Turbo and LDPC Codes: Implementation, Simulation, and Standardization

June 7, 2006

Matthew Valenti
Rohit Iyer Seshadri
West Virginia University
Morgantown, WV 26506-6109
mvalenti@wvu.edu

Tutorial Overview

- Channel capacity
- Convolutional codes
  - the MAP algorithm
- Turbo codes
  - Standard binary turbo codes: UMTS and cdma2000
  - Duobinary CRSC turbo codes: DVB-RCS and 802.16
- LDPC codes
  - Tanner graphs and the message passing algorithm
  - Standard binary LDPC codes: DVB-S2
- Bit interleaved coded modulation (BICM)
  - Combining high-order modulation with a binary capacity approaching code.
- EXIT chart analysis of turbo codes

1:15 PM Valenti
3:15 PM Iyer Seshadri
4:30 PM Valenti
Software to Accompany Tutorial

- Iterative Solution’s Coded Modulation Library (CML) is a library for simulating and analyzing coded modulation.
- Available for free at the Iterative Solutions website: www.iterativesolutions.com
- Runs in matlab, but uses c-mex for efficiency.
- Supported features:
  - Simulation of BICM
    - Turbo, LDPC, or convolutional codes.
    - PSK, QAM, FSK modulation.
    - BICM-ID: Iterative demodulation and decoding.
  - Generation of ergodic capacity curves (BICM/CM constraints).
  - Information outage probability in block fading.
  - Calculation of throughput of hybrid-ARQ.
- Implemented standards:
  - Binary turbo codes: UMTS/3GPP, cdma2000/3GPP2.
  - Duobinary turbo codes: DVB-RCS, wimax/802.16.
  - LDPC codes: DVB-S2.

Noisy Channel Coding Theorem

- Every channel has associated with it a capacity C.
  - Measured in bits per channel use (modulated symbol).
- The channel capacity is an upper bound on information rate r.
  - There exists a code of rate r < C that achieves reliable communications.
    - Reliable means an arbitrarily small error probability.
Computing Channel Capacity

The capacity is the *mutual information* between the channel’s input $X$ and output $Y$ maximized over all possible input distributions:

$$C = \max_{p(x)} \{ I(X;Y) \}$$

$$= \max_{p(x)} \left\{ \int \int p(x,y) \log_2 \frac{p(x,y)}{p(x)p(y)} \, dx \, dy \right\}$$

Capacity of AWGN with Unconstrained Input

Consider an AWGN channel with 1-dimensional input:

- $y = x + n$
- where $n$ is Gaussian with variance $N_0/2$
- $x$ is a signal with average energy (variance) $E_s$

The capacity in this channel is:

$$C = \max_{p(x)} \{ I(X;Y) \} = \frac{1}{2} \log_2 \left( \frac{2E_s}{N_0} + 1 \right) = \frac{1}{2} \log_2 \left( \frac{2E_b}{N_0} + 1 \right)$$

- where $E_b$ is the energy per (information) bit.

This capacity is achieved by a Gaussian input $x$.
- This is not a practical modulation.
Capacity of AWGN with BPSK Constrained Input

If we only consider antipodal (BPSK) modulation, then

\[ X = \pm \sqrt{E_s} \]

and the capacity is:

\[
C = \max_{p(x)} \left\{ I(X;Y) \right\} \\
= I(X;Y)_{p(x),p=\frac{1}{2}} \\
= H(Y) - H(N) \\
= \int_{-\infty}^{\infty} p(y) \log_2 p(y) dy - \frac{1}{2} \log_2 (\pi e N_o) \\
\]

This term must be integrated numerically with

\[
p_x(y) = p_y(x) = \int_{-\infty}^{\lambda} p_x(x)p_y(y-x) dx 
\]

Capacity of AWGN w/ 1-D Signaling

It is theoretically impossible to operate in this region.
A convolutional encoder comprises:
- k input streams
  - We assume k=1 throughout this tutorial.
- n output streams
- m delay elements arranged in a shift register.
- Combinatorial logic (OR gates).
  - Each of the n outputs depends on some modulo-2 combination of the k current inputs and the m previous inputs in storage.

The constraint length is the maximum number of past and present input bits that each output bit can depend on.
- K = m + 1
A convolutional encoder is a finite state machine, and can be represented in terms of a state diagram.

Since \( k = 1 \),
2 branches enter and 2 branches leave each state.

A trellis is an expansion of the state diagram which explicitly shows the passage of time.

- All the possible states are shown for each instant of time.
- Time is indicated by a movement to the right.
- The input data bits and output code bits are represented by a unique path through the trellis.
**Trellis Diagram**

Every branch corresponds to a particular data bit and 2-bits of the code word.

input and output bits for time $L = 4$

every sequence of input data bits corresponds to a unique path through the trellis

initial state

ew state after first bit is encoded

$m = 2$ tail bits

final state

---

**Recursive Systematic Convolutional (RSC) Codes**

- An **RSC** encoder is constructed from a standard convolutional encoder by feeding back one of the outputs.
- An RSC code is **systematic**.
  - The input bits appear directly in the output.
- An RSC encoder is an **Infinite Impulse Response** (IIR) Filter.
  - An arbitrary input will cause a “good” (high weight) output with high probability.
  - Some inputs will cause “bad” (low weight) outputs.
With an RSC code, the output labels are the same. However, input labels are changed so that each state has an input “0” and an input “1”.

Messages labeling transitions that start from $S_1$ and $S_2$ are complemented.

Trellis Diagram of RSC Code

$m = 2$ tail bits no longer all-zeros must be calculated by the encoder
Convolutional Codewords

- Consider the trellis section at time $t$.
  - Let $S(t)$ be the encoder state at time $t$.
  - When there are four states, $S(t) \in \{S_0, S_1, S_2, S_3\}$
- Let $u(t)$ be the message bit at time $t$.
  - The encoder state $S(t)$ depends on $u(t)$ and $S(t-1)$
- Depending on its initial state $S(t-1)$ and the final state $S(t)$, the encoder will generate an $n$-bit long word
  - $x(t) = (x_1, x_2, \ldots, x_n)$
- The word is transmitted over a channel during time $t$, and the received signal is:
  - $y(t) = (y_1, y_2, \ldots, y_n)$
  - For BPSK, each $y = (2x-1) + n$
- If there are $L$ input data bits plus $m$ tail bits, the overall transmitted codeword is:
  - $x = [x(1), x(2), \ldots, x(L), \ldots, x(L+m)]$
- And the received codeword is:
  - $y = [y(1), y(2), \ldots, y(L), \ldots, y(L+m)]$

MAP Decoding

- The goal of the maximum a posteriori (MAP) decoder is to determine $P(u(t)=1 \mid y)$ and $P(u(t)=0 \mid y)$ for each $t$.
  - The probability of each message bit, given the entire received codeword.
- These two probabilities are conveniently expressed as a log-likelihood ratio:
  $$\lambda(t) = \log \frac{P[u(t)=1 \mid y]}{P[u(t)=0 \mid y]}$$
Determining Message Bit Probabilities from the Branch Probabilities

- Let $p_{ij}(t)$ be the probability that the encoder made a transition from $S_i$ to $S_j$ at time $t$, given the entire received codeword.
  - $p_{ij}(t) = P(S_i(t-1) \rightarrow S_j(t) \mid y)$
  - where $S_j(t)$ means that $S(t)=S_j$
- For each $t$,
  \[\sum_{S_j \rightarrow S_i} P(S_i(t-1) \rightarrow S_j(t) \mid y) = 1\]
- The probability that $u(t) = 1$ is
  \[P(u(t) = 1 \mid y) = \sum_{S_j \rightarrow S_i, u=1} P(S_i(t-1) \rightarrow S_j(t) \mid y)\]
- Likewise
  \[P(u(t) = 0 \mid y) = \sum_{S_j \rightarrow S_i, u=0} P(S_i(t-1) \rightarrow S_j(t) \mid y)\]

Determining the Branch Probabilities

- Let $\gamma_{ij}(t)$ = Probability of transition from state $S_i$ to state $S_j$ at time $t$, given just the received word $y(t)$
  - $\gamma_{ij}(t) = P(S_i(t-1) \rightarrow S_j(t) \mid y(t))$
- Let $\alpha_i(t-1)$ = Probability of starting at state $S_i$ at time $t$, given all symbols received prior to time $t$.
  - $\alpha_i(t-1) = P(S_i(t-1) \mid y(1), y(2), ..., y(t-1))$
- $\beta_j(t)$ = Probability of ending at state $S_j$ at time $t$, given all symbols received after time $t$.
  - $\beta_j(t) = P(S_j(t) \mid y(t+1), ..., y(L+m))$
- Then the branch probability is:
  - $p_{ij}(t) = \alpha_i(t-1) \gamma_{ij}(t) \beta_j(t)$
Computing $\alpha$

- $\alpha$ can be computed recursively.
- Prob. of path going through $S_i(t-1)$ and terminating at $S_j(t)$, given $y(1)\ldots y(t)$ is:
  - $\alpha_i(t-1) \gamma_{i,j}(t)$
- Prob. of being in state $S_j(t)$, given $y(1)\ldots y(t)$ is found by adding the probabilities of the two paths terminating at state $S_j(t)$.
- For example,
  - $\alpha_3(t) = \alpha_1(t-1) \gamma_{1,3}(t) + \alpha_3(t-1) \gamma_{3,3}(t)$
- The values of $\alpha$ can be computed for every state in the trellis by “sweeping” through the trellis in the \textit{forward} direction.

Computing $\beta$

- Likewise, $\beta$ is computed recursively.
- Prob. of path going through $S_i(t+1)$ and terminating at $S_j(t)$, given $y(t+1), \ldots, y(L+m)$
  - $\beta_j(t+1) \gamma_{i,j}(t+1)$
- Prob. of being in state $S_i(t)$, given $y(t+1), \ldots, y(L+m)$ is found by adding the probabilities of the two paths starting at state $S_i(t)$.
- For example,
  - $\beta_3(t) = \beta_2(t+1) \gamma_{1,2}(t+1) + \beta_3(t+1) \gamma_{3,3}(t+1)$
- The values of $\beta$ can be computed for every state in the trellis by “sweeping” through the trellis in the \textit{reverse} direction.
Computing $\gamma$

- Every branch in the trellis is labeled with:
  - $\gamma_{i,j}(t) = P(S_i(t-1) \rightarrow S_j(t) \mid y(t))$
- Let $x_{i,j} = (x_1, x_2, \ldots, x_n)$ be the word generated by the encoder when transitioning from $S_i$ to $S_j$:
  - $\gamma_{i,j}(t) = P(x_{i,j} \mid y(t))$
- From Bayes rule,
  - $\gamma_{i,j}(t) = P(x_{i,j} \mid y(t)) = P(y(t) \mid x_{i,j}) P(x_{i,j}) / P(y(t))$
- $P(y(t))$
  - Is not strictly needed because will be the same value for the numerator and denominator of the LLR $\lambda(t)$.
  - Instead of computing directly, can be found indirectly as a normalization factor (chosen for numerical stability)
- $P(x_{i,j})$
  - Initially found assuming that code bits are equally likely.
  - In a turbo code, this is provided to the decoder as “a priori” information.

Computing $P(y(t) \mid x_{i,j})$

- If BPSK modulation is used over an AWGN channel, the probability of code bit $y$ given $x$ is conditionally Gaussian:
  $$P(y \mid x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{ -\frac{(y-m_y)^2}{2\sigma^2} \right\}$$
  $$m_y = \sqrt{E_y}(2x-1)$$
  $$\sigma^2 = \frac{N_0}{2}$$
  - In Rayleigh fading, multiply $m_y$ by $a$, the fading amplitude.
- The conditional probability of the word $y(t)$
  $$P(y \mid x) = \prod_{i=1}^{n} p(y_i \mid x_i)$$
Overview of MAP algorithm

- Label every branch of the trellis with $\gamma_{i,j}(t)$.
- Sweep through trellis in forward-direction to compute $\alpha_i(t)$ at every node in the trellis.
- Sweep through trellis in reverse-direction to compute $\beta_j(t)$ at every node in the trellis.
- Compute the LLR of the message bit at each trellis section:

$$\lambda(t) = \log \frac{P[u(t) = 1 | y]}{P[u(t) = 0 | y]}$$

$$= \log \frac{\sum S_{i,j} \gamma_{i,j}(t) \beta_j(t)}{\sum S_{i,j} \gamma_{i,j}(t) \beta_j(t)}$$

- MAP algorithm also called the “forward-backward” algorithm (Forney).

Log Domain Decoding

- The MAP algorithm can be simplified by performing in the log domain.
  - exponential terms (e.g. used to compute $\gamma$) disappear.
  - multiplications become additions.
  - Addition can be approximated with maximization.
- Redefine all quantities:
  - $\gamma_{i,j}(t) = \log P( S_i(t-1) \rightarrow S_j(t) | y(t) )$
  - $\alpha_i(t-1) = \log P( S_i(t-1) | y(1), y(2), ..., y(t-1) )$
  - $\beta_j(t) = \log P( S_j(t) | y(t+1), ..., y(L+m) )$
- Details of the log-domain implementation will be presented later…
Parallel Concatenated Codes with Nonuniform Interleaving

- A stronger code can be created by encoding in parallel.
- A nonuniform interleaver scrambles the ordering of bits at the input of the second encoder.
  - Uses a pseudo-random interleaving pattern.
- It is very unlikely that both encoders produce low weight code words.
- MUX increases code rate from 1/3 to 1/2.

Random Coding Interpretation of Turbo Codes

- **Random codes** achieve the best performance.
  - Shannon showed that as $n \to \infty$, random codes achieve channel capacity.
- However, random codes are not feasible.
  - The code must contain enough structure so that decoding can be realized with actual hardware.
- **Coding dilemma:**
  - “All codes are good, except those that we can think of.”
- With turbo codes:
  - The nonuniform interleaver adds apparent randomness to the code.
  - Yet, they contain enough structure so that decoding is feasible.
Comparison of a Turbo Code and a Convolutional Code

First consider a K=12 convolutional code.
- \( d_{\text{min}} = 18 \)
- \( \beta_d = 187 \) (output weight of all \( d_{\text{min}} \) paths)

Now consider the original turbo code.
- Same complexity as the K=12 convolutional code
- Constraint length 5 RSC encoders
- \( k = 65,536 \) bit interleaver
- Minimum distance \( d_{\text{min}} = 6 \)
- \( a_d = 3 \) minimum distance code words
- Minimum distance code words have average information weight of only\
  \( f_d = 2 \)

Comparison of Minimum-distance Asymptotes

Convolutional code:
- \( d_{\text{min}} = 18 \)
- \( c_{d_{\text{min}}} = 187 \)
- \( P_e \approx (187)Q\left( \sqrt{\frac{E_b}{N_o}} \right) \)

Turbo code:
- \( d_{\text{min}} = 6 \)
- \( c_{d_{\text{min}}} = \frac{a_{d_{\text{min}}} W_{d_{\text{min}}}}{k} = \frac{3 \cdot 2}{65536} \)
- \( P_e \approx (9.2 \times 10^{-5})Q\left( \sqrt{\frac{E_b}{N_o}} \right) \)
The Turbo-Principle

Turbo codes get their name because the decoder uses feedback, like a turbo engine.

Performance as a Function of Number of Iterations

- $K = 5$
  - constraint length
- $r = 1/2$
  - code rate
- $L = 65,536$
  - interleaver size
  - number data bits
- Log-MAP algorithm
Summary of Performance Factors and Tradeoffs

- Latency vs. performance
  - Frame (interleaver) size $L$
- Complexity vs. performance
  - Decoding algorithm
  - Number of iterations
  - Encoder constraint length $K$
- Spectral efficiency vs. performance
  - Overall code rate $r$
- Other factors
  - Interleaver design
  - Puncture pattern
  - Trellis termination

Tradeoff: BER Performance versus Frame Size (Latency)

Plot showing BER performance versus $E_b/N_b$ for different frame sizes $K$. The graph includes details such as $K = 5$, rate $r = 1/2$, 18 decoder iterations, and AWGN Channel conditions.
Characteristics of Turbo Codes

- Turbo codes have extraordinary performance at low SNR.
  - Very close to the Shannon limit.
  - Due to a low multiplicity of low weight code words.
- However, turbo codes have a BER “floor”.
  - This is due to their low minimum distance.
- Performance improves for larger block sizes.
  - Larger block sizes mean more latency (delay).
  - However, larger block sizes are not more complex to decode.
  - The BER floor is lower for larger frame/interleaver sizes.
- The complexity of a constraint length $K_{TC}$ turbo code is the same as a $K = K_{CC}$ convolutional code, where:
  - $K_{CC} = 2 + K_{TC} + \log_2(\text{number decoder iterations})$

UMTS Turbo Encoder

- From 3GPP TS 25 212 v6.6.0, Release 6 (2005-09)
  - UMTS Multiplexing and channel coding
- Data is segmented into blocks of $L$ bits.
  - where $40 \leq L \leq 5114$
UMTS Interleaver: Inserting Data into Matrix

- Data is fed row-wise into a R by C matrix.
  - R = 5, 10, or 20.
  - 8 ≤ C ≤ 256
  - If L < RC then matrix is padded with dummy characters.

In the CML, the UMTS interleaver is created by the function `CreateUMTSInterleaver`.
Interleaving and Deinterleaving are implemented by `Interleave` and `Deinterleave`.

UMTS Interleaver: Intra-Row Permutations

- Data is permuted *within* each row.
  - Permutation rules are rather complicated.
  - See spec for details.
UMTS Interleaver: Inter-Row Permutations

- Rows are permuted.
  - If \( R = 5 \) or 10, the matrix is reflected about the middle row.
  - For \( R = 20 \) the rule is more complicated and depends on \( L \).
    - See spec for \( R = 20 \) case.

\[
\begin{array}{cccccccc}
X_{40} & X_{35} & X_{33} & X_{37} & X_{38} & X_{32} & X_{34} \\
X_{28} & X_{27} & X_{31} & X_{29} & X_{30} & X_{25} & X_{22} \\
X_{18} & X_{22} & X_{21} & X_{19} & X_{20} & X_{17} & X_{24} \\
X_{10} & X_{12} & X_{11} & X_{15} & X_{13} & X_{14} & X_{9} & X_{16} \\
X_{2} & X_{6} & X_{5} & X_{7} & X_{3} & X_{4} & X_{1} & X_{8}
\end{array}
\]

UMTS Interleaver: Reading Data From Matrix

- Data is read from matrix column-wise.

\[
\begin{array}{cccccccc}
X_{40} & X_{36} & X_{35} & X_{39} & X_{37} & X_{38} & X_{33} & X_{34} \\
X_{28} & X_{27} & X_{31} & X_{29} & X_{30} & X_{25} & X_{32} \\
X_{18} & X_{22} & X_{21} & X_{19} & X_{20} & X_{17} & X_{24} \\
X_{10} & X_{12} & X_{11} & X_{15} & X_{13} & X_{14} & X_{9} & X_{16} \\
X_{2} & X_{6} & X_{5} & X_{7} & X_{3} & X_{4} & X_{1} & X_{8}
\end{array}
\]

- Thus:
  - \( X'_{1} = X_{40} \)
  - \( X'_{2} = X_{26} \)
  - \( X'_{3} = X_{18} \)
  - \( X'_{38} = X_{24} \)
  - \( X'_{40} = X_{8} \)
UMTS Constituent RSC Encoder

- Upper and lower encoders are identical:
  - Feedforward generator is 15 in octal.
  - Feedback generator is 13 in octal.

Trellis Termination

- After the Lth input bit, a 3 bit tail is calculated.
  - The tail bit equals the fed back bit.
  - This guarantees that the registers get filled with zeros.
- Each encoder has its own tail.
  - The tail bits and their parity bits are transmitted at the end.
Output Stream Format

- The format of the output stream is:

\[ x_1 z_1 x_2 z_2 \ldots x_L z_L x_{L+1} z_{L+1} x_{L+2} z_{L+2} x_{L+3} z_{L+3} x_{L+4} z_{L+4} x_{L+5} z_{L+5} \]

- L data bits and their associated 2L parity bits (total of 3L bits)
- 3 tail bits for upper encoder and their 3 parity bits
- 3 tail bits for lower encoder and their 3 parity bits

Total number of coded bits = 3L + 12

Code rate: \[ r = \frac{L}{3L + 12} = \frac{1}{3} \]

Channel Model and LLRs

- Channel gain: a
  - Rayleigh random variable if Rayleigh fading
  - a = 1 if AWGN channel
- Noise
  - variance is: \[ \sigma^2 = \frac{1}{2r} \left( \frac{E_b}{N_0} \right) \approx \frac{3}{2} \left( \frac{E_b}{N_0} \right) \]

\[ \text{BPSK Modulator} \]

\[ \text{a} \]

\[ \text{n} \]

\[ y \]

\[ r \]

\[ \frac{2a}{\sigma^2} \]
SISO-MAP Decoding Block

This block is implemented in the CML by the SisoDecode function

- Inputs:
  - $\lambda_{u,i}$ LLR’s of the data bits. This comes from the other decoder r.
  - $\lambda_{c,i}$ LLR’s of the code bits. This comes from the channel observations r.

- Two output streams:
  - $\lambda_{u,o}$ LLR’s of the data bits. Passed to the other decoder.
  - $\lambda_{c,o}$ LLR’s of the code bits. Not used by the other decoder.

Turbo Decoding Architecture

- Initialization and timing:
  - Upper $\lambda_{u,i}$ input is initialized to all zeros.
  - Upper decoder executes first, then lower decoder.
Log-MAP Algorithm: Overview

- Log-MAP algorithm is MAP implemented in log-domain.
  - Multiplications become additions.
  - Additions become special "max*" operator (Jacobi logarithm)
- Log-MAP is similar to the Viterbi algorithm.
  - Except "max" is replaced by "max*" in the ACS operation.
- Processing:
  - Sweep through the trellis in forward direction using modified Viterbi algorithm.
  - Sweep through the trellis in backward direction using modified Viterbi algorithm.
  - Determine LLR for each trellis section.
  - Determine output extrinsic info for each trellis section.
The max* operator

- max* must implement the following operation:

\[ z = \max(x, y) + \ln(1 + \exp(-|y - x|)) \]
\[ = \max(x, y) + f_c(|y - x|) \]
\[ = \max^*(x, y) \]

- Ways to accomplish this:
  - C-function calls or large look-up-table. log-MAP
  - (Piecewise) linear approximation.
  - Rough correction value.
    \[ z = \max(x, y) + \begin{cases} 
    0 & \text{if } |y - x| > 1.5 \\
    0.5 & \text{if } |y - x| \leq 1.5 
    \end{cases} \text{ constant-log-MAP} \]
  - Max operator.
    \[ z = \max(x, y) \text{ max-log-MAP} \]

The Correction Function

dec. type option in SisoDecode
0 For linear-log-MAP (DEFAULT)
1 For max-log-MAP algorithm
2 For Constant-log-MAP algorithm
3 For log-MAP, correction factor from small nonuniform table and interpolation
4 For log-MAP, correction factor uses C function calls
The Trellis for UMTS

- Dotted line = data 0
- Solid line = data 1
- Note that each node has one each of data 0 and 1 entering and leaving it.
- The branch from node $S_i$ to $S_j$ has metric $\gamma_{ij}$

\[
\gamma_{ij} = \alpha_i + \beta_j + Z_{ij} + \lambda_i + \lambda_j
\]

Data bit associated with branch $S_i \rightarrow S_j$

The two code bits labeling with branch $S_i \rightarrow S_j$

Forward Recursion

- A new metric must be calculated for each node in the trellis using:

\[
\alpha_j = \max\{\alpha'_i + \gamma_{ij}, \alpha'_i + \gamma_{ij}\}
\]

where $i_1$ and $i_2$ are the two states connected to $j$.

- Start from the beginning of the trellis (i.e. the left edge).

- Initialize stage 0:

$$\alpha_0 = 0$$

$$\alpha_i = -\infty \text{ for all } i \neq 0$$
Backward Recursion

- A new metric must be calculated for each node in the trellis using:

\[
\beta_i = \max\left\{\left(\beta_{j_1} + \gamma_{j_1}ight) \left(\beta_{j_2} + \gamma_{j_2}\right)\right\}
\]

- where \(j_1\) and \(j_2\) are the two states connected to \(i\).
- Start from the end of the trellis (i.e. the right edge).
- Initialize stage L+3:

\[
\beta_0 = 0 \\
\beta_i = -\infty \text{ for all } i \neq 0
\]

Log-likelihood Ratio

- The likelihood of any one branch is:

\[
\alpha_i + \gamma_{\hat{y}_i} + \beta_i
\]

- The likelihood of data 1 is found by summing the likelihoods of the solid branches.
- The likelihood of data 0 is found by summing the likelihoods of the dashed branches.
- The log likelihood ratio (LLR) is:

\[
\Lambda(X_i) = \ln \left( \frac{P[Y_i = 1]}{P[Y_i = 0]} \right) \\
= \max_{\hat{y}_i \rightarrow (X_{i-1}, X_{i-2})} \left\{ \alpha_i + \gamma_{\hat{y}_i} + \beta_i \right\} \\
- \max_{\hat{y}_i \rightarrow (X_{i-1}, X_{i-2})} \left\{ \alpha_i + \gamma_{\hat{y}_i} + \beta_i \right\}
\]
Memory Issues

- A naïve solution:
  - Calculate $\alpha$’s for entire trellis (forward sweep), and store.
  - Calculate $\beta$’s for the entire trellis (backward sweep), and store.
  - At the $k$th stage of the trellis, compute $\lambda$ by combining $\gamma$’s with stored $\alpha$’s and $\beta$’s.
- A better approach:
  - Calculate $\beta$’s for the entire trellis and store.
  - Calculate $\alpha$’s for the $k$th stage of the trellis, and immediately compute $\lambda$ by combining $\gamma$’s with these $\alpha$’s and stored $\beta$’s.
  - Use the $\alpha$’s for the $k$th stage to compute $\alpha$’s for state $k+1$.
- Normalization:
  - In log-domain, $\alpha$’s can be normalized by subtracting a common term from all $\alpha$’s at the same stage.
  - Can normalize relative to $\alpha_0$, which eliminates the need to store $\alpha_0$.
  - Same for the $\beta$’s.

Sliding Window Algorithm

- Can use a sliding window to compute $\beta$’s
  - Windows need some overlap due to uncertainty in terminating state.
  - Calculate $\alpha$ and $\lambda$ over this region.
Extrinsic Information

- The extrinsic information is found by subtracting the corresponding input from the LLR output, i.e.
  \[
  \lambda_{u,i}^{\text{lower}} = \lambda_{u,o}^{\text{upper}} - \lambda_{u,i}^{\text{upper}} \\
  \lambda_{u,i}^{\text{upper}} = \lambda_{u,o}^{\text{lower}} - \lambda_{u,i}^{\text{lower}}
  \]

- It is necessary to subtract the information that is already available at the other decoder in order to prevent “positive feedback”.

- The extrinsic information is the amount of new information gained by the current decoder step.

Performance Comparison

BER of 640 bit turbo code

- max-log-MAP
- constant-log-MAP
- log-MAP

EB/N0 in dB

BER

Fading

AWGN

10 decoder iterations
cdma2000 uses a rate $\frac{1}{5}$ constituent encoder.

- Overall turbo code rate can be $1/5$, $1/4$, $1/3$, or $1/2$.
- Fixed interleaver lengths:
  - 378, 570, 762, 1146, 1530, 2398, 3066, 4602, 6138, 9210, 12282, or 20730

---

**Turbo and LDPC Codes**

6/7/2006

---

**Graph:**

- Performance of cdma2000 turbo code in AWGN with interleaver length 1530

- Bit Error Rate vs. $E_b/N_0$ in dB

- Curves for rates $1/5$, $1/4$, $1/3$, and $1/2$
Circular Recursive Systematic Convolutional (CRSC) Codes

- CRSC codes use the concept of tailbiting.
  - Sequence is encode so that initial state is same as final state.
- Advantage and disadvantages
  - No need for tail bits.
  - Need to encode twice.
  - Complicates decoder.

Duobinary codes

- Duobinary codes are defined over GF(4).
  - Two bits taken in per clock cycle.
  - Output is systematic and rate 2/4.
- Hardware benefits
  - Half as many states in trellis.
  - Smaller loss due to max-log-MAP decoding.
DVB-RCS

  - Consumer-grade Internet service over satellite.
  - 144 kbps to 2 Mbps satellite uplink.
  - Uses same antenna as downlink.
  - QPSK modulation.
- DVB-RCS uses a pair of duobinary CRSC codes.
- Key parameters:
  - input of N = k/2 couples
  - N = {48, 64, 212, 220, 228, 424, 432, 440, 752, 848, 856, 864}
  - r={1/3, 2/5, 1/2, 2/3, 3/4, 4/5, 6/7}

DVB-RCS: Influence of Decoding Algorithm

- rate r=1/3
- length N=212
- 8 iterations.
- AWGN.
DVB-RCS: Influence of Block Length

- Rate \( \frac{1}{3} \)
- Max-log-MAP
- 8 iterations
- AWGN

DVB-RCS: Influence of Code Rate

- \( N=212 \)
- Max-log-MAP
- 8 iterations
- AWGN
802.16 (WiMax)

- The standard specifies an optional convolutional turbo code (CTC) for operation in the 2-11 GHz range.
- Uses same duobinary CRSC encoder as DVB-RCS, though without output W.
- Modulation: BPSK, QPSK, 16-QAM, 64-QAM, 256-QAM.
- Key parameters:
  - Input message size 8 to 256 bytes long.
  - $r = \{1/2, 2/3, 3/4, 5/6, 7/8\}$

Prelude to LDPC Codes:
Review of Linear Block Codes

- $V_n = n$-dimensional vector space over $\{0,1\}$
- A $(n, k)$ linear block code with dataword length $k$, codeword length $n$ is a $k$-dimensional vector subspace of $V_n$
- A codeword $c$ is generated by the matrix multiplication $c = uG$, where $u$ is the $k$-bit long message and $G$ is a $k$ by $n$ generator matrix
- The parity check matrix $H$ is a $n-k$ by $n$ matrix of ones and zeros, such that if $c$ is a valid codeword then, $cH^T = 0$
- Each row of $H$ specifies a parity check equation. The code bits in positions where the row is one must sum (modulo-2) to zero
Low-Density Parity-Check Codes

- **Low-Density Parity-Check** (LDPC) codes are a class of linear block codes characterized by sparse parity check matrices $H$.
  - $H$ has a low-density of 1’s.

- LDPC codes were originally invented by Robert Gallager in the early 1960’s but were largely ignored until they were “rediscovered” in the mid-1990’s by MacKay.

- Sparseness of $H$ can yield large minimum distance $d_{\text{min}}$ and reduces decoding complexity.

- Can perform within 0.0045 dB of Shannon limit.

Decoding LDPC codes

- Like Turbo codes, LDPC can be decoded iteratively.
  - Instead of a trellis, the decoding takes place on a **Tanner graph**.
  - Messages are exchanged between the v-nodes and c-nodes.
  - Edges of the graph act as information pathways.

- Hard decision decoding
  - Bit-flipping algorithm.

- Soft decision decoding
  - **Sum-product** algorithm
    - Also known as message passing/belief propagation algorithm.
  - **Min-sum** algorithm
    - Reduced complexity approximation to the sum-product algorithm.

- In general, the per-iteration complexity of LDPC codes is less than it is for turbo codes.
  - However, many more iterations may be required (max=100; avg=30).
  - Thus, overall complexity can be higher than turbo.
A Tanner graph is a bipartite graph that describes the parity check matrix $H$.

There are two classes of nodes:

- **Variable-nodes**: Correspond to bits of the codeword or equivalently, to columns of the parity check matrix.
  - There are $n$ v-nodes.

- **Check-nodes**: Correspond to parity check equations or equivalently, to rows of the parity check matrix.
  - There are $m=n-k$ c-nodes.

- **Bipartite** means that nodes of the same type cannot be connected (e.g. a c-node cannot be connected to another c-node).

The $i$th check node is connected to the $j$th variable node iff the $(i,j)$th element of the parity check matrix is one, i.e. if $h_{ij} = 1$.

- All of the v-nodes connected to a particular c-node must sum (modulo-2) to zero.

---

**Example: Tanner Graph for (7,4) Hamming Code**

$$H = \begin{bmatrix}
1 & 1 & 1 & 0 & 1 & 0 & 0 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 \\
1 & 0 & 1 & 1 & 0 & 0 & 1 \\
\end{bmatrix}$$
More on Tanner Graphs

- A cycle of length \( l \) in a Tanner graph is a path of \( l \) distinct edges which closes on itself.
- The girth of a Tanner graph is the minimum cycle length of the graph.
  - The shortest possible cycle in a Tanner graph has length 4.

![Tanner Graph Diagram](image)

Bit-Flipping Algorithm: (7,4) Hamming Code

- Received code word: \( y_0 = 1, y_1 = 1, y_2 = 1, y_3 = 0, y_4 = 0, y_5 = 1, y_6 = 1 \)
- Transmitted code word: \( c_0 = 1, c_1 = 0, c_2 = 1, c_3 = 0, c_4 = 0, c_5 = 0, c_6 = 1 \)
Bit-Flipping Algorithm:
(7,4) Hamming Code

\begin{align*}
y_0 &= 1 \\
y_1 &= 0 \\
y_2 &= 1 \\
y_3 &= 1 \\
y_4 &= 0 \\
y_5 &= 0 \\
y_6 &= 1 \\
\end{align*}
Generalized Bit-Flipping Algorithm

- Step 1: Compute parity-checks
  - If all checks are zero, stop decoding
- Step 2: Flip any digit contained in \( T \) or more failed check equations
- Step 3: Repeat 1 to 2 until all the parity checks are zero or a maximum number of iterations are reached
- The parameter \( T \) can be varied for a faster convergence

Generalized Bit Flipping: (15,7) BCH Code
Generalized Bit Flipping: (15,7) BCH Code

\[
\begin{align*}
\bar{f}_0 &= 0 \\
\bar{f}_1 &= 0 \\
\bar{f}_2 &= 0 \\
\bar{f}_3 &= 0 \\
\bar{f}_4 &= 0 \\
\bar{f}_5 &= 0 \\
\bar{f}_6 &= 0 \\
\bar{f}_7 &= 1 \\
\bar{y}_0 &= 0 \\
\bar{y}_1 &= 0 \\
\bar{y}_2 &= 0 \\
\bar{y}_3 &= 0 \\
\bar{y}_4 &= 0 \\
\bar{y}_5 &= 0 \\
\bar{y}_6 &= 0 \\
\bar{y}_7 &= 0 \\
\bar{y}_8 &= 0 \\
\bar{y}_9 &= 0 \\
\bar{y}_{10} &= 0 \\
\bar{y}_{11} &= 0 \\
\bar{y}_{12} &= 0 \\
\bar{y}_{13} &= 0 \\
\bar{y}_{14} &= 0
\end{align*}
\]
**Sum-Product Algorithm: Notation**

- $Q_0 = P(c_i = 0 | y, S_i), Q_1 = P(c_i = 1 | y, S_i)$
- $S_i =$ event that bits in $c$ satisfy the $d_v$ parity check equations involving $c_i$
- $q_{ij}(b) =$ extrinsic info to be passed from v-node $i$ to c-node $j$
  - Probability that $c_i = b$ given extrinsic information from check nodes and channel sample $y_i$
- $r_{ji}(b) =$ extrinsic info to be passed from c-node $j$ to v-node $i$
  - Probability of the $j$th check equation being satisfied given that $c_i = b$
- $C_i = \{j: h_{ji} = 1\}$
  - This is the set of row location of the 1's in the $i$th column
- $C_{ij} = \{j: h_{ji} = 1\} \setminus \{j\}$
  - The set of row locations of the 1's in the $i$th column, excluding location $j$
- $R_j = \{i: h_{ji} = 1\}$
  - This is the set of column location of the 1's in the $j$th row
- $R_{ji} = \{i: h_{ji} = 1\} \setminus \{i\}$
  - The set of column locations of the 1's in the $j$th row, excluding location $i$

**Sum-Product Algorithm**

Step 1: Initialize $q_{ij}(0) = 1 - p_i = 1/(1 + \exp(-2y_i/\sigma^2))$
$q_{ij}(1) = p_i = 1/(1 + \exp(2y_i/\sigma^2))$

$q_{ij}(b) =$ probability that $c_i = b$, given the channel sample
Sum-Product Algorithm

Step 2: At each c-node, update the messages

\[ r_j^{(0)} = \frac{1}{2} + \frac{1}{2} \prod_{i \in \mathcal{C}_j} (1 - 2q_i^{(0)}) \]
\[ r_j^{(1)} = 1 - r_j^{(0)} \]

\[ r_j(b) \] = probability that the \( j \)-th check equation is satisfied given \( c_j = b \)

Sum-Product Algorithm

Step 3: Update \( q_{kq}(0) \) and \( q_{kq}(1) \)

\[ q_{kq}^{(0)} = k_q (1 - p_r) \prod_{r \in \mathcal{C}_k} (r_j^{(0)}) \]
\[ q_{kq}^{(1)} = k_q (p_r) \prod_{r \in \mathcal{C}_k} (r_j^{(1)}) \]

Make hard decision

\[ \hat{c}_k = \begin{cases} 1 & \text{if } Q_1 \geq 0.5 \\ 0 & \text{otherwise} \end{cases} \]
Halting Criteria

- After each iteration, halt if:
  \[ \hat{H}^T = 0 \]

- This is effective, because the probability of an undetectable decoding error is negligible

- Otherwise, halt once the maximum number of iterations is reached

- If the Tanner graph contains no cycles, then \( Q_i \) converges to the true APP value as the number of iterations tends to infinity

Sum-Product Algorithm in Log Domain

- The sum-product algorithm in probability domain has two shortcomings
  - Numerically unstable
  - Too many multiplications

- A log domain version is often used for practical purposes

- LLR of the \( i \)th code bit (ultimate goal of algorithm)
  \[ Q_i = \log \left( \frac{P(c_i = 0 | y, S)}{P(c_i = 1 | y, S)} \right) \]

- \( q_{ij} = \log \left( \frac{q_i(0)}{q_i(1)} \right) \text{extrinsic info to be passed from v-node } i \text{ to c-node } j \)

- \( r_{ji} = \log \left( \frac{r_j(0)}{r_j(1)} \right) \text{extrinsic info to be passed from c-node } j \text{ to v-node } i \)
Sum-Product Decoder
(in Log-Domain)

- Initialize:
  - \( q_{ij} = \lambda_i = 2y_i/\sigma^2 \) = channel LLR value
- Loop over all i,j for which \( h_{ij} = 1 \)
  - At each c-node, update the r messages:
    \[
    r_{ji} = \left( \prod_{j \in R_i} \alpha_{ij} \right) \phi \left( \sum_{j \in R_i} \beta_{ij} \phi(\beta_{ij}) \right)
    \]
  - At each v-node update the q message and Q LLR:
    \[
    Q_i = \hat{\lambda}_i + \sum_{j \in C_i} r_{ji}
    \]
    \[
    q_{ij} = Q_i - r_{ji}
    \]
  - Make hard decision:
    \[
    \hat{c}_i = \begin{cases} 
    1 & \text{if } Q_i < 0 \\
    0 & \text{otherwise}
    \end{cases}
    \]

Sum-Product Algorithm:
Notation

- \( \alpha_{ij} = \text{sign}(q_{ij}) \)
- \( \beta_{ij} = |q_{ij}| \)
- \( \phi(x) = -\log \tanh(x/2) = \log( (e^x+1)/(e^x-1) ) = \phi^{-1}(x) \)
Min-Sum Algorithm

- Note that:
  \[
  \phi\left(\sum_j\phi(\beta_j)\right) = \phi\left(\min\beta_j\right) = \min\beta_j
  \]
- So we can replace the r message update formula with
  \[
  r_{ji} = \prod_{i\in K_j} \alpha_{i,j} \min\beta_{r_j}
  \]
- This greatly reduces complexity, since now we don’t have to worry about computing the nonlinear \(\phi\) function.
- Note that since \(\alpha\) is just the sign of \(q\), \(\prod\alpha\) can be implemented by using XOR operations.

BER of Different Decoding Algorithms

- Code #1: MacKay’s construction 2A
- AWGN channel
- BPSK modulation
Extrinsic-information Scaling

- As with max-log-MAP decoding of turbo codes, min-sum decoding of LDPC codes produces an extrinsic information estimate which is biased.
  - In particular, $r_{ji}$ is overly optimistic.

- A significant performance improvement can be achieved by multiplying $r_{ji}$ by a constant $\kappa$, where $\kappa < 1$.
  $$ r_{ji} = \kappa \left( \prod_{i \in R_{ji}} \alpha_{i,j} \right) \min_{i \in R_{ji}} \beta_{i,j} $$


- Experimentation shows that $\kappa=0.9$ gives best performance.

BER of Different Decoding Algorithms

- Code #1: MacKay’s construction 2A
- AWGN channel
- BPSK modulation

- Min-sum
- Min-sum w/ extrinsic info scaling
  Scale factor $\kappa=0.9$
Regular vs. Irregular LDPC codes

- An LDPC code is **regular** if the rows and columns of \( H \) have uniform weight, i.e., all rows have the same number of ones \( (d_v) \) and all columns have the same number of ones \( (d_c) \)
  - The codes of Gallager and MacKay were regular (or as close as possible)
  - Although regular codes had impressive performance, they are still about 1 dB from capacity and generally perform worse than turbo codes
- An LDPC code is **irregular** if the rows and columns have non-uniform weight
  - Irregular LDPC codes tend to outperform turbo codes for block lengths of about \( n > 10^5 \)
- The degree distribution pair \( (\lambda, \rho) \) for a LDPC code is defined as
  \[
  \lambda_i(x) = \sum_{j=0}^\infty \lambda_{ji} x^i \\
  \rho_i(x) = \sum_{j=0}^\infty \rho_{ij} x^{i-1}
  \]
  - \( \lambda_i, \rho_i \) represent the fraction of edges emanating from variable (check) nodes of degree \( i \)

Constructing Regular LDPC Codes: MacKay, 1996

- Around 1996, Mackay and Neal described methods for constructing sparse \( H \) matrices
- The idea is to randomly generate a \( M \times N \) matrix \( H \) with weight \( d_v \) columns and weight \( d_c \) rows, subject to some constraints
- Construction 1A: Overlap between any two columns is no greater than 1
  - This avoids length 4 cycles
- Construction 2A: \( M/2 \) columns have \( d_v = 2 \), with no overlap between any pair of columns. Remaining columns have \( d_v = 3 \). As with 1A, the overlap between any two columns is no greater than 1
- Construction 1B and 2B: Obtained by deleting select columns from 1A and 2A
  - Can result in a higher rate code
Constructing Irregular LDPC Codes: Luby, et. al., 1998

- Luby et. al. (1998) developed LDPC codes based on irregular LDPC Tanner graphs
- Message and check nodes have conflicting requirements
  - Message nodes benefit from having a large degree
  - LDPC codes perform better with check nodes having low degrees
- Irregular LDPC codes help balance these competing requirements
  - High degree message nodes converge to the correct value quickly
  - This increases the quality of information passed to the check nodes, which in turn helps the lower degree message nodes to converge
- Check node degree kept as uniform as possible and variable node degree is non-uniform
  - Code 14: Check node degree =14, Variable node degree =5, 6, 21, 23
- No attempt made to optimize the degree distribution for a given code rate

Density Evolution: Richardson and Urbanke, 2001

- Given an irregular Tanner graph with a maximum $d_v$ and $d_c$, what is the best degree distribution?
  - How many of the v-nodes should be degree $d_v$, $d_v-1$, $d_v-2$, etc. nodes?
  - How many of the c-nodes should be degree $d_c$, $d_c-1$, etc. nodes?
- Question answered using Density Evolution
  - Process of tracking the evolution of the message distribution during belief propagation
- For any LDPC code, there is a “worst case” channel parameter called the threshold such that the message distribution during belief propagation evolves in such a way that the probability of error converges to zero as the number of iterations tends to infinity
- Density evolution is used to find the degree distribution pair ($\lambda$, $\rho$) that maximizes this threshold
Density Evolution: Richardson and Urbanke, 2001

- **Step 1:** Fix a maximum number of iterations
- **Step 2:** For an initial degree distribution, find the threshold
- **Step 3:** Apply a small change to the degree distribution
  - If the new threshold is larger, fix this as the current distribution
- **Repeat Steps 2-3**

Richardson and Urbanke identify a rate $\frac{1}{2}$ code with degree distribution pair which is 0.06 dB away from capacity

Chung et al., use density evolution to design a rate $\frac{1}{2}$ code which is 0.0045 dB away from capacity

More on Code Construction

- LDPC codes, especially irregular codes exhibit error floors at high SNRs
- The error floor is influenced by $d_{\text{min}}$
  - Directly designing codes for large $d_{\text{min}}$ is not computationally feasible
- Removing short cycles indirectly increases $d_{\text{min}}$ (*girth conditioning*)
  - Not all short cycles cause error floors
- **Trapping sets** and **Stopping sets** have a more direct influence on the error floor

Error floors can be mitigated by increasing the size of minimum stopping sets

Trapping sets can be mitigated using *averaged belief propagation decoding*

LDPC codes based on *projective geometry* reported to have very low error floors
A linear block code is encoded by performing the matrix multiplication $c = uG$.

A common method for finding $G$ from $H$ is to first make the code systematic by adding rows and exchanging columns to get the $H$ matrix in the form $H = [P^T I]$. Then $G = [I P]$. However, the result of the row reduction is a non-sparse $P$ matrix. The multiplication $c = [u uP]$ is therefore very complex.

As an example, for a (10000, 5000) code, $P$ is 5000 by 5000. Assuming the density of 1's in $P$ is 0.5, then $0.5 \times (5000)^2$ additions are required per codeword.

This is especially problematic since we are interested in large $n (>10^5)$.

An often used approach is to use the all-zero codeword in simulations.

Richardson and Urbanke show that even for large $n$, the encoding complexity can be (almost) linear function of $n$


Using only row and column permutations, $H$ is converted to an approximately lower triangular matrix.

- Since only permutations are used, $H$ is still sparse.
- The resulting encoding complexity is almost linear as a function of $n$.

An alternative involving a sparse-matrix multiply followed by differential encoding has been proposed by Ryan, Yang, & Li....

Encoding LDPC Codes

- Let \( H = [H_1 \ H_2] \) where \( H_1 \) is sparse and

\[
H_2 = \begin{bmatrix}
1 & 1 & 1 \\
1 & 1 & \\
1 & \ldots & 1 \\
1 & & 1
\end{bmatrix}
\]

and \( H_2^T = \begin{bmatrix}
1 & 1 & \ldots & 1 \\
1 & 1 & & 1 \\
\ldots & & & \\
& & & 1
\end{bmatrix} \)

- Then a systematic code can be generated with \( G = [I \ H_1 H_2^T] \).

- It turns out that \( H_2^T \) is the generator matrix for an accumulate-code (differential encoder), and thus the encoder structure is simply:

\[
\begin{align*}
u & \quad \text{Multiply by } H_2^T \\
\oplus & \\
D & \\
\end{align*}
\]

\[ uH_1 H_2^T \]

- Similar to Jin & McEliece’s Irregular Repeat Accumulate (IRA) codes.
- Thus termed "Extended IRA Codes".

Performance Comparison

- We now compare the performance of the maximum-length UMTS turbo code against four LDPC code designs.

- Code parameters
  - All codes are rate \( \frac{1}{3} \)
  - The LDPC codes are length \( (n,k) = (15000, 5000) \)
    - Up to 100 iterations of log-domain sum-product decoding
    - Code parameters are given on next slide
  - The turbo code has length \( (n,k) = (15354, 5114) \)
    - Up to 16 iterations of log-MAP decoding

- BPSK modulation
- AWGN and fully-interleaved Rayleigh fading

- Enough trials run to log 40 frame errors
  - Sometimes fewer trials were run for the last point (highest SNR).
LDPC Code Parameters

- **Code 1: MacKay’s regular construction 2A**

- **Code 2: Richardson & Urbanke irregular construction**

- **Code 3: Improved irregular construction**
  - Idea is to avoid small stopping sets

- **Code 4: Extended IRA code**

LDPC Degree Distributions

- The distribution of row-weights, or check-node degrees, is as follows:

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>10000</td>
<td>4999</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>13</td>
<td>5458</td>
<td>5000</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>9987</td>
<td>4542</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- The distribution of column-weights, or variable-node degrees, is:

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5000</td>
<td>8282</td>
<td>9045</td>
<td>9999</td>
</tr>
<tr>
<td>3</td>
<td>10000</td>
<td>2238</td>
<td>2267</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2584</td>
<td>569</td>
<td>5000</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>206</td>
<td>1941</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>1689</td>
<td>1178</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
DVB-S2 LDPC Code

- The digital video broadcasting (DVB) project was founded in 1993 by ETSI to standardize digital television services.
- The latest version of the standard DVB-S2 uses a concatenation of an outer BCH code and inner LDPC code.
- The codeword length can be either $n = 64800$ (normal frames) or $n = 16200$ (short frames).
- Normal frames support code rates $9/10$, $8/9$, $5/6$, $4/5$, $3/4$, $2/3$, $3/5$, $1/2$, $2/5$, $1/3$, $1/4$.
  - Short frames do not support rate $9/10$.
- DVB-S2 uses an extended-IRA type LDPC code.
FER for DVB-S2 LDPC Code
Normal Frames in BPSK/AWGN

FER for DVB-S2 LDPC Code
Short Frames in BPSK/AWGN
M-ary Complex Modulation

- \( \mu = \log_2 M \) bits are mapped to the symbol \( x_k \), which is chosen from the set \( S = \{ x_1, x_2, \ldots, x_M \} \)
  - The symbol is multidimensional.
  - 2-D Examples: QPSK, M-PSK, QAM, APSK, HEX
  - M-D Example: FSK, block space-time codes (BSTC)
- The signal \( y = hx_k + n \) is received
  - \( h \) is a complex fading coefficient.
  - More generally (BSTC), \( Y = HX + N \)
- Modulation implementation in the ISCML
  - The complex signal set \( S \) is created with the CreateConstellation function.
  - Modulation is performed using the Modulate function.

Log-likelihood of Received Symbols

- Let \( p(x_k|y) \) denote the probability that signal \( x_k \in S \) was transmitted given that \( y \) was received.
- Let \( f(x_k|y) = K \ p(x_k|y) \), where \( K \) is any multiplicative term that is constant for all \( x_k \).
- When all symbols are equally likely, \( f(x_k|y) \propto f(y|x_k) \)
- For each signal in \( S \), the receiver computes \( f(y|x_k) \)
  - This function depends on the modulation, channel, and receiver.
  - Implemented by the Demod2D and DemodFSK functions, which actually computes \( \log f(y|x_k) \).
- Assuming that all symbols are equally likely, the most likely symbol \( x_k \) is found by making a hard decision on \( f(y|x_k) \) or \( \log f(y|x_k) \).
Example: QAM over AWGN.

- Let \( y = x + n \), where \( n \) is complex i.i.d. \( N(0,N_0/2) \) and the average energy per symbol is \( E[|x|^2] = E_s \)

\[
p(y|x_k) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{|y-x_k|^2}{2\sigma^2}\right)
\]

\[
f(y|x_k) = \exp\left(-\frac{|y-x_k|^2}{2\sigma^2}\right)
\]

\[
\log f(y|x_k) = \frac{|y-x_k|^2}{2\sigma^2} -\frac{1}{N_o} E_s |y-x_k|^2
\]

Log-Likelihood of Symbol \( x_k \)

- The log-likelihood of symbol \( x_k \) is found by:

\[
\Lambda_k = \log p(x_k | y) = \log \frac{p(x_k | y)}{\sum_{x_k \in S} p(x_k | y)}
\]

\[
= \log f(y | x_k) - \log \sum_{x_m \in S} f(y | x_m)
\]

\[
= \log f(y | x_k) - \log \sum_{x_m \in S} \exp\{\log f(y | x_m)\}
\]

\[
= \log f(y | x_k) - \max_{x_m \in S}\{\log f(y | x_m)\}
\]
The max* function

\[
\text{max}^* (x, y) = \log[\exp(x) + \exp(y)] \\
= \max(x, y) + \log(1 + \exp(-|y - x|)) \\
= \max(x, y) + f_c(|y - x|) \\
\]

\[
f_c(z) = \log(1 + \exp(-z))
\]

Capacity of Coded Modulation (CM)

- Suppose we want to compute capacity of M-ary modulation
  - In each case, the input distribution is constrained, so there is no need to maximize over \( p(x) \)
  - The capacity is merely the mutual information between channel input and output.

- The mutual information can be measured as the following expectation:

\[
C = I(X; Y) = E_{y\sim\mathcal{X}} \left[ \log M + \log p(x_k | y) \right] \text{ nats}
\]
Monte Carlo Calculation of the Capacity of Coded Modulation (CM)

- The mutual information can be measured as the following expectation:
  \[ C = I(X;Y) = E_{X_1,a}[\log M + \log p(x_k | y)] \text{ nats} \]
  \[ = \log M + E_{X_1,a}[\Lambda_k] \text{ nats} \]
  \[ = \log_2 M + \frac{E_{X_1,a}[\Lambda_k]}{\log(2)} \text{ bits} \]
  \[ = \mu + \frac{E_{X_1,a}[\Lambda_k]}{\log(2)} \text{ bits} \]

- This expectation can be obtained through Monte Carlo simulation.

Simulation Block Diagram

- This function is computed by the CML function `Demod2D`
- This function is computed by the CML function `Capacity`

Benefits of Monte Carlo approach:
- Allows high dimensional signals to be studied.
- Can determine performance in fading.
- Can study influence of receiver design.
Capacity of 2-D modulation in AWGN

- 256QAM
- 64QAM
- 16QAM
- 16PSK
- 8PSK
- QPSK
- BPSK

Capacity of M-ary Noncoherent FSK in AWGN


Noncoherent combining penalty
BICM

- Coded modulation (CM) is required to attain the aforementioned capacity.
  - Channel coding and modulation handled jointly.
  - e.g. trellis coded modulation (Ungerboeck); coset codes (Forney)
- Most off-the-shelf capacity approaching codes are binary.
- A pragmatic system would use a binary code followed by a bitwise interleaver and an M-ary modulator.
  - Bit Interleaved Coded Modulation (BICM); Caire 1998.
Transforming Symbol Log-Likehoods Into Bit LLRs

- Like the CM receiver, the BICM receiver calculates $\log f(y|x_k)$ for each signal in $S$.
- Furthermore, the BICM receiver needs to calculate the log-likelihood ratio of each code bit:

$$
\lambda_n = \log \frac{p(c_n = 1 | y)}{p(c_n = 0 | y)} = \log \frac{\sum_{x_k \in S_n^{(1)}} p(x_k | y)}{\sum_{x_k \in S_n^{(0)}} p(x_k | y)} = \log \frac{\sum_{x_k \in S_n^{(1)}} p(y | x_k) p[x_k]}{\sum_{x_k \in S_n^{(0)}} p(y | x_k) p[x_k]}
$$

- where $S_n^{(1)}$ represents the set of symbols whose $n$th bit is a 1.
- and $S_n^{(0)}$ is the set of symbols whose $n$th bit is a 0.

BICM Capacity

- BICM transforms the channel into $\mu$ parallel binary channels, and the capacity of the $n$th channel is:

$$
C_n = E_{c_n} \left[ \log(2) + \log p(c_n = 1 | y) \right] \text{ nats}
$$

$$
= \log(2) + E_{c_n} \left[ \log \frac{p(c_n = 1 | y)}{p(c_n = 0 | y) + p(c_n = 1 | y)} \right] \text{ nats}
$$

$$
= \log(2) - E_{c_n} \left[ \log \frac{p(c_n = 0 | y) + p(c_n = 1 | y)}{p(c_n | y)} \right] \text{ nats}
$$

$$
= \log(2) - E_{c_n} \left[ \max \left\{ \log \frac{p(c_n = 0 | y)}{p(c_n | y)}, \log \frac{p(c_n = 1 | y)}{p(c_n | y)} \right\} \right] \text{ nats}
$$

$$
= \log(2) - E_{c_n} \left[ \max \left\{ \log \frac{p(c_n = 0 | y)}{p(c_n | y)}, \log \frac{p(c_n = 1 | y)}{p(c_n | y)} \right\} \right] \text{ nats}
$$
BICM Capacity (Continued)

- Since capacity over parallel channels adds, the capacity of BICM is:

\[
C = \sum_{k=1}^{\mu} C_k = \sum_{k=1}^{\mu} \left[ \log(2) - E_{\epsilon_k, \alpha} \left[ \max \{ 0, (x_k)^c \lambda_k \} \right] \right] \text{ nats}
\]

\[
= \mu \log(2) - \sum_{k=1}^{\mu} E_{\epsilon_k, \alpha} \left[ \max \{ 0, (x_k)^c \lambda_k \} \right] \text{ nats}
\]

\[
= \mu - \frac{1}{\log(2)} \sum_{k=1}^{\mu} E_{\epsilon_k, \alpha} \left[ \max \{ 0, (x_k)^c \lambda_k \} \right] \text{ bits}
\]

BICM Capacity

- As with CM, this can be computed using a Monte Carlo integration.

Modulator:
Pick \( x_k \) at random from \( S \)

Receiver:
Compute \( p(y|x_k) \) for every \( x_k \in S \)

For each bit, calculate:

\[
\lambda_k = \log \left( \sum_{x_k \in S} p(y|x_k) \right) / \sum_{x_k \in S} p(y|x_k)
\]

For each symbol, calculate:

\[
\Lambda = \sum_{k=1}^{\mu} E_{\epsilon_k, \alpha} \left[ \max \{ 0, (x_k)^c \lambda_k \} \right] \text{ bits}
\]

Unlike CM, the capacity of BICM depends on how bits are mapped to symbols

This function is computed by the CML function \( \text{Somap} \)

After running many trials, calculate:

\[
C = \mu + \frac{E[\Lambda]}{\log(2)}
\]
The conventional BICM receiver assumes that all bits in a symbol are equally likely:

\[ \lambda_n = \log \frac{\sum_{x \in S_n^{(y)}} p(x | y)}{\sum_{x \in S_n^{(y)}} p(x)} = \log \frac{\sum_{x \in S_n^{(y)}} p(y | x)}{\sum_{x \in S_n^{(y)}} p(y)} \]

However, if the receiver has estimates of the bit probabilities, it can use this to weight the symbol likelihoods.

\[ \lambda_n = \log \frac{\sum_{x \in S_n^{(y)}} p(y | x)p(x | c_n = 1)}{\sum_{x \in S_n^{(y)}} p(y | x)p(x | c_n = 0)} \]

This information is obtained from decoder feedback.

- Bit Interleaved Coded Modulation with Iterative Demodulation
- Li and Ritcey 1999.
Now consider a receiver that has a priori information about the code bits (from a soft output decoder).

Assume the following:
- The a priori information is in LLR form.
- The a priori LLR’s are Gaussian distributed.
- The LLR’s have mutual information \( I_v \)

Then the mutual information \( I_z \) at the output of the receiver can be measured through Monte Carlo Integration.
- \( I_z \) vs. \( I_v \) is the **Mutual Information Transfer Characteristic**.
- ten Brink 1999.

---

**Generating Random a Priori Input**

There is a one-to-one correspondence between the mutual information and the variance of the Gaussian distributed a priori input.

![Graph showing the relationship between mutual information and variance](attachment://graph.jpg)
Mutual Information Characteristic

EXIT Chart

adding curve for a FEC code makes this an extrinsic information transfer (EXIT) chart
EXIT Chart Analysis of Turbo Codes

- PCCC (turbo) codes can be analyzed with an EXIT chart by plotting the mutual information transfer characteristics of the two decoders.
Conclusions

- It is now possible to closely approach the Shannon limit by using turbo and LDPC codes.
- Binary capacity approaching codes can be combined with higher order modulation using the BICM principle.
- These codes are making their way into standards
  - Binary turbo: UMTS, cdma2000
  - Duobinary turbo: DVB-RCS, 802.16
  - LDPC: DVB-S2 standard.
- Software for simulating turbo and LDPC codes can be found at www.iterativesolutions.com