Log-Likelihood Ratio (LLR) Demodulation

Try This Example

View MATLAB Command

This example shows the BER performance improvement for QPSK modulation when using log-likelihood ratio (LLR) instead of hard-decision demodulation in a convolutionally coded communication link. With LLR demodulation, one can use the Viterbi decoder either in the unquantized decoding mode or the soft-decision decoding mode. Unquantized decoding, where the decoder inputs are real values, though better in terms of BER, is not practically viable. In the more practical soft-decision decoding, the demodulator output is quantized before being fed to the decoder. It is generally observed that this does not incur a significant cost in BER while significantly reducing the decoder complexity. We validate this experimentally through this example.

For a Simulink™ version of this example, see LLR vs. Hard Decision Demodulation in Simulink.

Initialization

Initialize simulation parameters.

```
M = 4;                 % Modulation order
k = log2(M);           % Bits per symbol
bitsPerIter = 1.2e4; % Number of bits to simulate
EbNo = 3;              % Information bit Eb/No in dB
```

Initialize coding properties for a rate 1/2, constraint length 7 code.

```
codeRate = 1/2;          % Code rate of convolutional encoder
constLen = 7;            % Constraint length of encoder
codeGenPoly = [171 133]; % Code generator polynomial of encoder
tblen = 32;              % Traceback depth of Viterbi decoder
trellis = poly2trellis(constLen,codeGenPoly);
```

Create a `comm.ConvolutionalEncoder` System object™ by using `trellis` as an input.

```
enc = comm.ConvolutionalEncoder(trellis);
```

Modulator and Channel

Create a `comm.QPSKModulator` and two `comm.QPSKDemodulator` System objects. Configure the first demodulator to output hard-decision bits. Configure the second to output LLR values.

```
qpskMod = comm.QPSKModulator('BitInput',true);
demodHard = comm.QPSKDemodulator('BitOutput',true,...
    'DecisionMethod','Hard decision');
demodLLR = comm.QPSKDemodulator('BitOutput',true,...
    'DecisionMethod','Log-likelihood ratio');
```

Create an `comm.AWGNChannel` object. The signal going into the AWGN channel is the modulated encoded signal. To achieve the required noise level, adjust the Eb/No for coded bits and multi-bit symbols. Set this as the `EbNo` of the channel object.

```
chan = comm.AWGNChannel('NoiseMethod','Signal to noise ratio (Eb/No)', ...
    'BitsPerSymbol',k);
EbNoCoded = EbNo + 10*log10(codeRate);
chan.EbNo = EbNoCoded;
```

Viterbi Decoding

Create `comm.ViterbiDecoder` objects to act as the hard-decision, unquantized, and soft-decision decoders. For all three decoders, set the traceback depth to `tblen`.

```
decHard = comm.ViterbiDecoder(trellis,'InputFormat','Hard', ...
    'TracebackDepth',tblen);

decUnquant = comm.ViterbiDecoder(trellis,'InputFormat','Unquantized', ...
    'TracebackDepth',tblen);

decSoft = comm.ViterbiDecoder(trellis,'InputFormat','Soft', ...
    'SoftInputWordLength',3,'TracebackDepth',tblen);
```

Quantization for soft-decoding

Before using a `comm.ViterbiDecoder` object in the soft-decision mode, the output of the demodulator needs to be quantized. This example uses a `comm.ViterbiDecoder` object with a `SoftInputWordLength` of 3. This value is a good compromise between short word lengths and a small BER penalty. Define parition points for 3-bit quantization.

```
snrdB = EbNoCoded + 10*log10(k);
NoiseVariance = 10.^(-snrdB/10);
demodLLR.Variance = NoiseVariance;
paritionPoints = (-1.5:0.5:1.5)/NoiseVariance;
```

Calculating the Error Rate

Create `comm.ErrorRate` objects to compare the decoded bits to the original transmitted bits. The Viterbi decoder creates a delay in the decoded bit stream output equal to the traceback length. To account for this delay, set the `ReceiveDelay` property of the `comm.ErrorRate` objects to `tblen`.

```
errHard = comm.ErrorRate('ReceiveDelay',tblen);
errUnquant = comm.ErrorRate('ReceiveDelay',tblen);
errSoft = comm.ErrorRate('ReceiveDelay',tblen);
```

System Simulation

Generate `bitsPerIter` message bits. Then convolutionally encode and modulate the data.

```
txData = randi([0 1],bitsPerIter,1);
encData = enc(txData);
modData = qpskMod(encData);
```

Pass the modulated signal through an AWGN channel.

```
rxSig = chan(modData);
```

Demodulate the received signal and output hard-decision bits.

```
hardData = demodHard(rxSig);
```

Demodulate the received signal and output LLR values.

```
LLRData = demodLLR(rxSig);
```

Hard-decision decoding

Pass the demodulated data through the Viterbi decoder. Compute the error statistics.

```
rxDataHard = decHard(hardData);
berHard = errHard(txData,rxDataHard);
```

Unquantized decoding

Pass the demodulated data through the Viterbi decoder. Compute the error statistics.

```
rxDataUnquant = decUnquant(LLRData);
```

```
berUnquant = errUnquant(txData,rxDataUnquant);
```

Soft-decision decoding

Pass the demodulated data to the `quantiz` function. This data must be multiplied by `-1` before being passed to the quantizer, because, in soft-decision mode, the Viterbi decoder assumes that positive numbers correspond to 1s and negative numbers to 0s. Pass the quantizer output to the Viterbi decoder. Compute the error statistics.

```
quantizedValue = quantiz(-LLRData,paritionPoints);
rxDataSoft = decSoft(double(quantizedValue));
berSoft = errSoft(txData,rxDataSoft);
```

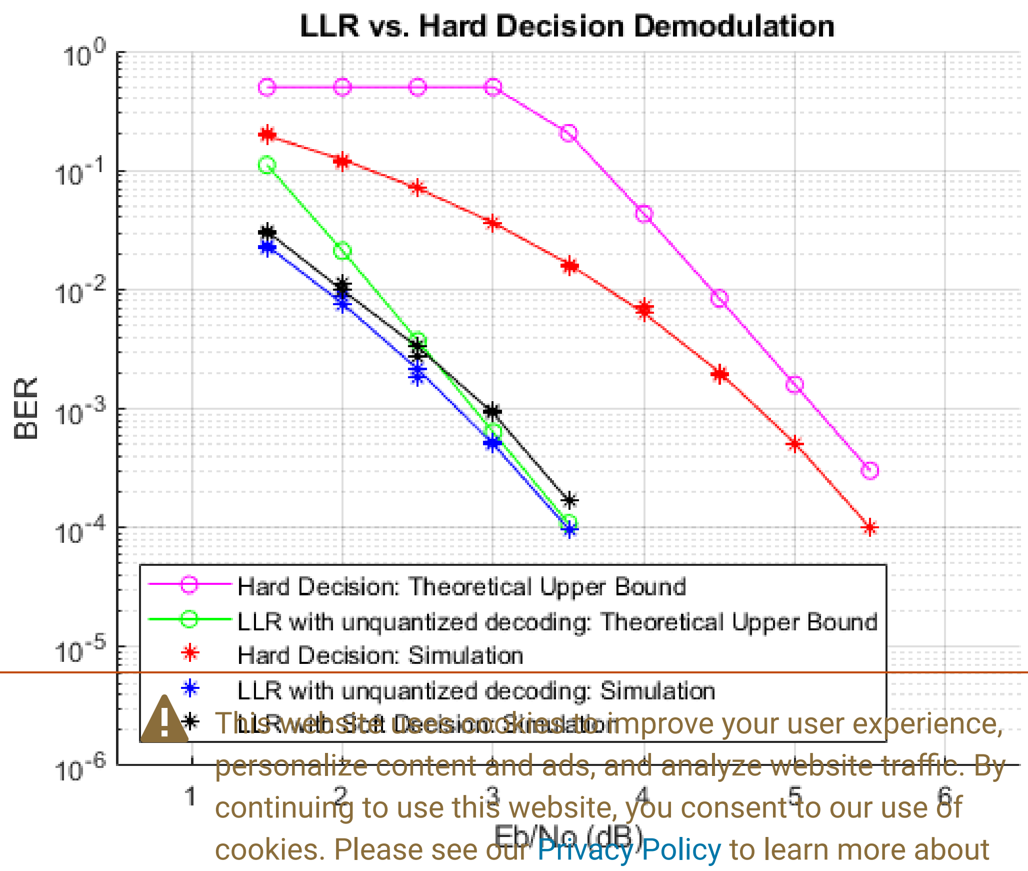
Running Simulation Example

Simulate the previously described communications system over a range of Eb/No values by executing the simulation file `simLLRvsHD`. It plots BER results as they are generated. BER results for hard-decision demodulation and LLR demodulation with unquantized and soft-decision decoding are plotted in red, blue, and black, respectively. A comparison of simulation results with theoretical results is also shown. Observe that the BER is only slightly degraded by using soft-decision decoding instead of unquantized decoding. The gap between the BER curves for soft-decision decoding and the theoretical bound can be narrowed by increasing the number of quantizer levels.

This example may take some time to compute BER results. If you have the Parallel Computing Toolbox™ (PCT) installed, you can set `usePCT` to `true` to run the simulation in parallel. In this case, the file `LLRvsHDwithPCT` is run.

To obtain results over a larger range of Eb/No values, modify the appropriate supporting files. Note that you can obtain more statistically reliable results by collecting more errors.

```
usePCT = false;
if usePCT && license('checkout','Distrib_Computing_Toolbox') ...
        && ~isempty(ver('parallel'))
    LLRvsHDwithPCT(1.5:0.5:5.5,5);
else
    simLLRvsHD(1.5:0.5:5.5,5);
end
```



This website uses cookies to improve your user experience, personalize content and ads, and analyze website traffic. By continuing to use this website, you consent to our use of cookies. Please see our Privacy Policy to learn more about cookies and how to change your settings.

×

Appendix

The following functions are used in this example:

- simLLRvsHD.m — Simulates system without PCT.
- LLRvsHDwithPCT.m — Simulates system with PCT.
- simLLRvsHDPCT.m — Helper function called by LLRvsHDwithPCT.