# 5G LDPC 译码算法详解和仿真

刘杰 hahaliu2001@gmail.com

## 1 简介

LDPC code 用于 5G PDSCH 和 PUSCH 信道

项目实现了三种 LDPC 译码算法

1. Bit Flipping decoding algorithm
   硬判决，低复杂度，性能差
2. Belief propagation algorithm
   也称作 sum-product algorithm.
   软判决，性能近最优，高复杂度，主要用于仿真
3. Min-Sum algorithm
   软判决，性能次优，复杂度低。当前产品实现基本都使用这个算法
   项目实现了四种 Min-sum algorithms:
   - Traditional Min-sum
   - Normalized Min-sum
   - Offset Min-sum
   - Mixed Min-sum,四种 min-sum 中性能最优

本文讲解这三种算法。

The python 实现代码：

https://github.com/hahaliu2001/python_5gtoolbox.git：py5gphy/ldpc

## 2 LDPC 码

LDPC 编码矩阵: $H \times \begin{bmatrix} c \\ w \end{bmatrix} = H \times v_n = 0$

其中：

H 是 MxN 校验矩阵 matrix, c 是输入信息 bit，w 是生成的校验 bit.

LDPC 编码就是根据输入信息 bit 和校验矩阵，生成校验 bit

下面文章解释 5G LDPC encoder 优化:

https://github.com/hahaliu2001/python_5gtoolbox.git :
docs/algorithm/LDPC_encoder_optimization.pdf

LDPC 译码的基本思想就是：

根据输入 LLR 信息，估计 $v_n$ vector，使其满足 $H \times v_n = 0$

# 3 Bit-flipping Decoding Algorithm

## 3.1 Reference

[1] 基于可靠性调度的 LDPC 码比特翻转译码算法

https://www.jsjkx.com/EN/Y2019/V46/I6A/329

[2] Two-Round Selection-Based Bit Flipping Decoding Algorithm for LDPC Codes

https://onlinelibrary.wiley.com/doi/10.1155/2023/6262929

[3] Multi-Stage Bit-Flipping Decoding Algorithms for LDPC Codes

https://www.researchgate.net/publication/333913937_Multi-Stage_Bit-Flipping_Decoding_Algorithms_for_LDPC_Codes

## 3.2 实现

这是 LDPC 硬判决算法

对于 LDPC code (N,K),

其中:

N 是 LDPC 码长，K 是信息 bit 长度，M=N-K 校验 bit 长度

H is M X N 校验矩阵

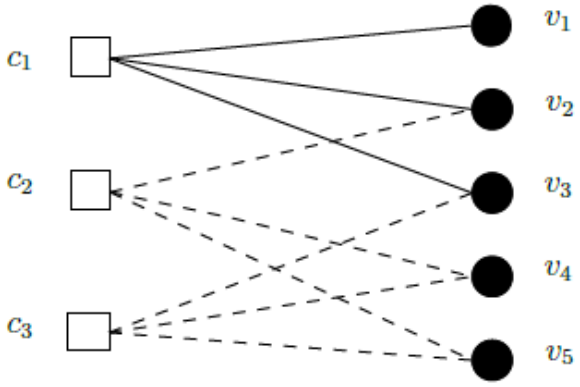
LDPC 译码是计算 $H \times v_n = s$，当 $s = 0$时译码成功

其中:

$v_n$称作 “variable nodes” with length=N

$s$ 称作 “check nodes”, with length=M

Tanner 图一般用于显示 variable nodes 和 check nodes 关系




$$H = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

H 矩阵每一行代表哪些 variable nodes 用于计算当前 check node

H 矩阵每一列代表哪些 check node 跟当前 variable node 关联

例如上图中,

第 0 行 [1,1,1,0,0]表示 variable nodes[0,1,2]用于计算 check node 0

第 1 列[1,1,0]表示 check nodes [0,1]关联到 variable node 1

**BF 译码过程：**

输入：N 个接收的 LLR 值

Step 1: 生成硬判决序列 $C_n$ from LLR, with $C_n$=0 if LLR >0 else 1

Step 2: 计算 $s = HC_n$

Step 3: 如果 s==0 -> 译码成功，退出译码过程

Step 4:计算 $C_n$的可靠性：$E_n = (2s-1)H$

$E_n$越高代表可靠性越低，也就是说这个 bit 更可能出错

Step 5: bit 翻转最大$E_n$值对应的 $C_n$ ，返回 step 2

**备注 1:**

一些 paper 描述可靠性 $E_n = sH$, 不是 $E_n = (2s-1)H$.

$E_n = sH$ 用于 regular LDPC(每一行‘1’的个数相同，每一列‘1’的个数相同，每行‘1’的个数和每列‘1’的个数可以不同)

5G 使用 irregular LDPC, $E_n = sH$ 是错误的。

**备注 2:**

一些 paper shows 说 Weighted Bit-Flipping Algorithm 性能比 non- Weighted Bit-Flipping Algorithm 更优，但是我测试时 Weighted Bit-Flipping Algorithm 不 work. BLER 在高 SNR 时仍然为 100%

Weighted Bit-Flipping Algorithm is:

$$e_k^{(2)} = \sum_{j \in M(k)} (2s_j - 1).r_{\min}^j,$$

其中 $r_{min}^j$ 是用于计算 jth 校验 bit 的所有 variable bits 中 LLR 最小绝对值

**备注 3:**

BF 译码性能比软判决性能差很多，比 BP 算法差 2.25dB

Python code

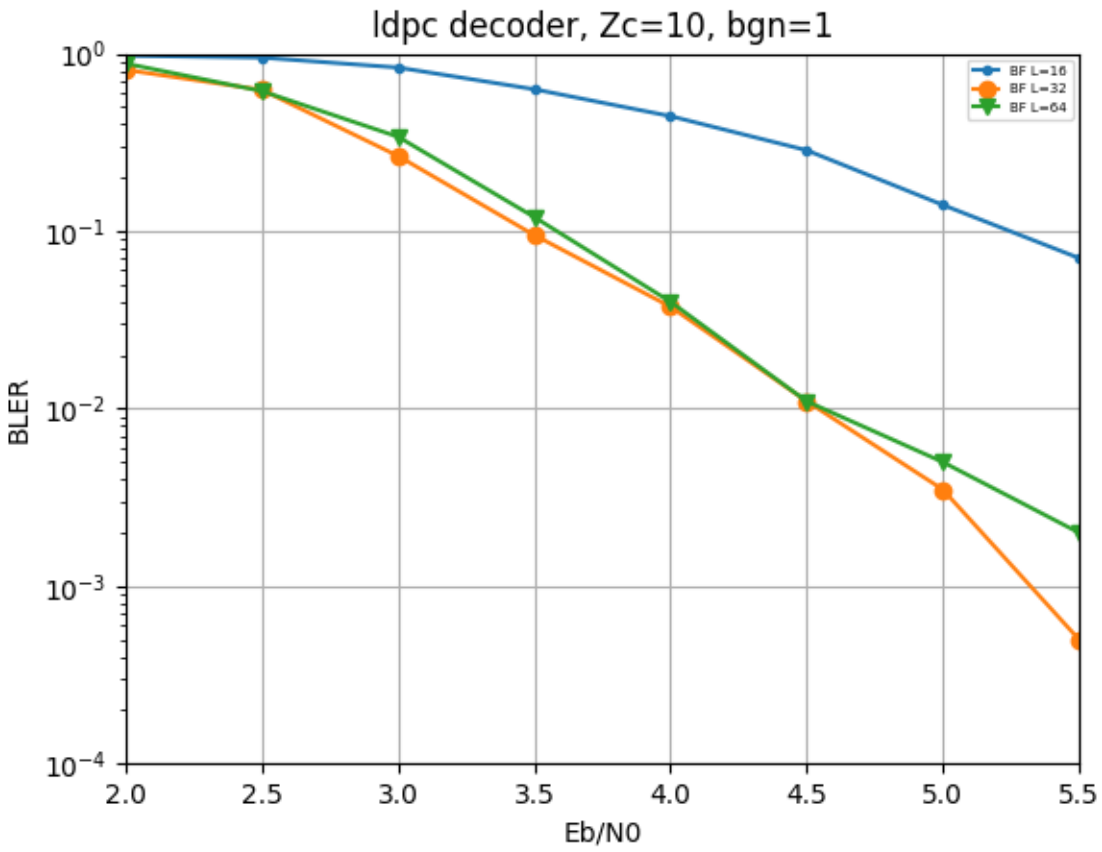
```
#hard coded LLRin to generate ck bit sequence, LLR >0 ->0, LLR<0 -> 1
ck = np.copy(LLRin)
ck[ck > 0] = 0
ck[ck < 0 ] = 1

#main loop
for iter in range(L):
    S = (H @ ck.T) % 2

    if not np.any(S):
        #if S is all zero sequence, decoding success, return Tue
        return ck, True

    #process if S is not all zero
    En = (2*S-1) @ H #this is correct equation
    max_value = np.max(En)
    #bit flip any ck bits with En value==max_value
    ck[En==max_value] = 1- ck[En==max_value]
```

## 3.3 BF 仿真




测试配置：

Zc = 10, bgn = 1, K = Zc*22=220, N=Zc*66=660

测试 L size 16, 32, 64 三个迭代值，测试结果显示 L=32 和 64 性能相当，比 L=16 优 1.6dB

BF 仿真代码在 script/sim_ldpc_decoder_bf.py in

https://github.com/hahaliu2001/python_5gtoolbox.git :

# 4 sum-product algorithm(或者 belief propagation in another name)

**备注:**

"belief propagation"和 "sum-product algorithm" 是同一个算法

## 4.1 Reference

- A Generalized Adjusted Min-Sum Decoder for 5G LDPC Codes: Algorithm and Implementation Yuqing Ren, Student Member, IEEE, Hassan Harb, Member, IEEE, Yifei Shen, Member, IEEE, Alexios Balatsoukas-Stimming, Member, IEEE, and Andreas Burg, Senior Member, IEEE

chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://arxiv.org/pdf/2310.15801

- https://github.com/PKU-HunterWu/LDPC-Encoder-Decoder/tree/main
- Improved Min-Sum Decoding of LDPC Codes Using 2-Dimensional Normalization Juntan Zhang and Marc Fossorier, Daqing Gu and Jinyun Zhang
- **Improved Sum-Min Decoding for Irregular LDPC Codes** Gottfried Lechner & Jossy Sayir
- chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/http://staff.ustc.edu.cn/~wyzhou/chapter8.pdf
- An Introduction to LDPC Codes, William E. Ryan, chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/http://tuk88.free.fr/LDPC/ldpcchap.pdf

## 4.2 LDPC 译码用到的概率知识

我看到的 paper 都是直接给结果，没有推导过程。这里给出 LDPC BP decoder 中用到的公式的推导。

### 4.2.1 多随机变量模 2 加概率

这个问题是说，假设输入变量 assume input variables $x1, x2, \ldots . xm$ 概率为:

$$P1 = P(x1 = 0), P2 = P(x2 = 0), \ldots Pm = P(xm = 0)$$

计算 $P(x1 \oplus . \oplus . \oplus xm)$

**首先,**

计算 $P(x1 \oplus x2)$

$$P(x1 \oplus x2 = 0) = P(x1 = 0)P(x2 = 0) + (P(x1 = 1)(x2 = 1) = P1P2 + (1 - P1)(1 - P2)$$
$$= 1 - (P1 + P2) + 2P1P1$$

得到: $2P(x1 \oplus x2 = 0) - 1 = 4P1P2 - 2(P1 + P2) + 1 = (2P1 - 1)(2P2 - 1)$ (A1)

**其次,**

根据$P(x1 \oplus x2)$ 计算 calculate $P(x1 \oplus x2 \oplus x3)$

$$2P(x1 \oplus x2 \oplus x3 = 0) - 1 = (2P(x1 \oplus x2 = 0) - 1)(2P3 - 1) = (2P1 - 1)(2P2 - 1)(2P3 - 1)$$

**递归得到通用公式:**

$$2P(x1\oplus.\oplus.\oplus xm=0)-1\ =\prod_{n=1}^{m}(2P_n-1)$$

然后得到

$$P(x1\oplus.\oplus.\oplus xm=0)\ =\frac{1}{2}+\frac{1}{2}\prod_{n=1}^{m}(2P_n-1)$$

$$P(x1\oplus.\oplus.\oplus xm=1)=1-P(x1\oplus.\oplus.\oplus xm=0)\ =\frac{1}{2}-\frac{1}{2}\prod_{n=1}^{m}(2P_n-1)$$

**LLR 公式**

LDPC 译码一般都用 LLR(log-likelihood ratio)值在传递概率

定义: $L(x)=LLR(x)=\log\frac{P(x=0)}{P(x=1)}$

得到:

$$P(x=0)=\frac{e^{L(x)}}{1+e^{L(x)}},\ \ P(x=1)=\frac{1}{1+e^{L(x)}}$$

得到:

$$2P(x=0)-1=\frac{e^{L(x)}-1}{e^{L(x)}+1}$$

由 $\tanh(t)=\frac{e^{2t}-1}{e^{2t}+1}$ 定义,

得到 $2P(x=0)-1=\tanh\left(\frac{L(x)}{2}\right)=\frac{e^{L(x)}-1}{e^{L(x)}+1}$

由此得到:

$$\prod_{n=1}^{m}(2P_n-1)=\prod_{n=1}^{m}\tanh\left(L_n(x)/2\right)$$

下面三个 LLR 公式在不同的 paper 中使用

**Equation 1:**

$$LLR_s=log\frac{P(x1\oplus.\oplus.\oplus xm=0)}{P(x1\oplus.\oplus.\oplus xm=1)}=log\frac{\frac{1}{2}+\frac{1}{2}\prod_{n=1}^{m}\tanh\left(L_n(x)/2\right)}{\frac{1}{2}-\frac{1}{2}\prod_{n=1}^{m}\tanh\left(L_n(x)/2\right)}=log\frac{1+\prod_{n=1}^{m}\tanh\left(L_n(x)/2\right)}{1-\prod_{n=1}^{m}\tanh\left(L_n(x)/2\right)}\qquad(B1)$$

**Equation 2:**

With definition: $tanh^{-1}(t) = \frac{1}{2}\log(\frac{1+t}{1-t})$

$$LLR_s = 2tanh^{-1}(\prod_{n=1}^{m}\tanh(L_n(x)/2)) \quad \text{(B2)}$$

**Equation 3:**

由 $\tanh(L_n(x)) = sign(L_n(x))\tanh(|L_n(x)/2|)$

得到：

$$\prod_{n=1}^{m}\tanh(L_n(x)/2) = \prod_{n=1}^{m}\text{sign}(L_n(x))\prod_{n=1}^{m}\tanh(|L_n(x)/2|) = sA$$

其中 $s = \prod_{n=1}^{m}\text{sign}(L_n(x))$, $A = \prod_{n=1}^{m}\tanh(|L_n(x)/2|)$

得到: $log\frac{1+sA}{1-sA} = \begin{cases} log\frac{1+A}{1-A} & s=1 \\ -log\frac{1+A}{1-A} & s=-1 \end{cases} = s * log\frac{1+A}{1-A}$

LLR 公式 3:

$$LLR_s = \prod_{n=1}^{m}\text{sign}(L_n(x)) * 2tanh^{-1}(\prod_{n=1}^{m}\tanh(|L_n(x)/2|)) \quad \text{(B3)}$$

### 4.2.2 多事件条件概率

由 $\frac{P(c|x)}{P(c)} = \frac{P(x|c)}{P(x)}$

得到

$$\frac{P(c|x1,x2,\dots,xm)}{P(c)} = \frac{P(x1,x2,\dots,xm|c)}{P(x1,x2,\dots,xm)} = \frac{P(x1|c)}{P(x1)}\frac{P(x2|c)}{P(x2)}\dots\frac{P(xm|c)}{P(xm)} =$$

$$\frac{P(c|x1)}{P(c)}\frac{P(c|x2)}{P(c)}\dots\frac{P(c|xm)}{P(c)}$$

其中:

$x1, x2, \dots, xm$ 相互独立

由此得到：

$$P(c=0|x1,x2,\dots,xm) = K\prod_{n=1}^{m} P(c=0|x_n),$$

$$P(c=1|x1,x2,\dots,xm) = K\prod_{n=1}^{m} P(c=1|x_n),$$

其中:

$K = \frac{1}{\prod_{n=1}^{m-1} P(c)}$ 为常量

LLR 公式:

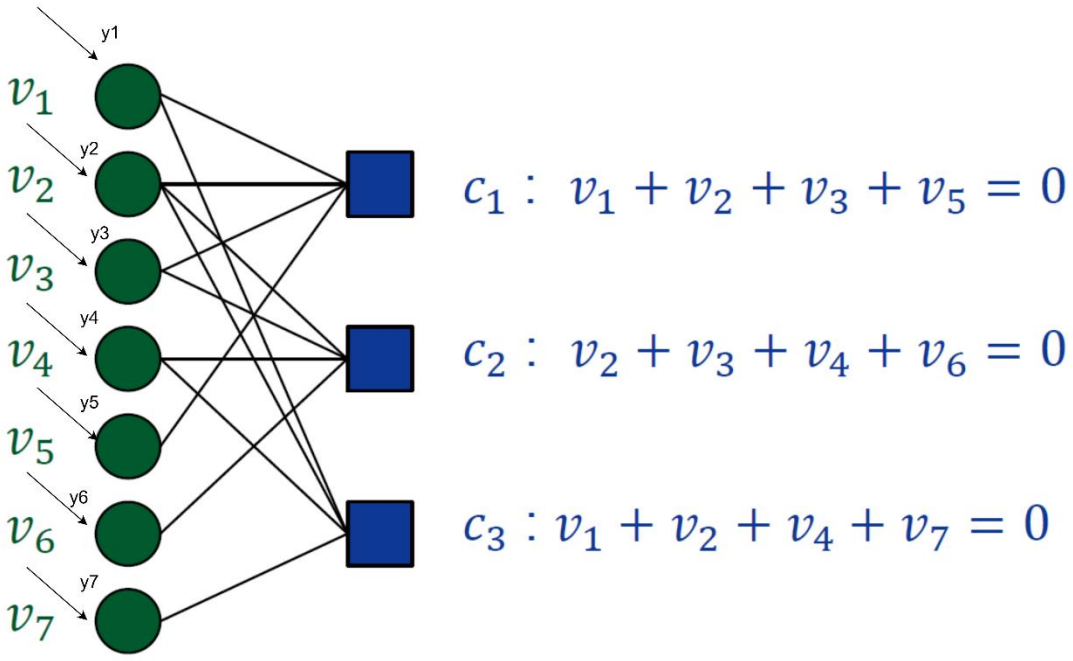
$$LLR(c|x1,x2,\dots,xm) = log\frac{P(c=0|x1,x2,\dots,xm)}{P(c=1|x1,x2,\dots,xm)} = \sum_{n=1}^{m} LLR(c|x_n) \qquad (C1)$$

## 4.3 通过例子学习 SP algorithm

首先最好通过一个例子来学习 SP algorithm.

Parity-check matrix:

$$\boldsymbol{H} = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$



上面是 (N=7, M=3) H 校验矩阵和 Tanner 图.

$v_1, v_2, \ldots v_7$ 是 variable nodes

$c_1, c_2, c_3$ 是 check nodes

$y_1, y_2, \ldots y_7$ 是译码器外部输入到 variable nodes 值，一般来自外部信道均衡输出

通常提供$y_1, y_2, \ldots y_7$ LLR 值

## Step 1: Check node 处理

每个 check node 期望值都是零，也就是说对$c1: v1 \oplus v2 \oplus v3 \oplus v5 = 0$, 如果 $v2 \oplus v3 \oplus v5 = 0\ or\ 1$, 那么$v1$= also 0 or 1

概率值$P(v1|c1) = P(v2 \oplus v3 \oplus v5)$ 是 check node c1 到 variable node v1 传递的概率值

根据公式 (B2) 得到:

$$LLR(v1|c1) = 2tanh^{-1}(\tanh(LLR(v2)/2)\tanh(LLR(v3)/2)\tanh(LLR(v5)/2))$$

类似对于 $c3: v1 \oplus v2 \oplus v4 \oplus v7 = 0$

$P(v1|c3) = P(v2 \oplus v4 \oplus v7)$是 check node c3 到 variable node v1 传递的概率值

$$LLR(v1|c3) = 2tanh^{-1}(\tanh(LLR(v2)/2)\tanh(LLR(v4)/2)\tanh(LLR(v7)/2))$$

## Step 2: Variable node 处理

$v1$ 接收三个外部信息:

$P(v1|y1), P(v1|c1), P(v1|c3)$

其中:

$P(v1|y1)$ 时译码器外部输入的初始信息

$P(v1|c1), P(v1|c3)$ 是从连接到$v1$的 check nodes 收到的信息

从公式 (C1) 得到:

$$LLR(v1) = LLR(v1|y1) + LLR(v1|c1) + LLR(v1|c3)$$

类似方法得到 $LLR(v2), \ldots, LLR(v7)$

**Step 3: 验证 LDPC decoding result**

从 LLR 值得到 7 个硬 bit 序列: $c_i = \begin{cases} 0 & LLR(v_i) > 0 \\ 1 & else \end{cases}$

如果$Hc = 0$, LDPC 译码成功，退出 LDPC 译码.

否则进入下一步.

**Step 4: 更新从 variable node 到 check node 的 LLR messages if LDPC decoding failed**

$LLR(v1)$ 使用初试 LLR 和来自 check node 的 LLR 值计算得到

更新从 v1 到任何一个 check node 的 LLR 值时，需要去掉来自这个 check node 的 LLR。例如 for v1:

$$LLR(v1 \to c1) = LLR(v1|y1) + LLR(v1|c3)$$
$$LLR(v1 \to c3) = LLR(v1|y1) + LLR(v1|c1)$$

类似方法计算所有从 variable node 到 check node 的 LLR，回到 step 1.

重复上面过程，直到 LDPC 译码成功，或者达到最大迭代次数.

### 4.4 通用 SP algorithm

**公式中使用到的标识**

$\mathrm{LQ}_n$ variable node n 总 LLR

$\mathrm{L}_n$ variable node n 初始 LLR

$\mathrm{Lq}_{nm}$ is the LLR from variable node n to check node m

$\mathrm{Lr}_{mn}$ is the LLR from check node m to variable node n

$\mathrm{c}_n$ is the decoded bit for variable node n

A(j) is '1' position list for H line j, is all variable nodes connecting to check node j

B(i) is '1' position list for H column I, if all check nodes to connect to variable node i

**流程**

Step 1 initialization $\mathrm{Lq}_{nm} = \mathrm{L}_n$

Step 2: calculate $\mathrm{Lr}_{mn}$,LLR from check node m to variable node n

$\mathrm{Lr}_{mn} = 2tanh^{-1}(\prod_{i\in A(m),i!=n} \tanh\,(Lq_{mi}/2))$

Step 3: calculate overall LLR for variable nodes

$$\mathrm{LQ}_n = \sum_{j\in B(n)} Lr_{jn}$$

Step 4: check LDPC result

Hard -bit decision: $c_n = \begin{cases} 0 & LQ_n > 0 \\ 1 & else \end{cases}$

If $Hc = 0$ or reach maximum iteration:

Terminate LDPC decoding

Else:

Step 5: update $\mathrm{Lq}_{nm}$ and go back step 1

$$\mathrm{Lq}_{nm} = \mathrm{LQ}_n - Lr_{mn}$$

## 5 Min-sum 最小和 algorithm

Step 2 in BP algorithm $\mathrm{Lr}_{mn} = 2tanh^{-1}(\prod_{i\in A(m),i!=n} \tanh\,(Lq_{mi}/2)))$ 硬件和软件实现的效率很低，Min-sum algorithm 用来简化这个步骤。

From equation (B3) we get:

$\mathrm{Lr}_{mn} = \prod_{i\in A(m),i!=n} \mathrm{sign}\,(Lq_{mi})) * 2tanh^{-1}(\prod_{i\in A(m),i!=n} \tanh\,(|Lq_{mi}/2|)))$

With approximation:

$$\prod_{i\in A(m),i!=n} \tanh\,(|Lq_{mi}/2|) \approx \min_{i\in A(m),i!=n} \tanh\,(\left|\frac{Lq_{mi}}{2}\right|)$$

We get:

$$\mathrm{Lr}_{mn} = \prod_{i\in A(m),i!=n} \mathrm{sign}\,(Lq_{mi})) * \min_{i\in A(m),i!=n} |Lq_{mi}|$$

用上面公式替换 BP $\mathrm{Lr}_{mn}$的计算，其他步骤不变，就是传统的 min-sum 算法.

下面公式用于 the optimization of min-sum approximation:

$$\prod_{i\in A(m),i!=n} \tanh\,(\left|\frac{Lq_{mi}}{2}\right|) \approx \alpha * \max\,(\min_{i\in A(m),i!=n} \tanh\,(\left|\frac{Lq_{mi}}{2}\right|) - \beta, 0)$$

- $\alpha = 1, \beta = 0$: traditional min-sum algorithm
- $\alpha\ in\ (0,1), \beta = 0$: normalized min-sum algorithm
- $\alpha = 1, \beta\ in(0,1)$ offset min-sum algorithm
- $\alpha\ in(0,1), \beta\ in(0,1)$ mixed min-sum algorithm

前三个 min-sum 算法都可以在 paper 中找到，matlab 5g toolbox 也实现了这三个算法。我没有看到 paper 或者 open source 提到 mixed min-sum. 但是从上面的公式可以很自然的想到'what happen if $\alpha\ in(0,1), \beta\ in(0,1)$'? 我的仿真中如果正确选择 $\alpha, \beta$值，mixed min-sum 性能时最好的。

## 关于 min-sum 的进一步优化，有兴趣的可以开发。

1. H 不同行可以分配单独的$\alpha, \beta$值

$\alpha, \beta$值很大的可能跟 H 矩阵每行 '1' 的个数有关。5G LDPC 是不规则矩阵，也就是 H 矩阵每行'1 的个数不用。每行选择不同的$\alpha, \beta$值，对 LDPC decoder 性能也许会有提升。这个功能。

2. 不同迭代分配不同的$\alpha, \beta$值

当前算法相同的$\alpha, \beta$值用于所有的迭代。如果每次分配不同的$\alpha, \beta$值，也许性能会有一定的提升

3. 不同 Eb/N0 分配不同的$\alpha, \beta$值

### 5.1 $\alpha, \beta$在不同 LDPC 配置下的搜索结果

$\alpha, \beta$值很大可能在不同 LDPC 配置下有不同的最优解。

这个仿真时选择几个 5G LDPC 配置，然后从[0.1,0.3,0.5,0.7,0.9]中搜索最优$\alpha, \beta$值.

NMS_ldpc_search_best_alpha.py, OMS_ldpc_search_best_beta.py, mixed_MS_ldpc_search_best_pair.py

Python script 用于搜索 best $\alpha$ for NMS, best $\beta$ for OMS and best $\alpha, \beta$ for mixed-MS.

测试中使用了不同的 5G LDPC ZC/bgn 配置

测试结果：

1. NMS 在所有 LDPC 配置下$\alpha = 0.7$最优
2. MS 在所有 LDPC 配置下 $\beta\ = 0.5$ 最优
3. mixed-MS $\alpha = 0.8,\ \beta = 0.3$ 最优
4. 对所有 bgn=1 配置最大行向量（H 矩阵每行'1'的个数）是 19
5. 对所有 bgn=2 配置最大行向量（H 矩阵每行'1'的个数）是 10

Below is the test result

| NMS alpha searching for - 0.5dB Eb/N0 and L=32 5G LDPC | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 5G LDPC configuration | | | | | BLER for different alpha value | | | | |
| Zc | bgn | N | max line weight | min line weight | 0.1 | 0.3 | 0.5 | 0.7 | 0.9 |
| 8 | 1 | 544 | 19 | 3 | 1 | 0.88 | 0.245 | 0.16 | 0.44 |
| 8 | 2 | 416 | 10 | 3 | 0.995 | 0.17 | 0.00375 | 0.0005 | 0.005 |
| 12 | 1 | 816 | 19 | 3 | 1 | 0.935 | 0.19 | 0.085 | 0.405 |
| 12 | 2 | 624 | 10 | 3 | 1 | 0.285 | 0 | 0 | 0.0005 |
| 28 | 1 | 1904 | 19 | 3 | 1 | 0.995 | 0.055 | 0.0125 | 0.35 |
| 28 | 2 | 1456 | 10 | 3 | 1 | 0.425 | 0 | 0 | 0 |
| 40 | 1 | 2720 | 19 | 3 | 1 | 1 | 0.02 | 0.0025 | 0.36 |
| 40 | 2 | 2080 | 10 | 3 | 1 | 0.405 | 0 | 0 | 0 |
| 72 | 1 | 4896 | 19 | 3 | 1 | 1 | 0.02 | 0.0005 | 0.35 |
| 72 | 2 | 3744 | 10 | 3 | 1 | 0.78 | 0 | 0 | 0 |
| 176 | 1 | 11968 | 19 | 3 | 1 | 1 | 0.035 | 0 | 0.21 |
| 176 | 2 | 9152 | 10 | 3 | 1 | 0.97 | 0 | 0 | 0 |
| 208 | 1 | 14144 | 19 | 3 | 1 | 1 | 0.055 | 0 | 0.145 |
| 208 | 2 | 10816 | 10 | 3 | 1 | 0.975 | 0 | 0 | 0 |
| 384 | 1 | 26112 | 19 | 3 | 1 | 1 | 0.05 | 0 | 0.115 |

| OMS beta searching for -0.5dB Eb/N0 and L=16 5G LDPC | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 5G LDPC configuration | | | | | BLER for different beta value | | | | |
| Zc | bgn | N | max line weight | min line weight | 0.1 | 0.3 | 0.5 | 0.7 | 0.9 |
| 12 | 1 | 816 | 19 | 3 | 0.54 | 0.245 | 0.2 | 0.195 | 0.44 |
| 12 | 2 | 624 | 10 | 3 | 0 | 0.0005 | 0 | 0 | 0 |
| 28 | 1 | 1904 | 19 | 3 | 0.54 | 0.14 | 0.09 | 0.09 | 0.295 |
| 28 | 2 | 1456 | 10 | 3 | 0 | 0 | 0 | 0 | 0 |
| 40 | 1 | 2720 | 19 | 3 | 0.505 | 0.19 | 0.0225 | 0.05 | 0.285 |
| 40 | 2 | 2080 | 10 | 3 | 0 | 0 | 0 | 0 | 0 |
| 72 | 1 | 4896 | 19 | 3 | 0.7 | 0.095 | 0.00625 | 0.02 | 0.355 |
| 72 | 2 | 3744 | 10 | 3 | 0 | 0 | 0 | 0 | 0 |
| 176 | 1 | 11968 | 19 | 3 | 0.685 | 0.0175 | 0.0005 | 0.0025 | 0.43 |
| 176 | 2 | 9152 | 10 | 3 | 0 | 0 | 0 | 0 | 0 |
| 208 | 1 | 14144 | 19 | 3 | 0.735 | 0.02 | 0 | 0.002 | 0.48 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 208 | 2 | 10816 | 10 | 3 | 0 | 0 | 0 | 0 | 0 |

| mixed MS [alpha,beta] searching for - 1dB Eb/N0 and L=16 5G LDPC | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 5G LDPC configuration | | | | | | [alpha, bler] pair and BLER | | | |
| Zc | bgn | N | max line weight | min line weight | Eb/N0 | [0.7,0.5] | [0.7,0.3] | [0.5,0.5] | [0.8,0.3] |
| 12 | 1 | 816 | 19 | 3 | -1dB | 0.55 | 0.32 | 0.96 | 0.26 |
| 12 | 1 | 816 | 19 | 3 | -0.5dB | 0.12 | 0.075 | 0.675 | 0.035 |
| 12 | 2 | 624 | 10 | 3 | -1dB | 0 | 0 | 0.06 | 0 |
| 12 | 2 | 624 | 10 | 3 | -0.5dB | 0 | 0 | 0.006 | 0 |
| 28 | 1 | 1904 | 19 | 3 | -1dB | 0.385 | 0.14 | 0.99 | 0.095 |
| 28 | 1 | 1904 | 19 | 3 | -0.5dB | 0.022 | 0.0025 | 0.74 | 0.00625 |

# 6 LDPC Simulation and comparison with matlab toolbox LDPC decoder

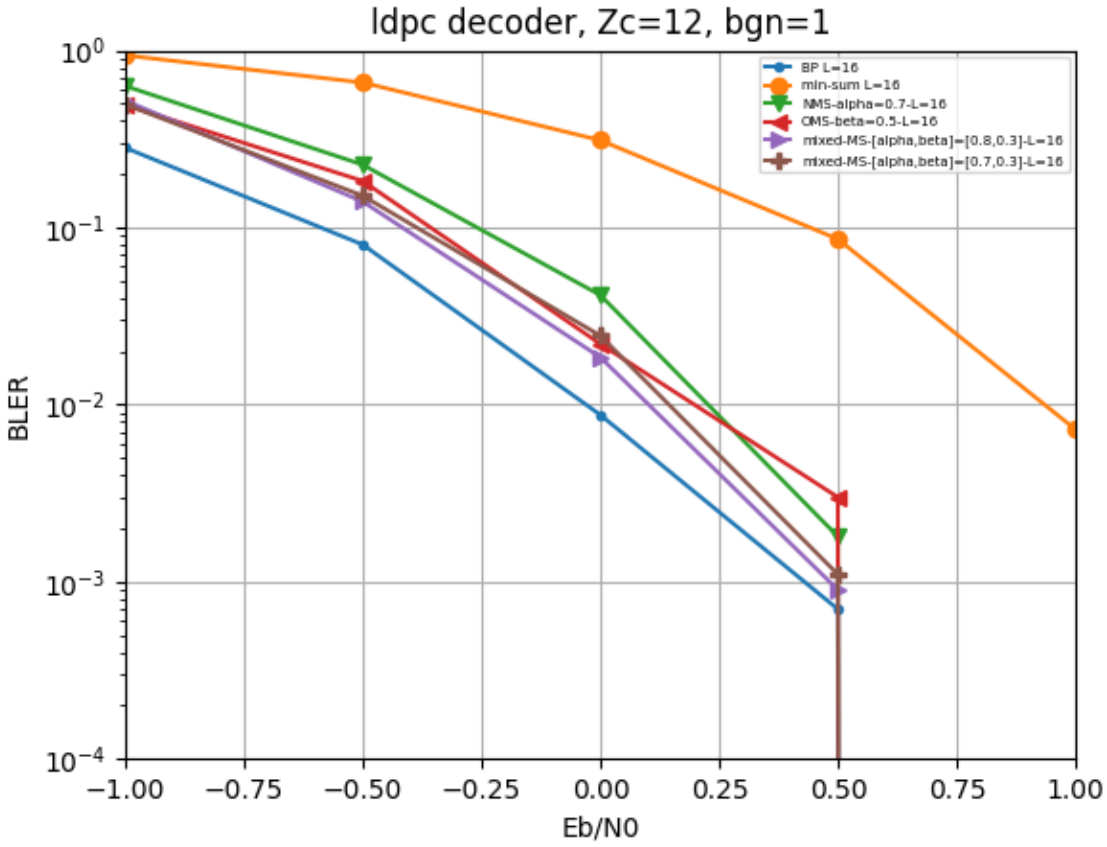
## 6.1 Summary of LDPC simulation and performance analysis

- $\alpha = 0.7$ is optimized for NMS for all LDPC configuration
- $\beta = 0.5$ is optimized for OMS for all LDPC configuration
- $\alpha = 0.8,\ \beta = 0.3$ is optimized for mixed-MS for all LDPC configuration
- BP has the best performance
- Mixed-MS performance is better than other mix-sum algorithms
  - 0.1dB worse than BP,
  - 0.1dB better than NMS and OMS
  - 0.75dB better than traditional min-sum
- L=32 and L=64 performance is close to each other and is 0.2dB better than L=16
-

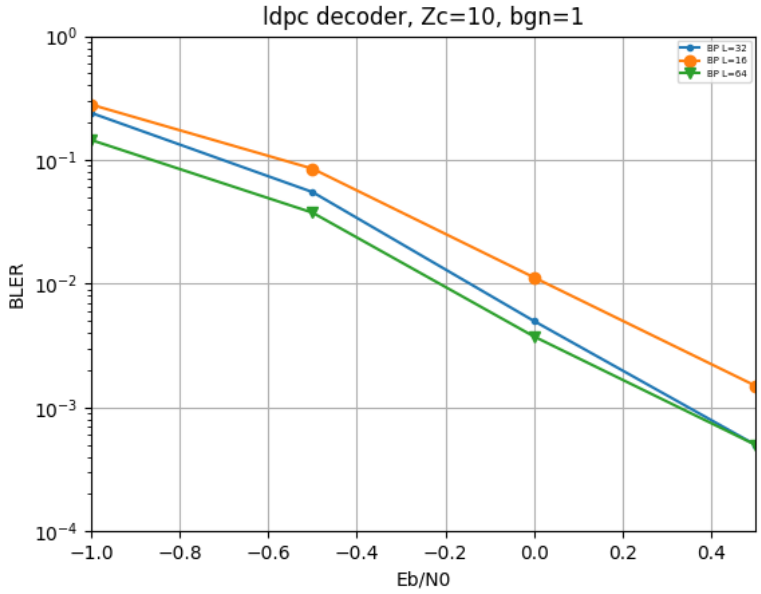
## 6.2 Different LDPC decoder simulation comparison

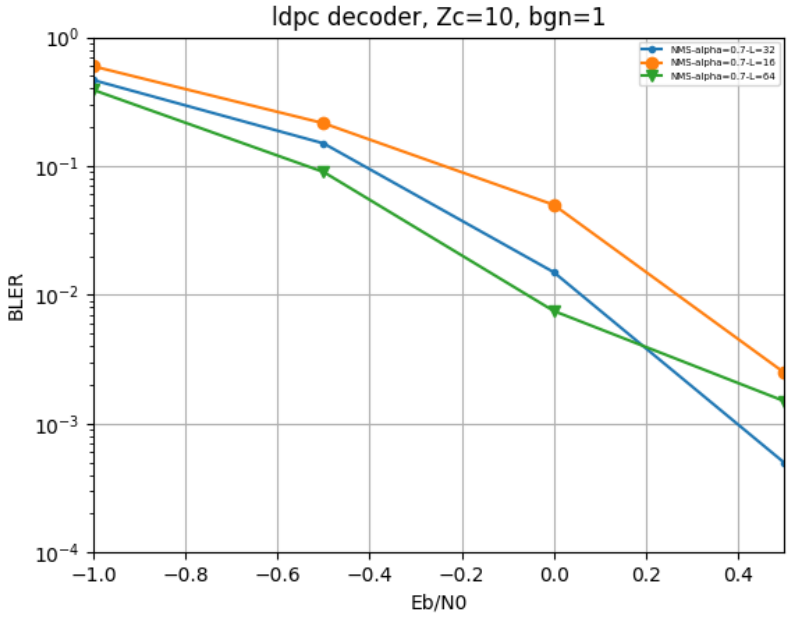
Below test used optimized $\alpha, \beta$ values.

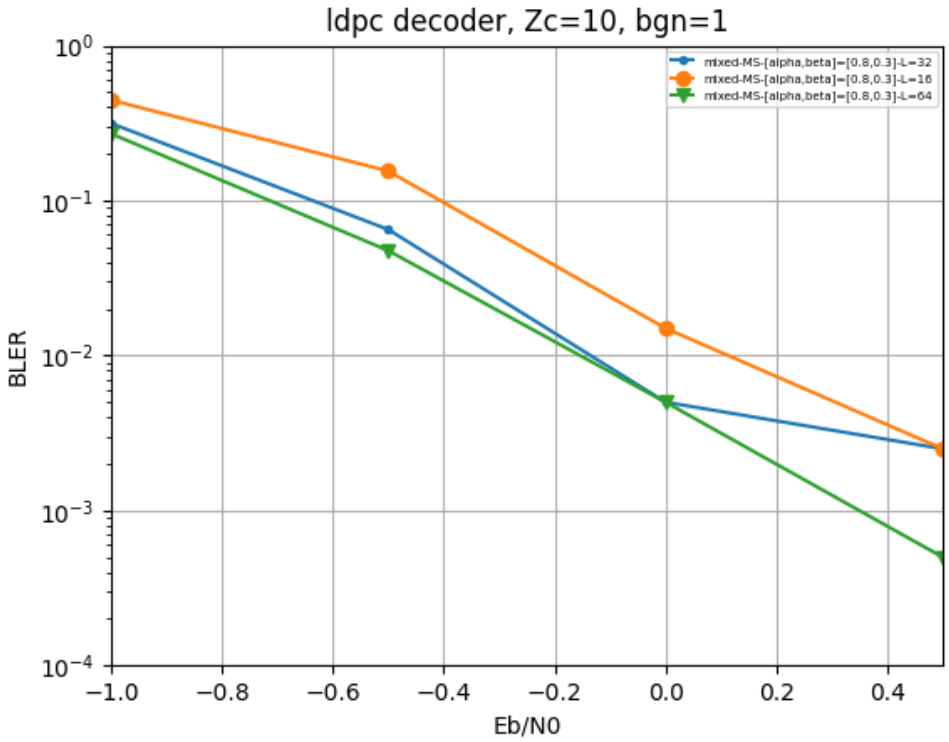
It shows that mixed MS with $\alpha = 0.8,\ \beta = 0.3$ has the best performance among all min-sum algorithm

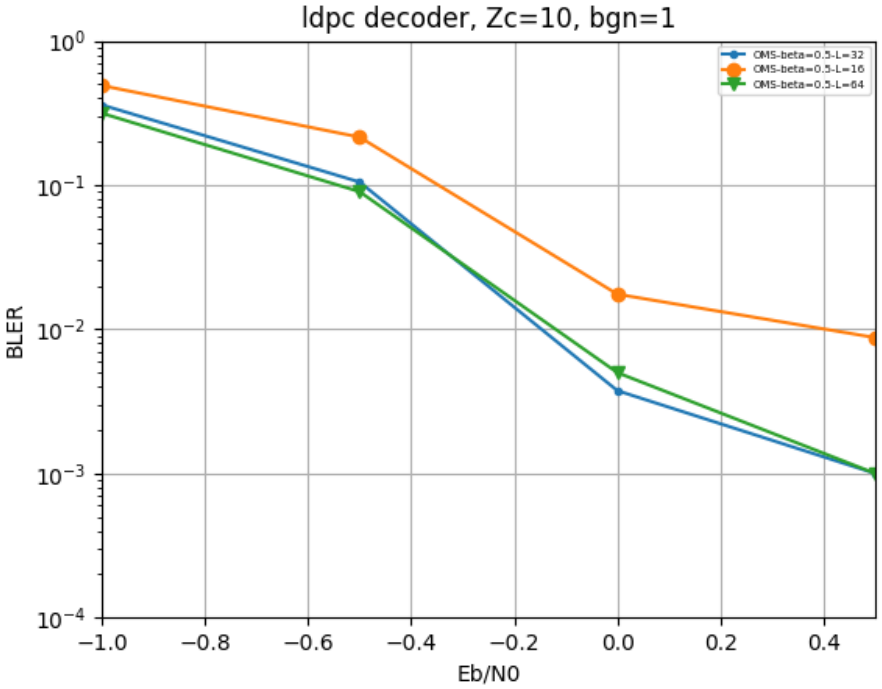



## 6.3 LDPC decoder performance with different iteration L value

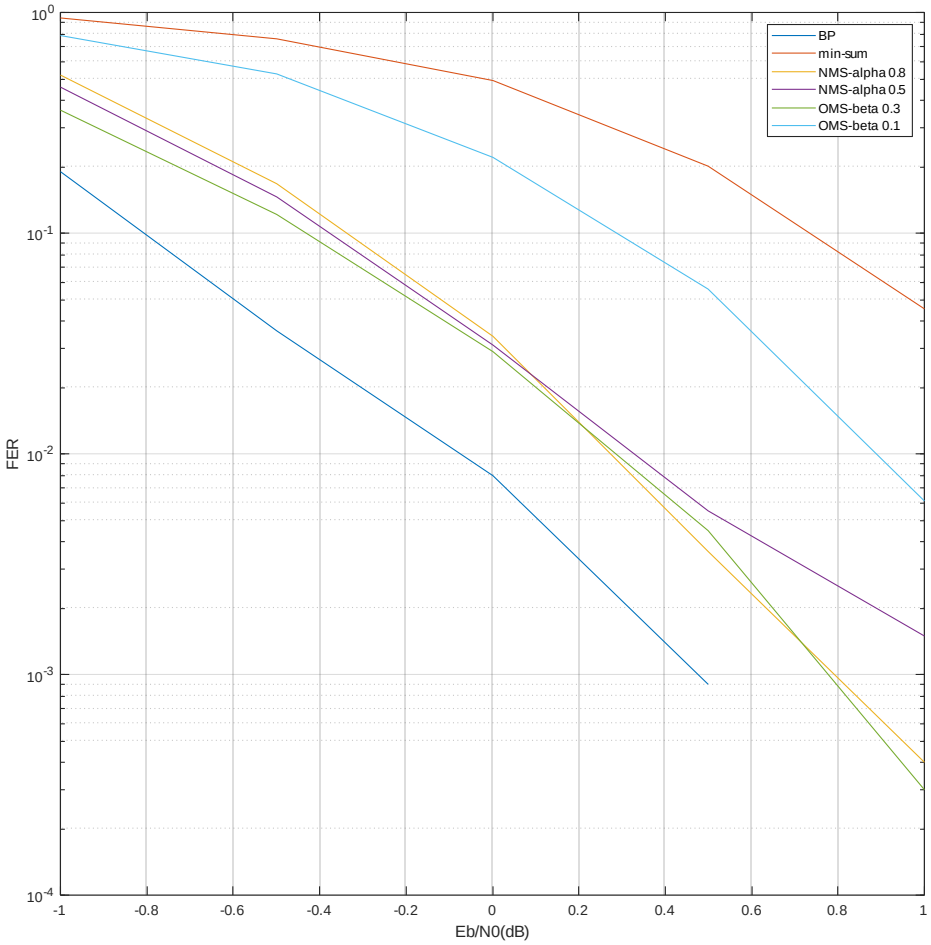


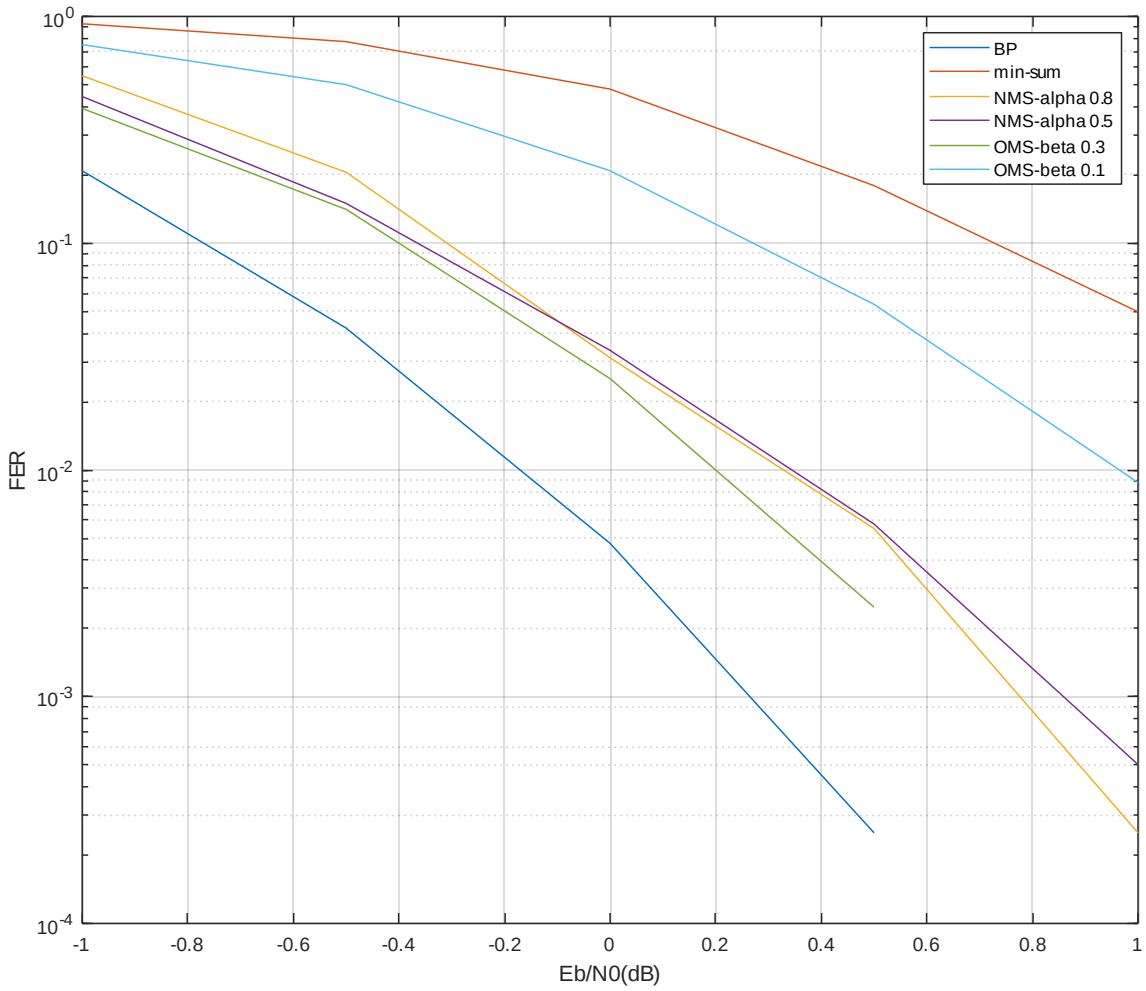

ldpc decoder, Zc=10, bgn=1
NMS-alpha=0.7-L=32
NMS-alpha=0.7-L=16
NMS-alpha=0.7-L=64
BLER
Eb/N0
$10^{0}$
$10^{-1}$
$10^{-2}$
$10^{-3}$
$10^{-4}$
−1.0
−0.8
−0.6
−0.4
−0.2
0.0
0.2
0.4


ldpc decoder, Zc=10, bgn=1
mixed-MS-[alpha,beta]=[0.8,0.3]-L=32
mixed-MS-[alpha,beta]=[0.8,0.3]-L=16
mixed-MS-[alpha,beta]=[0.8,0.3]-L=64
BLER
Eb/N0

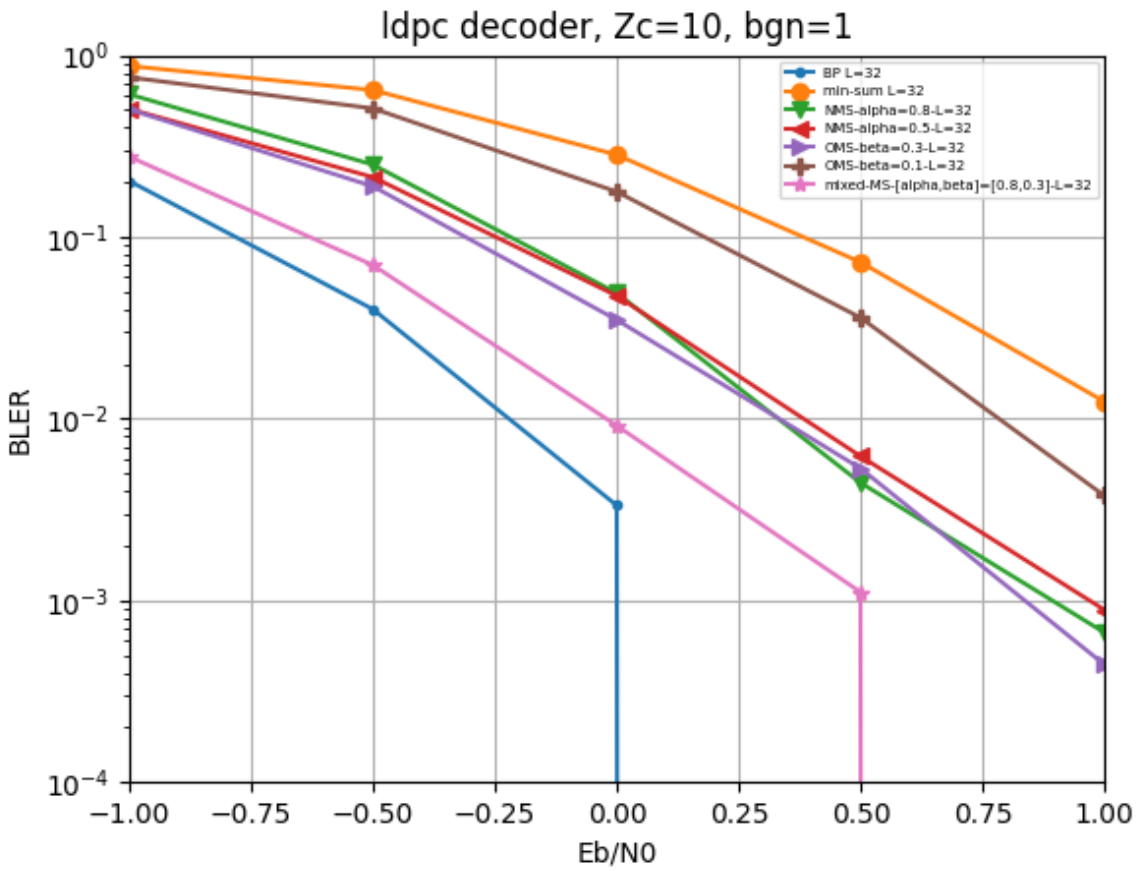

ldpc decoder, Zc=10, bgn=1
OMS-beta=0.5-L=32
OMS-beta=0.5-L=16
OMS-beta=0.5-L=64
BLER
Eb/N0

## 6.4 Matlab toolbox LDPC decoder simulation




BP
min-sum
NMS-alpha 0.8
NMS-alpha 0.5
OMS-beta 0.3
OMS-beta 0.1
FER
Eb/N0(dB)
10^0
10^-1
10^-2
10^-3
10^-4
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1

## 6.5 Py5gphy LDPC decoder simulation




## 6.6 Test analysis and comparison with Matlab toolbox

**Test configuration for both Py5gphy and matlab code**

- Zc = 10, bgn = 1 which means K = Zc*22=220, N = Zc*66==660
- Algorithm: BP, min-sim, normalized min-sum, offset min-sum and mixed-min-sum(Py5gphy only)
- Alpha values used for Normalized min-sim: [0.8, 0.5]
- Beta values used for offset min-sum: [0.3, 0.1]
- Alpha and beta pair used for mixed min-sum: [0.8, 0.3]
- LDPC decoder iteration number: 32

BLER result

| Eb/N0 db | -1 | -0.5 | 0 | 0.5 | 1 |
|---|---|---|---|---|---|
| Matlab BP | 0.21 | 0.0425 | 0.0047 | 0.0003 | 0 |
| Py5gphy BP | 0.203 | 0.04 | 0.0033 | 0 | 0 |
| Matlab min-sum | 0.93 | 0.78 | 0.4758 | 0.1807 | 0.0495 |
| Py5gphy min-sum | 0.87 | 0.64 | 0.28 | 0.073 | 0.012 |
| Matlab NMS alpha 0.8 | 0.545 | 0.205 | 0.0313 | 0.0055 | 0.0003 |
| Py5gphy NMS alpha 0.8 | 0.61 | 0.25 | 0.049 | 0.0044 | 0.00067 |
| Matlab NMS alpha 0.5 | 0.445 | 0.15 | 0.0338 | 0.0057 | 0.0005 |
| Py5gphy NMS alpha 0.5 | 0.51 | 0.21 | 0.048 | 0.0062 | 0.00089 |
| Matlab OMS beta 0.3 | 0.3925 | 0.14 | 0.0253 | 0.0025 | 0 |
| Py5gphy OMS beta 0.3 | 0.503 | 0.19 | 0.035 | 0.0053 | 0.00044 |
| Matlab OMS beta 0.1 | 0.75 | 0.5025 | 0.2075 | 0.0535 | 0.0088 |
| Py5gphy OMS beta 0.1 | 0.76 | 0.51 | 0.178 | 0.036 | 0.0038 |
| Py5gphy mixed MS | 0.28 | 0.07 | 0.0092 | 0.0011 | 0 |

**Observation**

- Py5gphy BP performance is a little better than Matlab BP

  In another test, above BLER for snr_db[-1,-0.5,0] is [0.13,0.03,0.0025]
- Normalized min-sum and offset min-sum performance is a little worse than matlab code
- Mixed min-sum performance is much better than Normalized min-sum and offset min-sum. It is very close to BP performance