Iterative Solutions
Coded Modulation Library
Theory of Operation

Oct. 3, 2005

Matthew Valenti
Associate Professor
West Virginia University
Morgantown, WV 26506-6109
mvalenti@wvu.edu

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.
Contents

- Shannon capacity under modulation constraints.
- Bit interleaved coded modulation.
- Iterative demodulation and decoding
  - EXIT charts.
- Turbo coding.
  - The SISO decoding algorithm.
- LDPC codes
  - The message passing algorithm.
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:

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

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

This capacity is achieved by a Gaussian input \( x \).
- This is not a practical modulation.
If we only consider antipodal (BPSK) modulation, then

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

and the capacity is:

\[
C = \max_{p(x)} \{ I(X; Y) \} \quad \text{maximized when two signals are equally likely}
\]

\[
= I(X; Y) \bigg|_{p(x): p=1/2}
\]

\[
= H(Y) - H(N)
\]

\[
= \int_{-\infty}^{\infty} p(y) \log_2 p(y) dy - \frac{1}{2} \log_2 (\pi e N_o)
\]

This term must be integrated numerically with

\[
p_Y(y) = p_X(y) * p_N(y) = \int_{-\infty}^{\infty} p_X(\lambda) p_N(y - \lambda) d\lambda
\]
Capacity of AWGN w/ 1-D Signaling

It is theoretically possible to operate in this region.

It is theoretically impossible to operate in this region.
Power Efficiency of Standard Binary Channel Codes

Spectral Efficiency vs. Code Rate $r$

Eb/No in dB

 arbitraily low BER: $P_b = 10^{-5}$
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
  - Currently only 2-D modulation is supported.
  - The complex signal set $S$ is created with the Create2D function.
  - Modulation is performed using the Mod2D 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 \texttt{Demod2D} function, 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. \( \mathcal{N}(0, N_0/2) \) and the average energy per symbol is \( E[|x|^2] = E_s \)

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

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

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

\[
= -\frac{E_s |y - x_k|^2}{N_o}
\]
Log-Likelihood of Symbol $x_k$

The log-likelihood of symbol $x_k$ is found by:

$$\Lambda_k = \log p(x_k | y)$$

$$= \log \frac{p(x_k | y)}{\sum_{x_m \in S} p(x_k | y)}$$

$$= \log \frac{f(y | x_k)}{\sum_{x_m \in S} f(y | x_m)}$$

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

$$= \log f(y | x_k) - \log \exp\{\log f(y | x_m)\}$$

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

- The pairwise max* function is defined as:

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

\[
= \max(x, y) + \log\left(1 + \exp\left(-|y - x|\right)\right)
\]

\[
= \max(x, y) + f_c(|y - x|)
\]

- Where \(f_c\) is a correction function that only depends on the magnitude of the difference between \(x\) and \(y\).

- For multiple arguments, it follows the recursive relationship:

\[
\text{max}^*(x, y, z) = \text{max}^*\left[\text{max}^*(x, y), z\right]
\]
The correction function

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

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

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

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

\[
C = I(X; Y) = E_{x_k, n} \left[ \log M + \log p(x_k | y) \right] \text{ nats}
= \log M + E_{x_k, n} \left[ \Lambda_k \right] \text{ nats}
= \log_2 M + \frac{E_{x_k, n} \left[ \Lambda_k \right]}{\log(2)} \text{ bits}
= \mu + \frac{E_{x_k, n} \left[ \Lambda_k \right]}{\log(2)} \text{ bits}
\]

- This expectation can be obtained through Monte Carlo simulation.
This function is computed by the ISCML function **Demod2D**

This function is computed by the ISCML function **Capacity**

---

**Modulator:**
Pick $x_k$ at random from $S$

**Noise Generator:**

$N_k$

**Receiver:**
Compute $\log f(y|x_k)$ for every $x_k \in S$

Calculate:

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

After running many trials, calculate:

$$C = \mu + \frac{E[\Lambda_k]}{\log(2)}$$

**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 2-D modulation in AWGN

Eb/No in dB
Capacity of M-ary Noncoherent FSK in AWGN


![Graph showing noncoherent combining penalty for different M values: M=2, M=4, M=16, M=64.](image)
Ergodic Capacity (Fully interleaved)
Assumes perfect fading amplitude estimates available to receiver

Capacity of M-ary Noncoherent FSK in Rayleigh Fading

Minimum Eb/No (in dB)

Rate R (symbol per channel use)
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-Likelihoods 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:

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

$$= \max_{x_k \in S_n^{(1)}} \log[f(y \mid x_k)] - \max_{x_k \in S_n^{(0)}} \log[f(y \mid x_k)]$$

- 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:

$$C_k = E_{c_n,n}[\log(2) + \log p(c_k | y)] \text{ nats}$$

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

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

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

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

$$= \log(2) - E_{c_n,n} \left[ \max^* \left\{ 0, (-1)^{c_k} \lambda_k \right\} \right] \text{ nats}$$
Since capacity over parallel channels adds, the capacity of BICM is:

\[
C = \sum_{k=1}^{\mu} C_k
\]

\[
= \sum_{k=1}^{\mu} \left\{ \log(2) - E_{c_n,n} \left[ \max* \left\{ 0, (-1)^c_k \lambda_k \right\} \right] \right\} \text{ nats}
\]

\[
= \mu \log(2) - \sum_{k=1}^{\mu} E_{c_n,n} \left[ \max* \left\{ 0, (-1)^c_k \lambda_k \right\} \right] \text{ nats}
\]

\[
= \mu - \frac{1}{\log(2)} \sum_{k=1}^{\mu} E_{c_n,n} \left[ \max* \left\{ 0, (-1)^c_k \lambda_k \right\} \right] \text{ bits}
\]
BICM Capacity

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

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

For each bit, calculate:

\[
\Lambda = -\sum_{k=1}^{\mu} E_{c_n,n} \left[ \max^* \left\{ 0, (-1)^{c_k} \lambda_k \right\} \right] \text{ bits}
\]

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

This function is computed by the ISCML function Somap

After running many trials, calculate:

\[
C = \mu + \frac{E[\Lambda]}{\log(2)}
\]
CM and BICM capacity for 16QAM in AWGN

- CM M=16 QAM AWGN
- BICM M=16 QAM gray
- BICM M=16 QAM SP
- BICM M=16 QAM MSP
- BICM M=16 QAM Antigray
- BICM M=16 QAM MSEW
The conventional BICM receiver assumes that all bits in a symbol are equally likely:

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

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

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

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

- 16-QAM
- AWGN
- 6.8 dB

Legend:
- gray
- SP
- MSP
- MSEW
- Antigray
EXIT Chart

16-QAM
AWGN
6.8 dB
adding curve for a FEC code makes this an extrinsic information transfer (EXIT) chart
EXIT Chart for Space Time Block Code

16-QAM
8 dB
Rayleigh fading
BICM-ID for Noncoherent FSK

Review: Convolutional Codes

- A *convolutional encoder* encodes a *stream* of data.
  - The size of the code word is unbounded.

- The encoder is a *Finite Impulse Response* (FIR) filter.
  - k binary inputs
  - n binary outputs
  - K -1 delay elements
  - Coefficients are either 1 or 0
  - All operations over GF(2)
    - Addition: XOR

Constraint Length $K = 3$
Recursive Systematic Convolutional (RSC) Encoding

- 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.

In the ISCML, convolutional encoding is implemented with the `ConvEncode` function.

The optional input `nsc_flag` determines if it is an RSC (flag = 0) or NSC (flag = 1) encoder. The default is for it to be an RSC encoder.
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:
\[ d_{\text{min}} = 18 \]
\[ c_{d_{\text{min}}} = 187 \]
\[ P_b \approx (187) Q\left( \sqrt{\frac{E_b}{N_o}} \right) \]

Turbo code:
\[ d_{\text{min}} = 6 \]
\[ \tilde{c}_{d_{\text{min}}} = \frac{a_{d_{\text{min}}} \tilde{w}_{d_{\text{min}}}}{k} = \frac{3 \cdot 2}{65536} \]
\[ P_b \approx \left(9.2 \times 10^{-5}\right) Q\left( \sqrt{\frac{6}{N_o}} \right) \]
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 of 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)

\[ K = 5 \]
\[ \text{Rate } r = 1/2 \]
\[ 18 \text{ decoder iterations} \]
\[ \text{AWGN Channel} \]
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:
  - $K_{CC} \approx 2 + K_{TC} + \log_2(\text{number decoder iterations})$
UMTS Turbo Encoder

- From ETSI TS 125 212 v3.4.0 (2000-09)
  - 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.
  - $8 \leq C \leq 256$
  - If $L < RC$ then matrix is padded with dummy characters.

In the ISCML, 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.

\[
\begin{array}{cccccccc}
X_2 & X_6 & X_5 & X_7 & X_3 & X_4 & X_1 & X_8 \\
X_{10} & X_{12} & X_{11} & X_{15} & X_{13} & X_{14} & X_9 & X_{16} \\
X_{18} & X_{22} & X_{21} & X_{23} & X_{19} & X_{20} & X_{17} & X_{24} \\
X_{26} & X_{28} & X_{27} & X_{31} & X_{29} & X_{30} & X_{25} & X_{32} \\
X_{40} & X_{36} & X_{35} & X_{39} & X_{37} & X_{38} & X_{33} & X_{34}
\end{array}
\]
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.

\[
\begin{array}{cccccccc}
X_{40} & X_{36} & X_{35} & X_{39} & X_{37} & X_{38} & X_{33} & X_{34} \\
X_{26} & X_{28} & X_{27} & X_{31} & X_{29} & X_{30} & X_{25} & X_{32} \\
X_{18} & X_{22} & X_{21} & X_{23} & X_{19} & X_{20} & X_{17} & X_{24} \\
X_{10} & X_{12} & X_{11} & X_{15} & X_{13} & X_{14} & X_{9} & X_{16} \\
X_{2} & X_{6} & X_{5} & X_{7} & X_{3} & X_{4} & X_{1} & X_{8}
\end{array}
\]
UMTS Interleaver: Reading Data From Matrix

- Data is read from matrix column-wise.

Thus:

- $X'_1 = X_{40}$  $X'_2 = X_{26}$  $X'_3 = X_{18}$  ...
- $X'_38 = X_{24}$  $X'_2 = X_{16}$  $X'_40 = X_8$
Recursive Systematic Convolutional (RSC) Encoder

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

- After the $L^{th}$ 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 steam is:

\[ X_1 Z_1 Z'_1 X_2 Z_2 Z'_2 \ldots X_L Z_L Z'_L X_{L+1} Z_{L+1} X_{L+2} Z_{L+2} X_{L+3} Z_{L+3} X'_{L+1} Z'_{L+1} X'_{L+2} Z'_{L+2} X'_{L+3} Z'_{L+3} \]

- 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: \[ r = \frac{L}{3L+12} \approx \frac{1}{3} \]
Channel Model and LLRs

- Channel gain: $a$
  - Rayleigh random variable if Rayleigh fading
  - $a = 1$ if AWGN channel

- Noise
  - Variance is:
    $$\sigma^2 = \frac{1}{2r\left(\frac{E_b}{N_o}\right)} \approx \frac{3}{2\left(\frac{E_b}{N_o}\right)}$$
SISO-MAP Decoding Block

This block is implemented in the ISCML by the SisoDecode function.

\[ \lambda^{u,i} \rightarrow \text{SISO MAP Decoder} \rightarrow \lambda^{u,o} \]

\[ \lambda^{c,i} \rightarrow \text{SISO MAP Decoder} \rightarrow \lambda^{c,o} \]

**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.
Initialization and timing:
- Upper $\lambda^{u,i}$ input is initialized to all zeros.
- Upper decoder executes first, then lower decoder.
Performance as a Function of Number of Iterations

BER of 640 bit turbo code in AWGN

- L=640 bits
- AWGN channel
- 10 iterations
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 revisited

- **max** must implement the following operation:
  
  \[
  z = \max(x, y) + \ln(1 + \exp(-|y - x|))
  \]
  
  \[
  = \max(x, y) + f_c(|y - x|)
  \]
  
  \[
  = \text{max}^*(x, y)
  \]

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

$$\gamma_{ij} = X_k(i, j)\lambda_k^{u,i} + X_k(i, j)\lambda_k^{c,i} + Z_k(i, j)\lambda_k^{c,i}$$

- 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:

\[
\alpha_j = \max^* \left\{ \left( \alpha'_{i_1} + \gamma_{i,j} \right), \left( \alpha'_{i_2} + \gamma_{i,j} \right) \right\}
\]

- 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:

\[
\beta_i = \max^* \left\{ (\beta'_{j_1} + \gamma_{ij_1}), (\beta'_{j_2} + \gamma_{ij_2}) \right\}
\]

- 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:
  \[ \alpha_i + \gamma_{ij} + \beta_j \]

- 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:
  \[ \Lambda(X_k) = \ln \left( \frac{P[X_k = 1]}{P[X_k = 0]} \right) \]
  \[ \quad = \max^{\ast}_{S_i \rightarrow S_j : X_k = 1} \left\{ \alpha_i + \gamma_{ij} + \beta_j \right\} \]
  \[ \quad - \max^{\ast}_{S_i \rightarrow S_j : X_k = 0} \left\{ \alpha_i + \gamma_{ij} + \beta_j \right\} \]
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

BER vs. Eb/No in dB

10 decoder iterations

Fading

AWGN
Practical Implementation

Issues

- No need to store both alpha and beta in trellis.
  - Compute beta first and store in trellis.
  - Then compute alpha and derive LLR estimates at the same time.
    - No need to store alpha trellis.

- The metrics keep getting larger and larger.
  - Floating point: loss of precision.
  - Fixed point: overflow.
  - Solution: normalize the metrics:

\[
(\alpha_i)_{\text{normalized}} = \alpha_i - \min_i(\alpha_i) \quad \text{or} \quad (\alpha_i)_{\text{normalized}} = \alpha_i - \alpha_0
\]

\[
(\beta_i)_{\text{normalized}} = \beta_i - \min_i(\beta_i) \quad \text{or} \quad (\beta_i)_{\text{normalized}} = \beta_i - \beta_0
\]
Issues: Channel Estimation

- In an AWGN Channel:
  - log-MAP and constant-log-MAP need an estimate of $\sigma^2$
  - Estimate of $\sigma^2$ is not needed for max-log-MAP.
  - Constant-log-MAP is more vulnerable to bad estimates than log-MAP.

- In addition, for fading channels:
  - All three algorithms require estimates of the channel gain $a$. 
Issues: Memory Limitations

- Storing the entire beta trellis can require a significant amount of memory:
  - 8L states
  - For L=5114 there will have to be 40,912 stored values.
  - The values only need to be single-precision floats.
- An alternative is to use a “sliding window” approach.
Issues: Halting the Iterations

- Turbo decoding progresses until a fixed number of iterations have completed.
  - However, the decoder will often converge early.
  - Can stop once it has converged (i.e. BER = 0).
  - Stopping early can greatly increase the throughput.

- For a simulation, it is reasonable to automatically halt once the BER goes to zero.
  - Requires knowledge of the data.

- For an actual system, a “blind” method for halting is desirable.
  - Blind --- don’t know the data.
Simple Method for Halting

A simple method is as follows:

- After each decoder iteration, determine the smallest output LLR value:

\[
\Lambda_{\text{min}} = \min_{1 \leq k \leq K} \{|\Lambda(X_k)|\}
\]

- Then compare this value to a threshold.
- If the minimum LLR is more than the threshold, then halt:
  - Halt if: \( \Lambda_{\text{min}} > llr_{\text{halt}} \)
BER of Different Values of llr_halt in AWGN channel

BER of 640 bit turbo code

- llr_halt = 1
- llr_halt = 5
- llr_halt = 10
- llr_halt = 0
FER of Different Values of $llr_{\text{halt}}$ in AWGN channel

FER of 640 bit turbo code

$llr_{\text{halt}} = 1$

$llr_{\text{halt}} = 5$

$llr_{\text{halt}} = 10$

$llr_{\text{halt}} = 0$

EB/No in dB
Average Number of Iterations

Number of iterations for 640 bit turbo code

- $llr_{\text{halt}} = 1$
- $llr_{\text{halt}} = 5$
- $llr_{\text{halt}} = 10$
- $llr_{\text{halt}} = 0$

Eb/No in dB vs Number of decoder iterations
LDPC Code Overview

- Low Density Parity Check (LDPC) codes are a class of linear block codes characterized by sparse parity check matrices $H$.

- Review of parity check matrices:
  - For a $(n,k)$ code, $H$ is a $n-k$ by $n$ matrix of ones and zeros.
  - A codeword $c$ is valid iff $cH^T = 0$
  - 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.
  - In an LDPC code, only a few bits (~4 to 6) participate in each parity check equation.

- 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.
Decoding LDPC Codes

- Like turbo codes, LDPC codes are iteratively decoded.
- Instead of a trellis, LDPC codes use a “Tanner graph”
  - A Tanner graph consists of **variable-nodes** (corresponding to the received code bits) and **check-nodes** (corresponding to the parity check equations).
  - The H matrix determines the connections between v-nodes and c-nodes.
  - Messages iteratively flow back and forth between v-nodes and c-nodes.
- In general, the per-iteration complexity of LDPC codes is less than it is for turbo codes.
  - E.g. the best LDPC code investigated here needs half as many additions per iteration than the UMTS turbo code.
- However, many more iterations may be required (max≈100; avg≈30).
  - Thus, overall complexity can be higher than turbo.
- More on the decoding algorithm and complexity later.
Detecting Errors

- With LDPC codes, error detection comes for free.
- Errors can be detected by computing $cH^T = 0$
  - The probability of undetected error is very small.
- This can be used to dynamically halt the iterations.
- No outer error detecting code is needed.
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 and all columns have the same number of ones.
  - 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.
  - Typically the rows have similar weight (e.g. 5 and 6) while the columns have vastly different weight (e.g. from 2 to 15 or 20).
  - Around 2000, Richardson and Urbanke showed that long irregular LDPC codes could come within ~0.1 dB of capacity.
    - One of their $r=1/2$ designs (with Chung) came within 0.04 dB of capacity.
  - Irregular LDPC codes tend to outperform turbo codes for block lengths of about $n>10^5$
A linear block code is encoded by performing the matrix multiplication $c = uG$, where $u$ is the $k$-bit long message and $G$ is a $k$ by $n$ generator matrix.

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 $c = uG = [u \ uP]$ is now very complex, as the calculation of $uP$ is $O(k(n-k))$

Solutions:

- Richardson and Urbanke (2001) show how to transform the $H$ matrix into a (fairly) sparse (almost) lower triangular matrix. Then encoding complexity is approximately $O(k)$.
- An alternative involving a sparse-matrix multiply followed by differential encoding has been proposed by Ryan, Yang, & Li....
Extended IRA Codes  
(Ryan, Yang, & Li)

- Let $H = [H_1 \ H_2]$ where $H_1$ is sparse and
  
  $H_2 = \begin{bmatrix} 
  1 \\
  1 \\
  1 \\
  1 \\
  \end{bmatrix}$
  
  and $H_2^{-T} = \begin{bmatrix} 
  1 & 1 & \ldots & 1 \\
  1 & 1 & \ldots & 1 \\
  \vdots & \vdots & \ddots & \vdots \\
  1 & 1 & \ldots & 1 \\
  \end{bmatrix}$

- Then a systematic code can be generated with $G = [I \ H_1^T H_2^{-T}]$.
- It turns out that $H_2^{-T}$ is the generator matrix for an accumulate-code (differential encoder), and thus the encoder structure is simply:

  \[
  \begin{array}{c}
  u \\
  \rightarrow \quad \text{Multiply} \\
  \text{by } H_1^T \\
  + \rightarrow \quad \text{D} \\
  \downarrow \rightarrow \quad uH_1^T H_2^{-T} \\
  \end{array}
  \]

- 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:

  \[
  \begin{array}{c|cccc}
  i & 1 & 2 & 3 & 4 \\
  \hline
  3 &  &  &  & 1 \\
  4 & 10000 & & 4999 \\
  5 & & 13 & 5458 & 5000 \\
  6 & & 9987 & 4542 & \\
  \end{array}
  \]

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

  \[
  \begin{array}{c|cccc}
  i & 1 & 2 & 3 & 4 \\
  \hline
  1 & &  &  & 1 \\
  2 & 5000 & 8282 & 9045 & 9999 \\
  3 & 10000 & 2238 & 2267 & \\
  4 & & 2584 & 569 & 5000 \\
  5 & & 206 & 1941 & \\
  8 & & & 1 & \\
  15 & & 1689 & 1178 & \\
  \end{array}
  \]

Code number:
1 = MacKay construction 2A
2 = Richardson & Urbanke
3 = Jones, Wesel, & Tian
4 = Ryan’s Extended-IRA
BER in AWGN

BPSK/AWGN Capacity:
-0.50 dB for r = 1/3
FER in AWGN

![Graph showing FER vs. Eb/No in dB for different codes]

- Code #1: Mackay 2A
- Code #2: R&U
- Code #3: JWT
- Code #4: IRA

**EB/No in dB**

**FER (Error Floor Rate)**
Average Number of Iterations in AWGN

- Code #1: Mackay 2A
- Code #2: R&U
- Code #3: JWT
- Code #4: IRA

The graph shows the average number of iterations required for different codes as a function of Eb/No in dB.
BER in Fully-Interleaved Rayleigh Fading

BPSK/Fading Capacity:
0.5 dB for $r = 1/3$

turbo

Code #1: Mackay 2A
Code #2: JWT
Code #3: R&U
Code #4: IRA
FER in Fully-Interleaved Rayleigh Fading

Eb/No in dB

10^{-4} 10^{-3} 10^{-2} 10^{-1} 10^{0} 10^{1} 10^{2} 10^{3}

Code #1: Mackay 2A
Code #2: JWT
Code #3: R&U
Code #4: IRA

FER

turbo
Average Number of Iterations
Rayleigh Fading

![Graph showing average number of iterations for different codes under Rayleigh fading conditions. The x-axis represents Eb/No in dB, and the y-axis represents the average number of iterations. Codes include Mackay 2A, IRA, R&U, JWT, and Turbo.]
Tanner Graphs

- 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

\[ H = \begin{bmatrix}
1 & 1 & 1 & 0 & 1 & 0 & 0 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 \\
1 & 0 & 1 & 1 & 0 & 0 & 1 \\
\end{bmatrix} \]
Message Passing Decoding: Notation

- $Q_i =$ LLR of the $i^{th}$ code bit (ultimate goal of algorithm)
- $q_{ij} =$ extrinsic info to be passed from v-node $i$ to c-node $j$
- $r_{ji} =$ extrinsic info to be passed from c-node $j$ to v-node $i$
- $C_i = \{j : h_{ji} = 1\}$
  - This is the set of row location of the 1’s in the $i^{th}$ column.
- $R_j = \{i : h_{ji} = 1\}$
  - This is the set of column location of the 1’s in the $j^{th}$ row
- $R_{ji} = \{i' : h_{ji'} = 1\}\{i\}$
  - The set of column locations of the 1’s in the $j^{th}$ row, excluding location $i$. 
More Notation

- \( \alpha_{ij} = \text{sign}(q_{ij}) \)
- \( \beta_{ij} = |q_{ij}| \)
- \( \phi(x) = -\log \tanh(x/2) = \log\left(\frac{e^x+1}{e^x-1}\right) = \phi^{-1}(x) \)
Sum-Product Decoder
(in Log-Domain)

- Initialize:
  - $q_{ij} = \lambda_i = 2\text{Re}\{a_i^*r_i\}/\sigma^2 = \text{channel LLR value}$

- Loop over all $i,j$ for which $h_{ij} = 1$
  - At each c-node, update the $r$ messages:
    $$r_{ji} = \left( \prod_{i' \in R_{ji}} \alpha_{i'j} \right) \phi \left( \sum_{i' \in R_{ji}} \phi \left( \beta_{i'j} \right) \right)$$
  - At each v-node update the $q$ message and $Q$ LLR:
    $$Q_i = \lambda_i + \sum_{j \in C_i} r_{ji}$$
    $$q_{ij} = Q_i - r_{ji}$$
  - Make hard decision:
    $$\hat{c}_i = \begin{cases} 1 & \text{if } Q_i < 0 \\ 0 & \text{otherwise} \end{cases}$$
Halting Criteria

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

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

- Otherwise, halt once the maximum number of iterations is reached.
Min-Sum Decoder
(Log-Domain)

- Note that:
  \[
  \phi\left(\sum_{i'} \phi(\beta_{i',j})\right) \approx \phi(\max_{i'} \phi(\beta_{i',j})) = \phi\left(\min_{i'} \beta_{i',j}\right) = \min_{i'} \beta_{i',j}
  \]

- So we can replace the r message update formula with
  \[
  r_{ji} = \left(\prod_{i' \in R_{ji}} \alpha_{i',j}\right) \min_{i' \in R_{ji}} \beta_{i',j}
  \]

- 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.
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 $\kappa < 1$.

$$ r_{ji} = \kappa \left( \prod_{i' \in R_{ji}} \alpha_{i'j} \right) \min_{i' \in R_{ji}} \beta_{i'j} $$


- Here we compare the performance of sum-product, min-sum, and min-sum with extrinsic info scaling.
  - 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
w/ extrinsic info scaling
Scale factor $\kappa=0.9$

Sum-product

EB/No in dB

BER
FER of
Different Decoding Algorithms

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

Min-sum

Min-sum
w/ extrinsic info scaling
Scale factor $\kappa = 0.9$

Sum-product

FER

$10^0$
$10^{-1}$
$10^{-2}$
$10^{-3}$
$10^{-4}$

Eb/No in dB

$0$
$0.2$
$0.4$
$0.6$
$0.8$
$1$
$1.2$
$1.4$
$1.6$
$1.8$
Consider the complexity of an efficiently implemented max-log-MAP decoding of the UMTS turbo code.


**Forward Recursion.**
- For each stage of the trellis, compute $2^2 = 4$ branch metrics.
  - This only requires 1 addition**
- Compute the pair of partial path metrics entering each node
  - This requires 12 additions**
- Select from among the two branches entering each node.
  - This requires 8 pairwise comparisons.
- Normalization
  - Requires 7 subtractions** (subtraction has same complexity as addition)
- Thus the forward recursion requires (where k is # info bits):
  - 20k additions.
  - 8k comparisons.
Complexity of Turbo Decoding

- **Backward Recursion**
  - Complexity is identical to forward recursion

- **LLR Calculation:**
  - Likelihood of one: Compare 8 branches*
  - Likelihood of zero: Compare 7 branches*
  - So, 13 pairwise comparisons per trellis section.
  - Subtract the two likelihoods
  - Thus LLR Calculation requires:
    - k additions
    - 13k comparisons

- **Extra overhead:**
  - Need to subtract input from output LLR to get extrinsic info, and need to add extrinsic info to channel LLR.
  - Amounts to an extra 3k additions / iteration.**
Summary: Complexity of Turbo Decoding

- Since each SISO decoder is executed twice per iteration, the per iteration decoding complexity of the UMTS turbo code is:
  - 44k additions / iteration
  - 39k pairwise comparisons / iteration
  - where k is the number of data bits

- If log-MAP decoding is used instead of max-log-MAP, then the 39k pairwise comparisons are replaced with max* operators.

- This neglects the extra overhead of detecting the data bits, implementing an adaptive stopping criterion, or scaling the extrinsic information.
Complexity of Min-Sum Decoding of LDPC Codes

- Let $m_i$ be the number of rows of $H$ with weight $i$.
- Let $n_i$ be the number of columns of $H$ with weight $i$.
- Update $r$:
  \[
  r_{ji} = \left( \prod_{i' \in R_{ji}} \alpha_{i', j} \right) \min_{i' \in R_{ji}} \beta_{i', j}
  \]
  - For a $c$-node of degree $i$, there will be $i(i-2)$ xors and $i(i-2)$ pairwise comparisons.
  - Thus overall complexity of the update $r$ stage is:
    \[
    \sum_i m_i i(i-2) \text{ xors and } \sum_i m_i i(i-2) \text{ pairwise comparisons}
    \]
- Update $Q$:
  \[
  Q_i = \lambda_i + \sum_{j \in C_i} r_{ji}
  \]
  - For $v$-node of degree $i$, this requires $i$ additions.
  - So overall complexity of the update $Q$ stage is: $\sum_i n_i i$ additions
Complexity of Min-Sum Decoding of LDPC Codes

- Update q:
  \[ q_{ij} = Q_i - r_{ji} \]
  
  - This is a single subtraction at every branch leaving a v-node, and thus the complexity is also:
  \[ \sum_i n_i \text{ additions} \]

- And so the per iteration complexity of min-sum decoding of LDPC codes is:
  \[ \sum_i m_i(i - 2) \text{ xors} \]
  \[ \sum_i m_i(i - 2) \text{ pairwise comparisons} \]
  \[ 2 \sum_i n_i \text{ additions} \]

- This neglects the extra overhead of detecting the data bits, checking for errors, or scaling the extrinsic information.
Complexity of Sum-Product Decoding

If sum-product decoding is used, then

\[ r_{ji} = \left( \prod_{i' \in R_{j/i}} \alpha_{i'j} \right) \phi \left( \sum_{i' \in R_{j/i}} \phi(\beta_{i'j}) \right) \]

- Thus every comparison operation is replaced with an addition.
  - Complexity can be slightly reduced by first taking the sum of all \( \phi(\beta) \) and then subtracting off the particular \( \beta_i' \).
- Also, there will now be two nonlinear \( \phi \) function calls per iteration for every edge in the graph.
  - One for every \( \beta \) and another for every \( r \).
Complexity Comparison

Here we compare the complexity of the (15000, 4996) turbo code with the four (15000,5000) LDPC codes.
- Per iteration complexity.
- Overall, assuming 5 turbo iterations and 30 LDPC iterations.

<table>
<thead>
<tr>
<th>Code</th>
<th>Per iteration complexity</th>
<th>Overall Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>additions</td>
<td>comparisons*</td>
</tr>
<tr>
<td>Turbo</td>
<td>219,824</td>
<td>194,844</td>
</tr>
<tr>
<td>LDPC #1 (MacKay 2A)</td>
<td>80,000</td>
<td>80,000</td>
</tr>
<tr>
<td>LDPC #2 (R&amp;U)</td>
<td>119,974</td>
<td>239,883</td>
</tr>
<tr>
<td>LDPC #3 (JWT)</td>
<td>109,084</td>
<td>190,878</td>
</tr>
<tr>
<td>LDPC #4 (extended IRA)</td>
<td>79,998</td>
<td>114,995</td>
</tr>
</tbody>
</table>

* For LDPC codes, one XOR is also required for every pairwise comparison.