# WiMAX IEEE 802.16 LDPC Code Performance Evaluation

Channel Coding Final Project

Tommaso Zugno

tommasozugno@gmail.com

12/01/2018

# Outline

# WiMAX Technology

WiMAX (Worldwide Interoperability for Microwave Access) is *a standards-based technology enabling the delivery of last mile wireless broadband access as an alternative to cable and DSL* [1] .

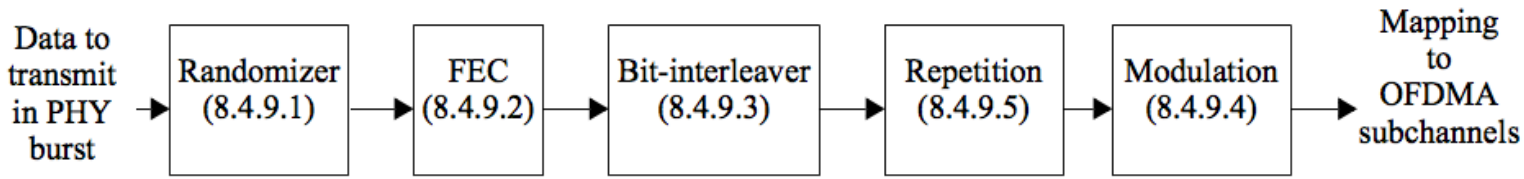
WiMAX is based upon std. IEEE 802.16e-2005 which provides multiple PHY and MAC options.



[1] www.WiMAXforum.org/technology/

# Channel coding in WirelessMAN-OFDMA PHY layer

Channel coding procedure:




The standard specifies different FEC options:

- Convolutional Coding (CC)
- Convolutional Turbo Coding (CTC)
- Block Turbo Coding
- **Low Density Parity Check codes**

# LDPC codes

The standard specifies a LDPC code for each combination of rate $R$ and codeword length $n$. There are:

- 6 possible rates (1/2, 2/3A, 2/3B, 3/4A, 3/4B, 5/6)
- 19 possible codeword lengths

We implemented in Matlab the encoder/decoder couple for some codes. In particular we considered:

- 4 different rates (1/2, 2/3B, 3/4A, 5/6)
- 3 different codeword lengths (576, 1344, 2304 bits)

# LDPC codes

Each code in the set of LDPC codes is specified by a parity check matrix $H$ of size $m \times n$.
$H$ is defined as:

$$H = \begin{bmatrix} P_{0,0} & P_{0,1} & \cdots & P_{0,n_b-1} \\ P_{1,0} & P_{1,1} & \cdots & P_{1,n_b-1} \\ \vdots & \vdots & & \vdots \\ P_{m_b-1,0} & P_{m_b-1,1} & \cdots & P_{m_b-1,n_b-1} \end{bmatrix}$$

where $P_{i,j}$ is a $z \times z$ permutation matrices or a $z \times z$ zero matrix with $z = \frac{n}{n_b} = \frac{m}{m_b}$ and $n_b = 24$.
The permutations used are circular right shifts.

# LDPC codes

$H$ is expanded from a base model matrix $H_{bm}$ of size $m_b \times n_b$:

- each positive entry $p(i,j)$ is replaced by a $P_{i,j}$ matrix with circular shift size equal to $p(i,j)$
- each blank or negative entry p(i,j) is replaced by a $P_{i,j}$ zero matrix

The standard defines $H_{bm}$ for the largest codeword length ($n = 2304$) of each code rate. $H_{bm}$ for a different codeword length $n'$ is obtained by scaling the shift sizes proportionally

$$p'(i,j) = \begin{cases} p(i,j) & p(i,j) \leq 0 \\ \left\lfloor \frac{p(i,j) z_f}{z_0} \right\rfloor & p(i,j) > 0 \end{cases}$$

where $z_0 = \frac{2304}{n_b}$ and $z_f = \frac{n'}{n_b}$.

# Encoding procedure

For an efficient encoding, H is divided into the form:

$$H = \begin{bmatrix} A_{(m-z)\times Rn} & B_{(m-z)\times z} & T_{(m-z)\times(m-z)} \\ C_{z\times z} & D_{z\times k} & E_{z\times(m-z)} \end{bmatrix}$$

We define: $M_1 = ET^{-1}A + C$, $M_2 = T^{-1}A$, $M_3 = T^{-1}B$.

Given the infoword $u$, we compute:

- $p_1^T = M_1 u$
- $p_2^T = M_2 u + M_3 p_1^T$

The corresponding codeword is $v = [u, p_1, p_2]$

# Encoder implementation

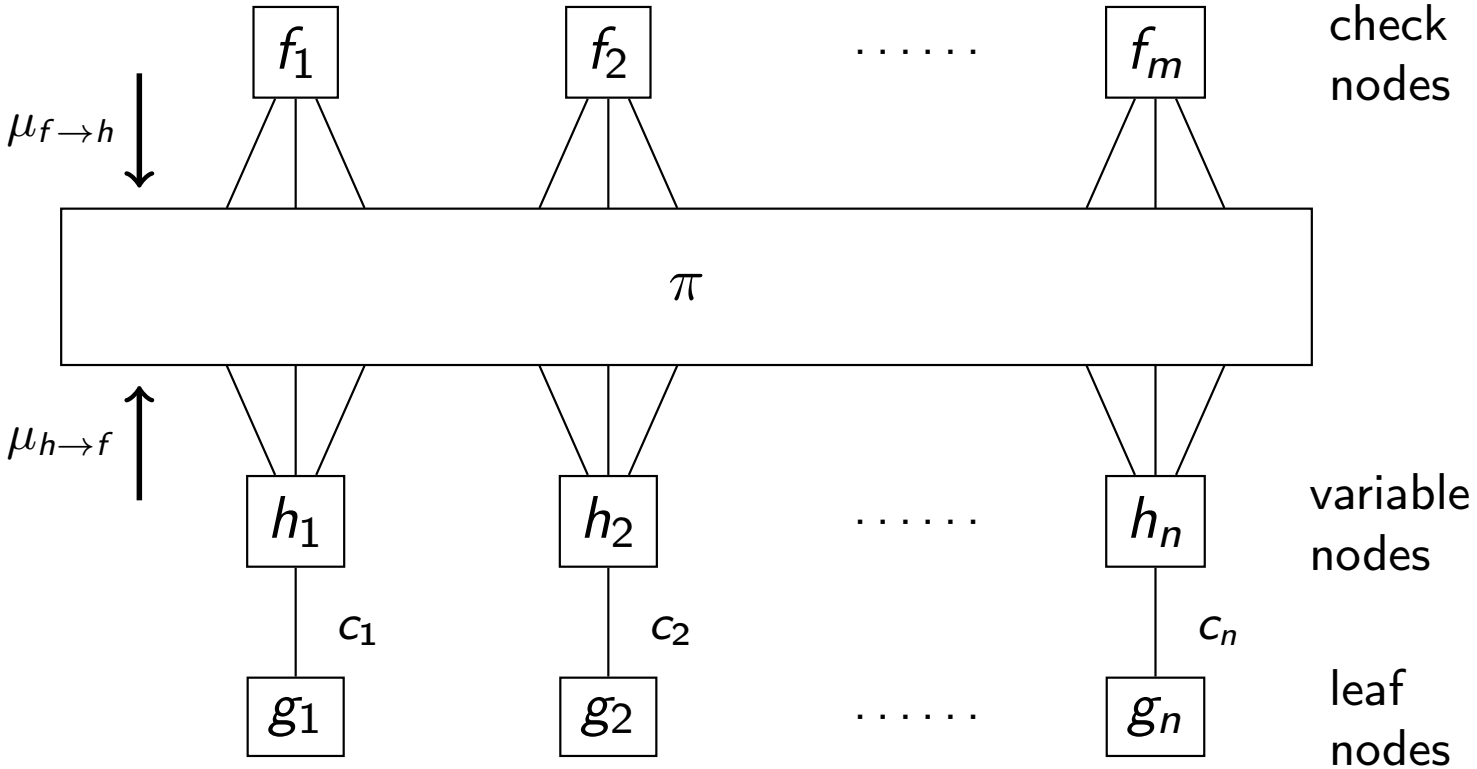
Given the chosen values for the rate $R$ and codeword length $n$, the script `generate-matrices2.m`:

- generates the matrix $H$ using the corresponding base matrix $H_{bm}$
- computes and saves the matrices $M_1$, $M_2$ and $M_3$

At the begin of each simulation, the matrices $M_1$, $M_2$ and $M_3$ are loaded and used during the encoding procedure.

# Decoding binary LDPC

The decoding is performed using the Message Passing Algorithm (MPA) in the logarithmic domain. The MPA is based on this factor graph:



# Message Passing Algorithm

**Initialization:** Messages $\mu_{h \to f}$ are initialized with the channel a-priori knowledge $g$.

**Schedule:**

1. run message passing on check nodes
2. run message passing on variable nodes
3. perform ongoing marginalization:

$$\hat{c}_l = \begin{cases} 0 & , \mu_{h_l \to c_l} + \mu_{g_l \to c_l} \geq 0 \\ 1 & , otherwise \end{cases}$$

4. go back to step 1 until the maximum number of iterations is reached or a codeword is found

## Leaf nodes

Under the hypothesis of equally probable input symbols, the messages from leaf to variable nodes are:

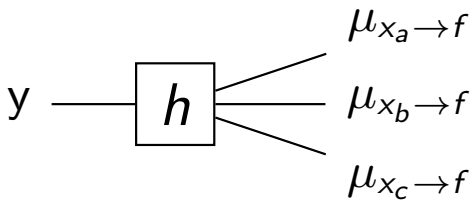
$$g_l = -\frac{2r_l}{\sigma_w^2}$$

where $r_l$ is the l-th received symbol and $\sigma_w^2$ is the noise variance.

## Variable nodes

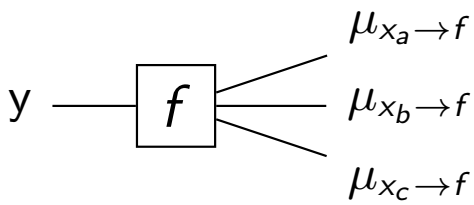
The update rule for the variable nodes is:

$$\mu_{h \to y} = \sum_i \mu_{x_i \to h}$$



## Check nodes

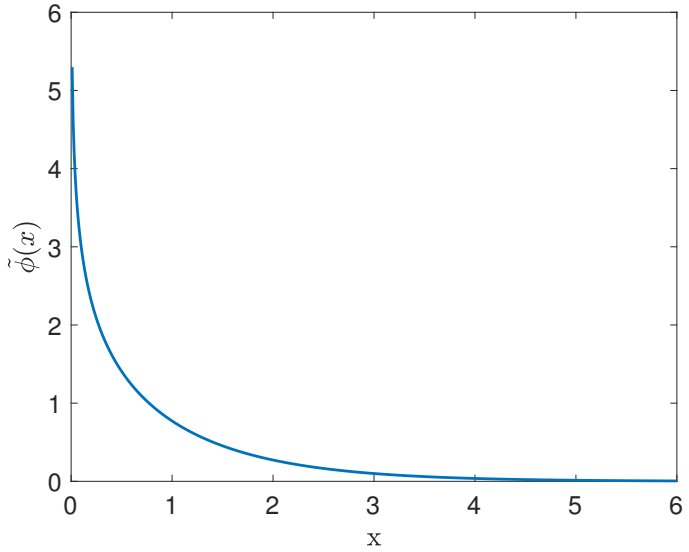
The update rule for the check nodes is:




$$\mu_{f \to y} = \tilde{\phi}\left( \sum_i \tilde{\phi}\left(|\mu_{x_i \to f}|\right) \right) \prod_i sign(\mu_{x_i \to f})$$

where:

$$\tilde{\phi} = -\log\left( tanh\left( \frac{1}{2} x \right) \right) = \tilde{\phi}^{-1}$$



# Decoder implementation

The decoder is implemented by the function `decode2.m` which outputs the estimate $\widehat{\mathbf{u}}$ of the transmitted infoword $\mathbf{u}$, given the received vector $\mathbf{r}$ and the noise variance $\sigma_w^2$.

- the vector `g` contains the messages exiting the leaf nodes and going upwards
- the matrix `mu_hf` contains the messages variable $\rightarrow$ check
- the matrix `mu_fh` contains the messages check $\rightarrow$ variable
- the vector `mu_hg` contains the messages exiting the leaf nodes and going downwards

# Decoder implementation

`decode2.m` implements the Message Passing schedule described above:

**Initialization:** compute g and initialize `mu_hf`
**while** max n of iterations is not reached AND codeword is not found

- update `mu_fh`

$$\tilde{\phi}\left(|\texttt{mu\_hf}|\right) = \begin{bmatrix} \tilde{\phi}(|\mu_{h_1 \to f_1}|) & \tilde{\phi}(|\mu_{h_1 \to f_2}|) & \dots & \tilde{\phi}(|\mu_{h_1 \to f_m}|) \\ \vdots & \vdots & & \vdots \\ \tilde{\phi}(|\mu_{h_n \to f_1}|) & \tilde{\phi}(|\mu_{h_n \to f_2}|) & \dots & \tilde{\phi}(|\mu_{h_n \to f_m}|) \end{bmatrix}$$

$$\texttt{tmp3} = \left( \begin{bmatrix} \sum_1^n \tilde{\phi}(|\mu_{h_i \to f_1}|) \\ \vdots \\ \sum_1^n \tilde{\phi}(|\mu_{h_i \to f_m}|) \end{bmatrix} \begin{bmatrix} 1 & \dots & 1 \end{bmatrix} \right) .* H - \tilde{\phi}(|\texttt{mu\_hf}|)^T$$

# Decoder implementation

$$sign\left(\texttt{mu\_hf}\right) = \begin{bmatrix} sign(\mu_{h_1 \to f_1}) & sign(\mu_{h_1 \to f_2}) & \dots & sign(\mu_{h_1 \to f_m}) \\ \vdots & \vdots & & \vdots \\ sign(\mu_{h_n \to f_1}) & sign(\mu_{h_n \to f_2}) & \dots & sign(\mu_{h_n \to f_m}) \end{bmatrix}$$

$$\texttt{mu\_fh} = \tilde{\phi}(\texttt{tmp3}).* \left( \begin{bmatrix} \prod_1^n sign(\mu_{h_i \to f_1}) \\ \vdots \\ \prod_1^n sign(\mu_{h_i \to f_m}) \end{bmatrix} \begin{bmatrix} 1 & \dots & 1 \end{bmatrix} \right) .* sign(\texttt{mu\_hf})^T$$

# Decoder implementation

- Update mu_hf and mu_hg

$$\texttt{mu\_hf} = \left[ \left( \begin{bmatrix} \sum_1^m \mu_{f_i \to h_1} \\ \vdots \\ \sum_1^m \mu_{f_i \to h_n} \end{bmatrix} + \texttt{g} \right) \begin{bmatrix} 1 & \dots & 1 \end{bmatrix} \right] .* H^T - \texttt{mu\_fh}^T$$

$$\texttt{mu\_hg} = \begin{bmatrix} \sum_1^m \mu_{f_i \to h_1} & \sum_1^m \mu_{f_i \to h_2} & \cdots & \sum_1^m \mu_{f_i \to h_n} \end{bmatrix}$$

- marginalize

$$\hat{c} = (\texttt{mu\_hg} + \texttt{g}) < 0$$

# Decoder implementation

The $\tilde{\phi}$ function is truncated to avoid over/underflow errors:

$$\tilde{\phi}(x) = \begin{cases} 12.5 & x < 10^{-5} \\ -\log\left(tanh\left(\frac{1}{2}x\right)\right) & 10^{-5} \leq x < 50 \\ 0 & x \geq 50 \end{cases}$$

# Decoding BICM

The BICM decoder was implemented by adding the conform nodes at the bottom of the LDPC factor graph.
The bit-interleaving operation was not implemented.

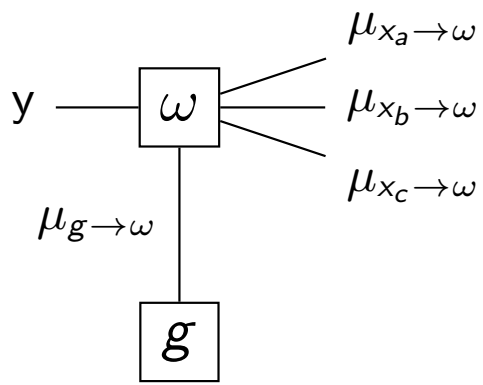
The Message Passing schedule modifies into:

- **Initialization:** compute the messages leaf $\rightarrow$ conform
- **Schedule:**
  1. run message passing on the conform nodes
  2. update messages variable $\rightarrow$ check
  3. run message passing on the check nodes
  4. update messages variable $\rightarrow$ conform
  5. marginalize
  6. go back to step 1 until the maximum number of iterations is reached or a codeword is found

# Decoding BICM

The update rule for the conform nodes is:

$$\mu_{\omega \to y}(y) = \log \left( \frac{\sum_{x_a, x_b, \ldots} g(map(0, x_a, x_b, \ldots)) e^{(1 \oplus x_a)\mu_{x_a \to \omega} + (1 \oplus x_b)\mu_{x_b \to \omega} + \ldots}}{\sum_{x_a, x_b, \ldots} g(map(1, x_a, x_b, \ldots)) e^{(1 \oplus x_a)\mu_{x_a \to \omega} + (1 \oplus x_b)\mu_{x_b \to \omega} + \ldots}} \right)$$



# BICM decoder implementation

The BICM decoder is implemented by the function `decodeBICM.m` which outputs the estimate $\widehat{\mathbf{u}}$ of the transmitted infoword $\mathbf{u}$ given the received vector $\mathbf{r}$ and the noise variance $\sigma_w^2$.

The BICM decoder was implemented starting from `decode2.m`. The initialization and schedule was modified as described above to include the contribute of the conform nodes.

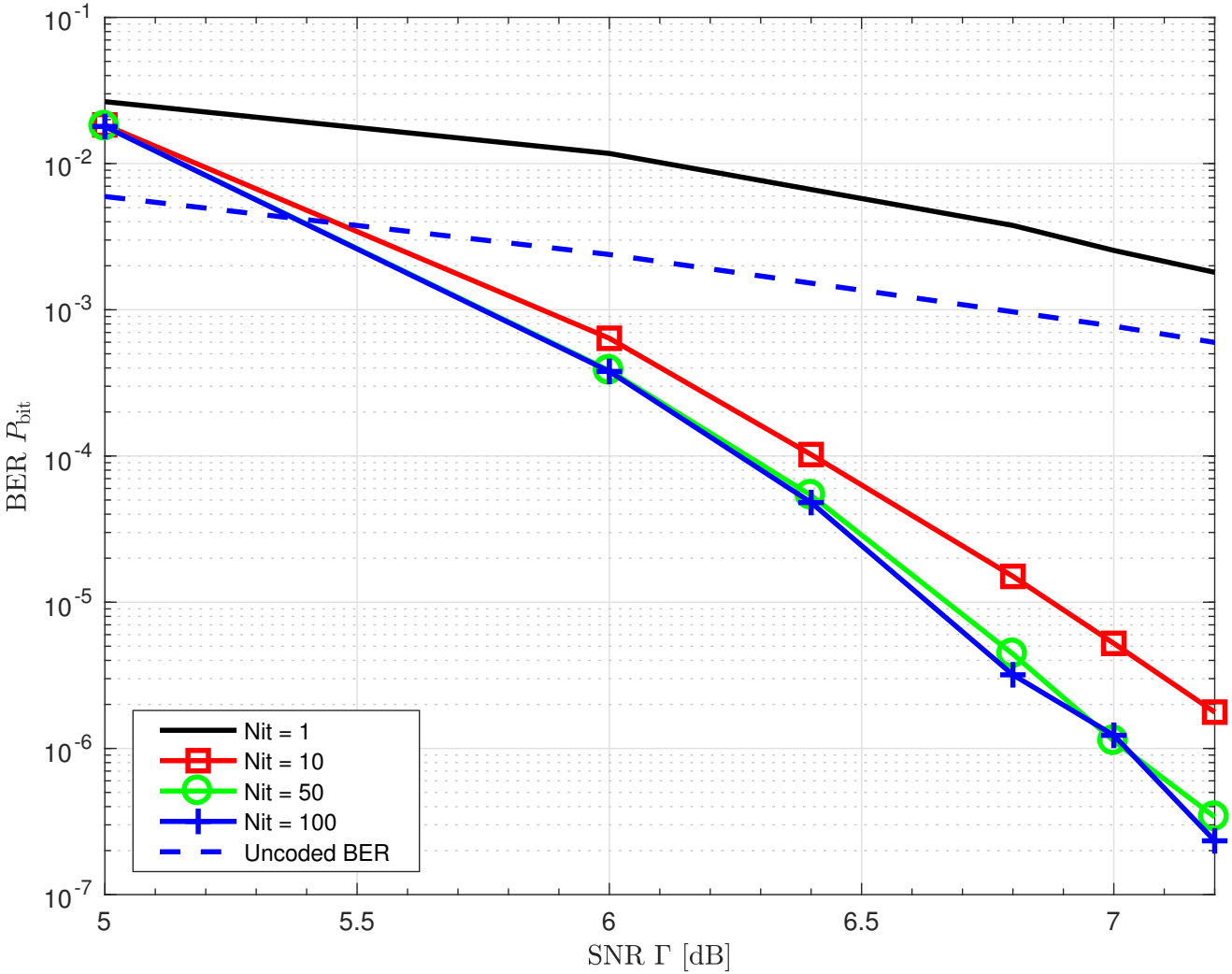
# main.m

The script `main.m` runs a simulation campaign to test the performance of a code in terms of $P_{bit}$ vs $SNR$.
For each $SNR$ value to test, the simulation follows this schedule:

1. generate a random infoword $\mathbf{u}$
2. encode $\mathbf{u}$ to obtain $\mathbf{c}$
3. modulate $\mathbf{c}$ to obtain $\mathbf{c_{mod}}$
4. add AWGN noise with variance $\sigma_w^2$ to obtain $\mathbf{r}$
5. decode $\mathbf{r}$ to obtain $\widehat{\mathbf{u}}$
6. compute the number of errors comparing $\mathbf{u}$ and $\widehat{\mathbf{u}}$
7. go back to step 1 until the total number of errors or the number of transmitted packets reach the threshold

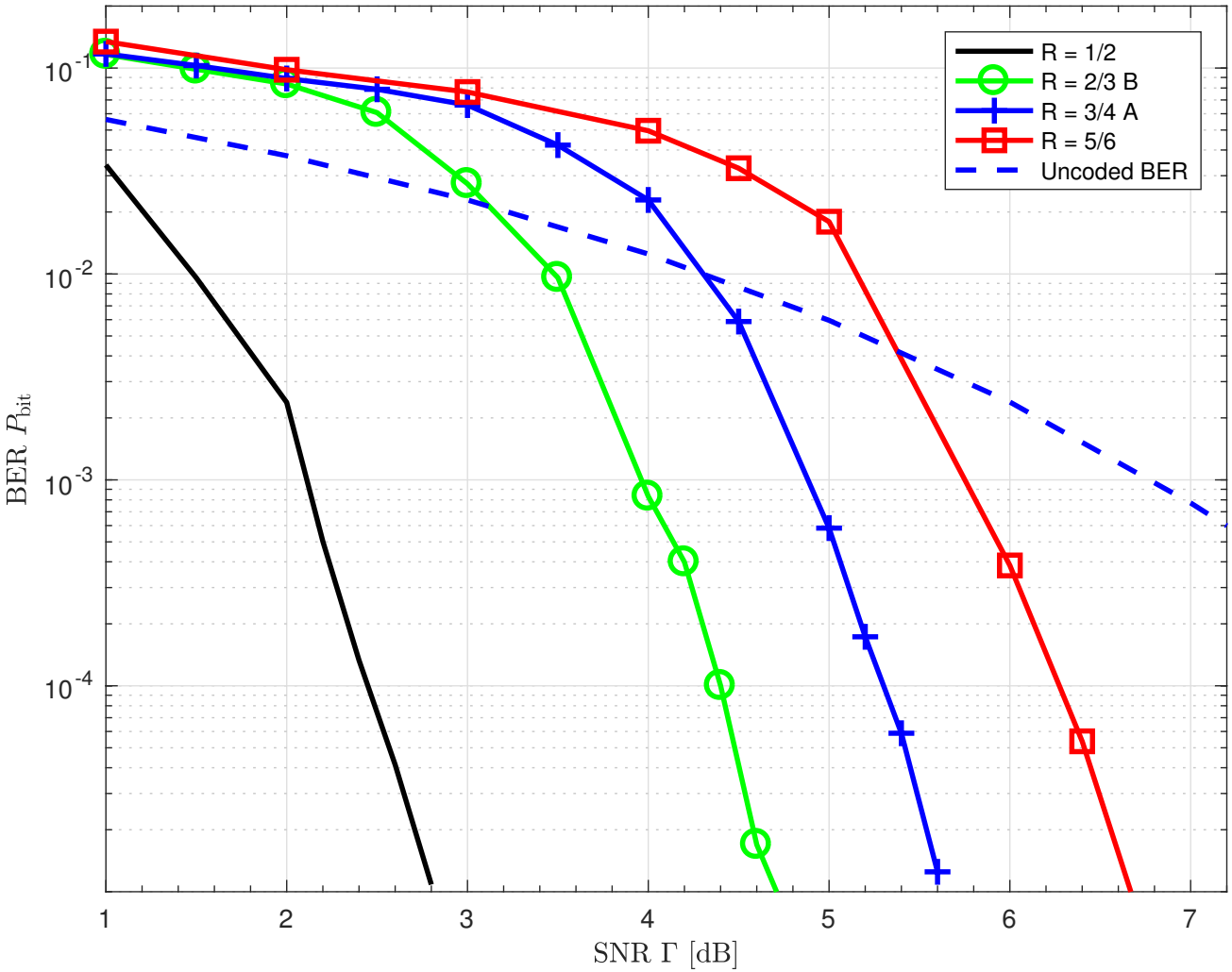
# Results

Performance of binary LDPC ($n = 576$, $R = 5/6$) for different number of iterations.



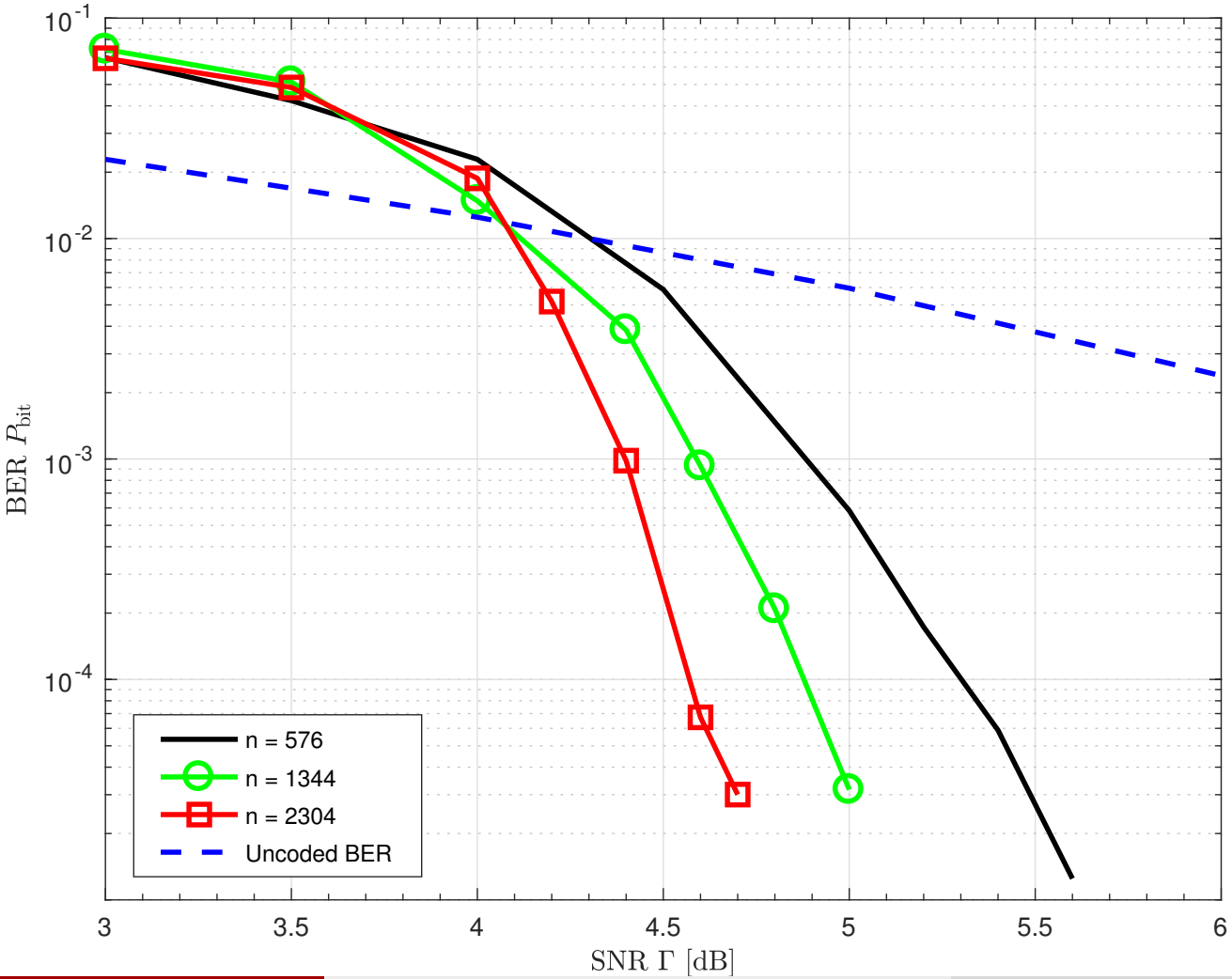
# Results

Performance of binary LDPC ($n = 576$, $N_{it} = 50$) for different rates.



# Results

Performance of binary LDPC ($R = 3/4A$, $N_{it} = 50$) for different codeword lengths.



# Results

Performance of BICM - LDPC ($n = 576$, $R = 5/6$, $N_{it} = 50$) for different modulation schemes.