 1:numIter

    % message to transmit
    bits = randi(2,[(n_sym+delay)*log2(M), 1])-1;
    msg = bits2msg(bits, M);

    % modulation
    tx = qammod(msg, M);  % QAM modulation

    % Sequence of Training Symbols
    train_seq = tx(1:num_train);

    % transmit (convolve) through channel
    txChan_rs = filter(chan,1,tx);  % Apply the channel.

    % Adding AWGN. First need to convert from EbNo to SNR.
    noise_addition = 10*log10(log2(M));
    tx_noisy = awgn(txChan_rs, noise_addition+SNR_Val, 'measured');

    rx_demod_signal = equalize(eqobj,tx_noisy,train_seq);

    % de-modulation
    rx_eq = qamdemod(rx_demod_signal, M);  % QAM
    rx_no_eq = qamdemod(tx_noisy, M);
```

```
    rx_msg_eq = msg2bits(rx_eq, M);
    rx_msg_no_eq = msg2bits(rx_no_eq, M);

    % Compute and store the BER for this iteration
    % We're interested in the BER, which is the 2nd output of BITERR
    [~, berVec_eq(i,1)] = biterr(bits(1+num_train:end-delay*log2(M)),
 rx_msg_eq(1+num_train+delay*log2(M):end));
    [~, berVec_no_eq(i,1)] = biterr(bits, rx_msg_no_eq);

end      % End numIter iteration

% Compute and plot the mean equalizer BER
ber_eq = mean(berVec_eq,1);
ber_no_eq = mean(berVec_no_eq,1);
berTheory = berawgn(SNR_Val, 'qam', M); % QAM

Types = {'Equalized', 'Not Equalized', 'Theoretical'}';
BER_Rate = [ber_eq, ber_no_eq, berTheory]';
Section_1_Table = table(Types, BER_Rate);
disp(Section_1_Table)

% Note that unlike in part 1 where we got BPSK down to 10^-4, we
 almost
% nearly got 16-ary QAM down to the same requirement.
```

Section 2: BCH and BPSK

```
% With the reference taps implemented we moved on to error correcting
% codes. At this point our group decided to work in parallel, with
 each
% member looking at different error correcting codes.

% The first code we tried was BCH with BPSK modulation. Although we
 didn't
% got this encoding working, we decided to move on to convolutional
% encoding and so we never finalized it into a working implementation.
%
% As such the code below neither works nor is not cleaned and
 efficient
% but instead extensively documented. This was done by our group so
 that
% we all understood the what, how and why behind every line. The
 following
% sections showcase a more complete implementation of other encodings.


% This try block is just a way to keep the syntax looking nice even
 though
% the code inside this block gives an error.
try
    % Parameters:
    numIter = 10;  % The number of iterations of the simulation
    n_sym = 10000;    % The number of symbols per packet
```

```
SNR_Vec = 12;
lenSNR = length(SNR_Vec);

% The M-ary number. 2 corresponds to binary modulation.
M = 2;

% Modulation
%  - 1 = PAM
%  - 2 = QAM
%  - 3 = PSK
modulation = 2;

chan = [1 .2 .4]; % Somewhat invertible channel impulse response

% number of training symbols
num_train = 175;

% Adaptive Algorithm
%  - 0 = varlms
%  - 1 = lms
%  - 2 = rls
adaptive_algo = 2;

% Equalizer
%  - 0 = lineareq
%  - 1 = dfe
equalize_val = 0;

% equalizer hyperparameters
NWeights = 6;
NWEIGHTS_Feedback = 5;
numRefTap = 2;
stepsize = 0.005;
forgetfactor = 1; % between 0 and 1

%%%%%%%%%%%%%%%%%%%%%%%%% CREATING EQUALIZER %%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%

% adaptive filter algorithm
if isequal(adaptive_algo, 0)
    AdapAlgo = varlms(stepsize,0.01,0,0.01);
elseif isequal(adaptive_algo, 1)
    AdapAlgo = lms(stepsize);
else
    AdapAlgo = rls(forgetfactor);
end

% Equalizer Object
if isequal(equalize_val, 0)
    eqobj = lineareq(NWeights,AdapAlgo); %comparable to an FIR
else
    eqobj = dfe(NWeights, NWEIGHTS_Feedback,
 AdapAlgo); %comparable to an IIR
end
```

```
   eqobj.ResetBeforeFiltering = 0;
   eqobj.RefTap = numRefTap;
   delay = (numRefTap-1)/eqobj.nSampPerSym;
   n_sym = n_sym - delay;

   %%%%%%%%%%%%%%%%%%% CREATING ERROR CONTROL CODING SCHEME %%%%%%%%%
%%%%%%%%%
   % The information to be encoded consists of message symbols and
 the code
   % that is produced consists of codewords. Each block of K message
 symbols
   % is encoded into a codeword that consists of N message symbols. K
 is
   % called the message length, N is called the codeword length, and
 the code
   % is called an [N,K] code.

   % You can structure messages and codewords as binary vector
 signals, where
   % each vector represents a message word or a codeword. At a given
 time, the
   % encoder receives an entire message word, encodes it, and outputs
 the
   % entire codeword. The message and code signals operate over the
 same
   % sample time.

   % BCH: For these codes, the codeword length N must have the form
 2M-1,
   % where M is an integer from 3 to 16 (default is 15). The message
 length K
   % is restricted to particular values that depend on N. To see
 which values
   % of K are valid for a given N, see the comm.BCHEncoder System
 object™
   % reference page. No known analytic formula describes the
 relationship
   % among the codeword length, message length, and error-correction
   % capability for BCH codes. Message length default is 5.

   X = 4; % integer from 3 to 16;
   % NOTE: The documentation uses the variable M in place of x, but
 this is
   % confusing because this value is different than the modulation
 value M.

   % CODEWORD LENGTH:
   n = 2^X-1; % default is 15, max allowed is 65,535
   % bchnumerr(n) will return all possible K/message values for a
 particular
   % N and the number of correctable errors in a three column matrix.

   %MESSAGE LENGTH:
   k = 5; % default is 5,
```

```
    % Example: 5 specifies a Galois array with five elements, 2^m
 (second
    % value in gf).

    paritypos='beginning';
    % paritypos = 'end' or 'beginning' specify whether parity bits
 appear at
    % the end or beginning of the signal.

    % The message must be fed into the encoder using the Galois field
 array of
    % symbols over GF(2). Each K-element row of msg represents a
 message word,
    % where the leftmost symbol is the most significant symbol.

    %%%%%%%%%%% THIS IS LATER ON: msgTx = gf(x,1) will create a Galois
 array
    % in GF(2^m).
    % The elements of x must be integers between 0 and 2^m-1, if only
 using bits
    % (x={0,1}), than m=1/second argument and we are in GF(2).

    %%%%%%%%%%% THIS IS LATER ON: enc = bchenc(msgTx,n,k,paritypos)
 occurs
    % before adding noise and after converting from message to bits.
    % THIS IS HOW NOISE ADDITION IS DONE IN THE DOCUMENTATION
    % EX:
    % Corrupt up to t bits in each codeword where t = bchnumerr(n,k)
    % noisycode = enc + randerr(numbits,n,1:t). This is for full
    % reconstruction with no errors and so doesnt concern us, because
 we are
    % anayways adding AWGN.

    %%%%%%%%%%% THIS IS LATER ON:msgRx = bchdec(noisycode,n,k) will
 decode
    % the noisy message

    % Order of Operations
    % source:https://www.researchgate.net/figure/System-
model_fig2_303940773
    % Data Source --> Conversion to Bits --> BCH Encoder -->
 Modulation -->
    % Filter/Channel --> Recieve Filter/AWGN --> Demodulation --> BCH
 Decoder
    % --> Convert back to message

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%% RUNNING SIMULATION %%%%%%%%%%%%%%%%%%%
%%%%%%

    % Create a vector to store the BER computed during each iteration
    berVec_bch = zeros(numIter, lenSNR);
    berVec_no_bch = zeros(numIter, lenSNR);
```

```
    for i = 1:numIter

        % message to transmit
        bits = randi(2,[(n_sym+delay)*log2(M), 1])-1;

        % BCH encoding:
        % Must first reshape msg so that each row has k elements
        bits_reshape = reshape(bits,k,[]).';

        bits_gf = gf(bits_reshape,1);
        bits_enc = bchenc(bits_gf,n,k,paritypos);

        msg_bch = bits2msg(bits_enc, M);
        msg_no_bch = bits2msg(bits, M);

        % Not totally sure if encoding should occur inside or outside
of
        % this loop:
        for j = 1:lenSNR % one iteration of the simulation at each SNR
Value

            % Now must unwrap matrix into vector to input into
modulation fns
            % msg_enc = msg_enc_matrix.';
            % msg_enc = msg_enc(:);
            % ISSUE IS FEEDING Galois Field Array into modulation
schemes

            % modulation
            if isequal(modulation, 1)
                tx_bch = pammod(msg_bch, M);  % PAM modulation
            elseif isequal(modulation, 2)
                tx_bch = qammod(msg_bch, M);  % QAM modulation
                tx_no_bch = qammod(msg_no_bch, M); %QAM nonencoded mod
            else
                tx_bch = pskmod(msg_bch, M);  % PSK modulation
            end

            % Sequence of Training Symbols
            trainseq_bch = tx_bch(1:num_train);
            trainseq_no_bch = tx_no_bch(1:num_train); %no encoding

            % transmit (convolve) through channel
            if isequal(chan,1)
                txChan_bch = tx_bch;
            elseif isa(chan,'channel.rayleigh')
                reset(chan) % Draw a different channel each iteration
                txChan_bch = filter(chan,tx_bch);
            else
                txChan_bch = filter(chan,1,tx_bch);  % Apply the
channel.
                txChan_no_bch = filter(chan,1,tx_no_bch);
            end
```

```
            % Convert from EbNo to SNR.
            noise_addition = round(10*log10(2*log2(M)));
            txNoisy_bch = awgn(txChan_bch, 3+SNR_Vec(j), 'measured');
            txNoisy_no_bch = awgn(txChan_no_bch,
 3+SNR_Vec(j), 'measured');

            % equalize
            yd_bch = equalize(eqobj, txNoisy_bch, trainseq_bch);
            yd_no_bch = equalize(eqobj, txNoisy_no_bch,
 trainseq_no_bch)

            % de-modulation
            if isequal(modulation, 1)
                rx_bch = pamdemod(yd_bch, M);  % PAM
            elseif isequal(modulation, 2)
                rx_bch = qamdemod(yd_bch, M);  % QAM
                rx_no_bch = qamdemod(yd_no_bch, M);
            else
                rx_bch = pskdemod(yd_bch, M);  % PSK
            end

            % back to bits
            rxMSG_bch = msg2bits(msg_rx_bch, M);
            rxMSG_no_bch = msg2bits(rx2,M);

            % BCH decoder
            msg_rx_bch = bchdec(rx_bch, n, k);

            % Compute and store the BER for this iteration
            % We're interested in the BER, which is the 2nd output of
 BITERR
            [~, berVec_no_bch(i,j)] = biterr(bits(1+num_train:end-
delay*log2(M)), ...
                        rxMSG_no_bch(1+num_train
+delay*log2(M):end)); %no coding
            [~, berVec_bch(i,j)] = biterr(bits,rxMSG_bch); %with
 coding

        end  % End SNR iteration
    end      % End numIter iteration

    % Compute and plot the mean EQUALIZER BER
    ber_no_bch = mean(berVec_no_bch,1);
    ber_bch = mean(berVec_bch,1);

    % Compute the theoretical BER for this scenario
    % NOTE: there is no theoretical BER when you have a multipath
 channel
    if isequal(modulation, 1) || (M<4) % if M<4, qam berawgn is
 anyways pam berawng
        berTheory = berawgn(SNR_Vec, 'pam', M); % PAM
    elseif isequal(modulation, 2)
        berTheory = berawgn(SNR_Vec, 'qam', M); % QAM
    else
```

```
        berTheory = berawgn(SNR_Vec, 'psk', M); % PSK
    end

    Types = {'With BCH Encoding', 'No BCH Encoding', 'Theoretical'}';
    BER_Rate = [ber_bch, ber_no_bch, berTheory]';
    Section_2_Table = table(Types, BER_Rate)
end
```

Section 3: Reed Solomon

```
% At the same time that we were looking at BCH, we we were also trying
 to
% get the Reed-Solomon symbol level code working. We got closer to
% implementing this one, and our code for it is more cleaned up as
 compared
% to the bch encoding.
%
% Like with the BCH encoding, however, we left Reed Solomon before we
% hit the 10^-6 BER requierement, focusing our time on Convolutional
% encoding.


numIter = 20;  % The number of iterations of the simulation
n_sym = 1005;    % The number of symbols per packet
SNR_Vec = 12;
lenSNR = length(SNR_Vec);
M = 8;

% Modulation
%  - 1 = PAM
%  - 2 = QAM
%  - 3 = PSK
modulation = 3;

%chan = 1;          % No channel%
chan = [1 .2 .4]; % Somewhat invertible channel impulse response,
 Moderate ISI
%chan = [0.227 0.460 0.688 0.460 0.227]';   % Not so invertible,
 severe ISI

num_train = 350;

% Adaptive Algorithm300
%  - 0 = varlms
%  - 1 = lms
%  - 2 = rls
adaptive_algo = 2;

% Equalizer
%  - 0 = lineareq
%  - 1 = dfe
equalize_val = 0;
```

```
% equalizer parameters
NWeights = 13;
NWEIGHTS_Feedback = 6;
numRefTap = 1;
stepsize = 0.01;
forgetfactor = 1; % between 0 and 1

% Creating the equalizer:
% adaptive filter algorithm
if isequal(adaptive_algo, 0)
    AdapAlgo = varlms(stepsize,0.01,0,0.01);
elseif isequal(adaptive_algo, 1)
    AdapAlgo = lms(stepsize);
else
    AdapAlgo = rls(forgetfactor);
end

% Equalizer Object
if isequal(equalize_val, 0)
    eqobj = lineareq(NWeights,AdapAlgo); %comparable to an FIR
else
    eqobj = dfe(NWeights, NWEIGHTS_Feedback, AdapAlgo); %comparable to
 an IIR
end
eqobj.ResetBeforeFiltering = 0;
eqobj.RefTap = numRefTap;
delay = (numRefTap-1)/eqobj.nSampPerSym;
n_sym = n_sym-delay; % reed solomon requieres specific number of bits

% Reed Solomon parameters:
X = 3;
j = 2^X-1; % codeword length; default is 15, max allowed is 65,535
k = 3;
paritypos='end';

% Create a vector to store the BER computed during each iteration
berVec_rs = zeros(numIter, lenSNR);
berVec_no_rs = zeros(numIter, lenSNR);

for i = 1:numIter

    % message to transmit
    bits = randi(2,[(n_sym+delay)*log2(M), 1])-1;
    msg = bits2msg(bits, M);
    msg = reshape(msg,[(n_sym+delay)/k,k]);

    % encode message using reed solomon
    msg_gf = gf(msg,log2(M));
    msg_RS = rsenc(msg_gf,j,k);
    msg_RS_x = msg_RS.x;
    msg_RS_x = double(msg_RS_x(:));

    % modulation
    if isequal(modulation, 1)
```

```
        tx_rs = pammod(msg_RS_x, M);  % PAM modulation
        tx_no_rs = pammod(msg, M);
    elseif isequal(modulation, 2)
        tx_rs = qammod(msg_RS_x, M);% QAM modulation
        tx_no_rs = qammod(msg(:),M);
    else
        tx_rs = pskmod(msg_RS_x, M,[],'gray');  % PSK modulation
        tx_no_rs = pskmod(msg(:),M, [], 'gray');
    end
    trainseq_rs = tx_rs(1:num_train);
    trainseq_no_rs = tx_no_rs(1:num_train);
    % transmit (convolve) through channel
    if isequal(chan,1)
         txChan_rs = tx_rs;
        txChan_no_rs = tx_no_rs;
    elseif isa(chan,'channel.rayleigh')
        reset(chan) % Draw a different channel each iteration
        txChan_rs = filter(chan, tx_rs);
        txChan_no_rs = filter(chan, tx_no_rs);

    else
        txChan_rs = filter(chan, 1, tx_rs);  % Apply the channel.
        txChan_no_rs = filter(chan, 1, tx_no_rs);
    end

    % Convert from EbNo to SNR.
    % Note: Because No = 2*noiseVariance^2, we must add ~3 dB to get
 SNR (because 10*log10(2) ~= 3).
    noise_addition = 10*log10(log2(M));
    for snr = 1:lenSNR % one iteration of the simulation at each SNR
 Value


        txNoisy_rs = awgn(txChan_rs, noise_addition
+SNR_Vec(snr), 'measured'); % Add AWGN
        txNoisy_no_rs = awgn(txChan_no_rs, noise_addition
+SNR_Vec(snr), 'measured');

        yd_rs = equalize(eqobj, txNoisy_rs, trainseq_rs);
        yd_no_rs = equalize(eqobj, txNoisy_no_rs, trainseq_no_rs);

        % de-modulation
        if isequal(modulation, 1)
            rx_rs = pamdemod(yd_rs, M);  % PAM
            rx_no_rs = pamdemod(yd_no_rs, M);
        elseif isequal(modulation, 2)
            rx_rs = qamdemod(yd_rs, M);  % QAM
            rx_no_rs = qamdemod(yd_no_rs,M);
        else
            rx_rs = pskdemod(yd_rs, M,[],'gray');  % PSK
            rx_no_rs = pskdemod(yd_no_rs,M,[],'gray');
        end
        rx_rs = gf(reshape(rx_rs, [size(rx_rs,1)/j,j]),msg_gf.m,
 msg_gf.prim_poly);
```

```
        [rx_rs_decode, cnummerr] = rsdec(rx_rs,j,k);
        rx_rs_decode = rx_rs_decode.x;
        rx_rs_decode = double(rx_rs_decode(:));

        rxMSG_no_rs = msg2bits(rx_no_rs, M);
        rxMSG_rs= msg2bits(rx_rs_decode, M);

        % Compute and store the BER for this
        % We're interested in the BER, which is the 2nd output of
 BITERR
        numTrainBits = num_train*log2(M);
        [~, berVec_rs(i,snr)] = biterr(bits(1+num_train:end-
delay*log2(M)), rxMSG_rs(1+num_train+delay*log2(M):end));
        [~, berVec_no_rs(i,snr)] = biterr(bits(1+num_train:end-
delay*log2(M)), rxMSG_no_rs(1+num_train+delay*log2(M):end));
    end  % End SNR iteration
end      % End numIter iteration


% Compute and plot the mean EQUALIZER BER
ber_rs = mean(berVec_rs, 1);
ber_no_rs = mean(berVec_no_rs, 1);

% Compute the theoretical BER for this scenario
% NOTE: there is no theoretical BER when you have a multipath channel
if isequal(modulation, 1) || (M<4) % if M<4, qam berawgn is anyways
 pam berawng
    berTheory = berawgn(SNR_Vec, 'pam', M); % PAM
elseif isequal(modulation, 2)
    berTheory = berawgn(SNR_Vec, 'qam', M); % QAM
else
    berTheory = berawgn(SNR_Vec, 'psk', M, 'nondiff'); % PSK
end

Types = {'With Reed Solomon Encoding', 'No Reed Solomon
 Encoding', 'Theoretical'}';
BER_Rate = [ber_rs, ber_no_rs, berTheory]';
Section_3_Table = table(Types, BER_Rate);
disp(Section_3_Table)
```

Types	*BER_Rate*
'With Reed Solomon Encoding'	*0.0061351*
'No Reed Solomon Encoding'	*0.03878*
'Theoretical'	*6.3379e-05*

Section 4: Convolutional Encoding

```
% During our forray into BCH and Reed Solomon it became known to us
 that
% convolutional encoding is the superior error correcting code of the
```

```
% three. We therefore turned all our attention to it.
%
% To implement the convolutional encoding, we reworked our skeleton
 script
% we have been using up until now. We made it more similar to the
 MATLAB
% documentation.
%
% Some of the things we tried to do to optimize the bit rate, while
 still
% maintaining the threshold BER was to twiddle with the equalizer
% parameters as well as the length of the training sequence and the
% trellis & rate combination.
%
% We met the specs for M = 2, 8, and 16. However we spent the majority
 of
% our time was spent trying to get 32-ary to meet the specs. We
 decided to
% spend our effort on 32-ary QAM due to the boost in bits our bit rate
% would get - for every extra symbol we would be able to send an extra
 bit
% as compared to 16-ary QAM, which ammounts to 900 extra bits in
 total.
%
% As such even though we are trying to optimize 32-ary, if we had
 wanted to
% optimize 16-ary we could have lowered the training sequence length
 while
% still keeping the BER below the threshold, therby increasing the bit
% rate.
%
% Our results are below, and these are our 'final' results that we
 would
% like considered for our project.


close all; clear all;

% Parameters:
numIter = 1000;
n_sym = 1000;    % The number of symbols per packet
SNR_Vec = 8:2:16;

% The M-ary number. Two corresponds to binary modulation.
M1 = 16;
M2 = 32;

% Modulation type
%  - 1 = PAM
%  - 2 = QAM
%  - 3 = PSK
modulation = 2;

% Channel to use
```

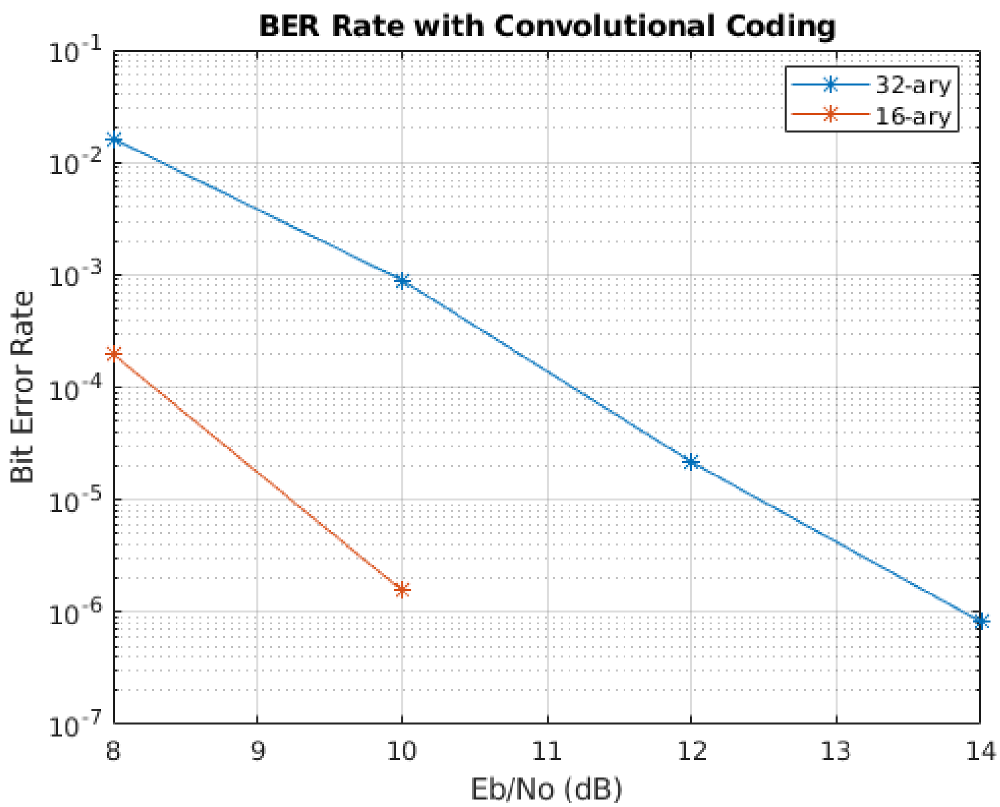
```
%chan = 1;          % No channel
chan = [1 .2 .4]; % Somewhat invertible channel impulse response,
 Moderate ISI
%chan = [0.227 0.460 0.688 0.460 0.227]'; % Not so invertible, severe
 ISI

% Time-varying Rayleigh multipath channel, try it if you dare.
% ts = 1/1000;
% chan = rayleighchan(ts,1);
% chan.pathDelays = [0 ts 2*ts];
% chan.AvgPathGaindB = [0 5 10];
% chan.StoreHistory = 1; % Uncomment if you want to do plot(chan)

% Number of training symbols (max=len(msg)=1000)
num_train = 100;

% Adaptive Algorithm
%  - 0 = varlms
%  - 1 = lms
%  - 2 = rls
adaptive_algo = 2;

% Equalizer
%  - 0 = lineareq
%  - 1 = dfe
equalize_val = 0;

% Convolutional encoding parameters
num_sym_per_frame = 1000;   % same as n_sym
trellis = poly2trellis(7,[171 133]);
rate = 1/2; % determined by trellis
tbl = 32;

% equalizer hyperparameters
n_weights = 6;
n_weights_feedback = 7;
sigconst_M1 = qammod((0:M1-1)',M1);
sigconst_M2 = qammod((0:M2-1)',M2);
numRefTap = 1;
stepsize = 0.005;
forgetfactor = 1; % between 0 and 1

% Building the equalizer:
% adaptive filter algorithm
if isequal(adaptive_algo, 0)
    adaptive_algo = varlms(stepsize, 0.01, 0, 0.01);
elseif isequal(adaptive_algo, 1)
    adaptive_algo = lms(stepsize);
else
    adaptive_algo = rls(forgetfactor, 0.1);
end

% equalizer object
if isequal(equalize_val, 0)
```

```
    eqobj = lineareq(n_weights, adaptive_algo); % like FIR
else
    eqobj = dfe(n_weights, n_weights_feedback, adaptive_algo); % like
 IIR
end
eqobj_M1 = eqobj;
eqobj_M2 = eqobj;

eqobj_M1.SigConst = sigconst_M1'; % Set signal constellation.
eqobj_M2.SigConst = sigconst_M2'; % Set signal constellation.
eqobj_M1.ResetBeforeFiltering = 0; % Maintain continuity between
 iterations.
eqobj_M2.ResetBeforeFiltering = 0; % Maintain continuity between
 iterations.
eqobj_M1.RefTap = numRefTap;
eqobj_M2.RefTap = numRefTap;
delay = (numRefTap-1)/eqobj.nSampPerSym;

ber_vec_32 = zeros(numIter, length(SNR_Vec));
ber_vec_16 = zeros(numIter, length(SNR_Vec));
for i = 1:numIter
    for j = 1:length(SNR_Vec)

        % Generate binary data and convert to symbols
        bits_16 = randi([0 1], (num_sym_per_frame)*log2(M1), 1);
        bits_32 = randi([0 1], (num_sym_per_frame)*log2(M2), 1);

        % Convolutionally encode the data
        data_enc_16 = convenc(bits_16, trellis);
        data_enc_32 = convenc(bits_32, trellis);

        % QAM modulate
        tx_signal_16 = qammod(data_enc_16, M1, 'InputType', 'bit', ...
                                                'UnitAveragePower',
 true);
        tx_signal_32 = qammod(data_enc_32, M2, 'InputType', 'bit', ...
                                                'UnitAveragePower',
 true);

        train_seq_16 = tx_signal_16(1:num_train);
        train_seq_32 = tx_signal_32(1:num_train);

        % pass through channel
        txChan_16 = filter(chan,1,tx_signal_16);
        txChan_32 = filter(chan,1,tx_signal_32);


        % Pass through AWGN channel and equalize
        noise_addition_M1 = 10*log10(log2(M1)*rate);
        noise_addition_M2 = 10*log10(log2(M2)*rate);

        rx_mod_signal_16 = awgn(txChan_16, SNR_Vec(j) +
 noise_addition_M1, 'measured');
```

```
        rx_mod_signal_32 = awgn(txChan_32, SNR_Vec(j) +
 noise_addition_M2, 'measured');

        rx_demod_signal_16 = equalize(eqobj_M1, rx_mod_signal_16,
 train_seq_16);
        rx_demod_signal_32 = equalize(eqobj_M2, rx_mod_signal_32,
 train_seq_32);

        % Demodulate the noisy signal using hard decision (bit) and
        % soft decision (approximate LLR) approaches.
        noise_var_M1 = 10.^(-(SNR_Vec(j) + noise_addition_M1)/10);
        noise_var_M2 = 10.^(-(SNR_Vec(j) + noise_addition_M2)/10);

        rx_data_soft_16 = qamdemod(rx_demod_signal_16,
 M1, 'OutputType', ...
                    'approxllr', 'UnitAveragePower', true, ...
                    'NoiseVariance', noise_var_M1);
        rx_data_soft_32 = qamdemod(rx_demod_signal_32,
 M2, 'OutputType', ...
                    'approxllr', 'UnitAveragePower', true, ...
                    'NoiseVariance', noise_var_M2);

        % Viterbi algo to decode the demodulated data
        dataSoft_16 = vitdec(rx_data_soft_16, trellis,
 tbl, 'cont', 'unquant');
        dataSoft_32 = vitdec(rx_data_soft_32, trellis,
 tbl, 'cont', 'unquant');

        % Calculate the number of bit errors in the frame. Adjust for
 the
        % decoding delay, which is equal to the traceback depth.
        [~, ber_vec_16(i,j)] = biterr(bits_16(num_train:end-tbl-
delay*log2(M1)), ...
                            dataSoft_16(num_train+tbl
+delay*log2(M1):end));
        [~, ber_vec_32(i,j)] = biterr(bits_32(num_train:end-tbl-
delay*log2(M2)), ...
                            dataSoft_32(num_train+tbl
+delay*log2(M2):end));

    end    % end of SNR iteration
end        % end of numIter iteration

ber_16 = mean(ber_vec_16, 1);
ber_32 = mean(ber_vec_32, 1);

semilogy(SNR_Vec, ber_32, '-*', 'DisplayName', '32-ary')
hold on
semilogy(SNR_Vec, ber_16, '-*', 'DisplayName', '16-ary')
legend('location','best')
grid
xlabel('Eb/No (dB)')
ylabel('Bit Error Rate')
title('BER Rate with Convolutional Coding')
```

```
M = [M1; M2];
BER = [ber_16(3); ber_32(3)];
Symbol_Rate = [((n_sym - num_train) / 1000); ((n_sym - num_train) /
 1000)];

% the number of usuable bits per symbol sent
Bit_Rate = [log2(M1)*((n_sym - num_train) / 1000); ...
                (log2(M2)*((n_sym - num_train) / 1000))];
Section_4_Table = table(M, BER, Symbol_Rate, Bit_Rate);
disp(Section_4_Table)
```

M	BER	Symbol_Rate	Bit_Rate
16	0	0.9	3.6
32	2.1565e-05	0.9	4.5



Helper Functions

```
% As in part 1, we have our symbol-bit conversion helper functions
 below.


function [msg] = bits2msg(bits, M)
```

```
    % Convert the message from bits into the correct integer values
    % based on the inputted M-ary modulation.
    % NOTE: M has to be a multiple of 2.

    % The length of bits that will be converted into decimal.
    len = log2(M);

    msg = zeros(size(bits,1)/len, 1);

    for i = 1:size(bits,1)/len
        msg(i) = bi2de(bits(1+(i-1)*len : 1+(i-1)*len + (len-1))');
    end

end


function [bits] = msg2bits(msg, M)
    % Convert the message from integers into the bit values
    % based on the inputted M-ary modulation.
    % NOTE: M has to be a multiple of 2.

    % The length of bits that will be converted into decimal.
    len = log2(M);

    bits = zeros(1, size(msg,1)*len);

    for i = 1:size(msg,1)
        bits(1+(i-1)*len:1:1+(i-1)*len + (len-1)) = de2bi(msg(i),
 len);
    end
    bits = bits';
end
```

Types	*BER_Rate*
'Equalized'	*0.23753*
'Not Equalized'	*0.14479*
'Theoretical'	*0.00013866*

Published with MATLAB® R2018b