中山大學

SUN YAT-SEN UNIVERSITY

Sun yat-sen University

School of Electronics and Information Technology

数值计算期末大作业报告

李杨 *20309131*

2022 年 1 月 23 日

目录

插图

1 文献一

在期中大作业的基础上我们有了准备了复数类、复数矩阵类等相应的基础准备，并在期中报告中给出，此报告不再列出原有的基础而只列出新的改变以及添加。

在 MIMO 系统中用户和基站之间的通信可以表示为：

$$\overline{\mathbf{y}} = \overline{\mathbf{H}}\overline{\mathbf{s}} + \overline{\mathbf{n}}$$

其中 H 为信道矩阵，s 为调制后的发送信号，n 为高斯白噪声。若令

$$\mathbf{y} = \begin{bmatrix} \Re\{\overline{\mathbf{y}}\} \\ \Im\{\overline{\mathbf{y}}\} \end{bmatrix} \in \mathbb{R}^{N\times 1} \quad \mathbf{s} = \begin{bmatrix} \Re\{\overline{\mathbf{s}}\} \\ \Im\{\overline{\mathbf{s}}\} \end{bmatrix} \in \mathbb{R}^{K\times 1}$$

$$\mathbf{n} = \begin{bmatrix} \Re\{\overline{\mathbf{n}}\} \\ \Im\{\overline{\mathbf{n}}\} \end{bmatrix} \in \mathbb{R}^{N\times 1}$$

$$\text{及}\mathbf{H} = \begin{bmatrix} \Re\{\overline{\mathbf{H}}\} & -\Im\{\overline{\mathbf{H}}\} \\ \Im\{\overline{\mathbf{H}}\} & \Re\{\overline{\mathbf{H}}\} \end{bmatrix} \in \mathbb{R}^{N\times K}$$

原式可化为：

$$\mathbf{y} = \mathbf{H}\mathbf{s} + \mathbf{n}$$

而待求发送信号为：

$$\hat{\mathbf{s}} = \left(\mathbf{H}^T\mathbf{H} + \sigma_n^2\mathbf{I}_K\right)^{-1}\mathbf{H}^T\mathbf{y} = \mathbf{A}^{-1}\mathbf{b} \qquad (1)$$

通过对该式的进一步变化可知要求发送信号即寻找函数

$$f(\mathbf{s}) = \frac{1}{2}\mathbf{s}^T\mathbf{A}\mathbf{s} - \mathbf{b}^T\mathbf{s}$$

的最小值。即寻找 $\hat{\mathbf{s}}$ 使 $f(\hat{\mathbf{s}})$ 最小。

在此基础上即可通过最优化方法来还原发送信号。

1.1 BFGS 算法

BFGS 算法的迭代格式如下所示：

$$\mathbf{s}_{k+1} = \mathbf{s}_k + \alpha_k\mathbf{d}_k$$

其中 α_k 为步长，$\mathbf{d}_k$ 为搜索方向。中 α_k 可由以下式子得到：

$$\begin{aligned}\alpha_k &= \arg\min_{\alpha\geq 0} f\left(\mathbf{s}_k + \alpha\mathbf{d}_k\right) \\ &= -\frac{\mathbf{g}_k^T\mathbf{d}_k}{\mathbf{d}_k^T\mathbf{A}\mathbf{d}_k},\end{aligned}$$

其中 $\mathbf{g}_k$ 为 $f(\mathbf{s})$ 的梯度。在拟牛顿法中第 k 次迭代的搜索方向 $\mathbf{d}_k$ 是由

$$\mathbf{d}_k = -\mathbf{B}_k\mathbf{g}_k$$

给出。

对于 $\mathbf{B}_k$, 其迭代过程如下：

$$\begin{aligned}\mathbf{B}_{k+1} &= \mathbf{B}_k - \frac{\mathbf{p}_k\mathbf{q}_k^T\mathbf{B}_k + \mathbf{B}_k\mathbf{q}_k\mathbf{p}_k^T}{\mathbf{p}_k^T\mathbf{q}_k} + \left(1 + \frac{\mathbf{q}_k^T\mathbf{B}_k\mathbf{q}_k}{\mathbf{p}_k^T\mathbf{q}_k}\right)\frac{\mathbf{p}_k\mathbf{p}_k^T}{\mathbf{p}_k^T\mathbf{q}_k} \\ &= \left(\mathbf{I} - \frac{\mathbf{p}_k\mathbf{q}_k^T}{\mathbf{p}_k^T\mathbf{q}_k}\right)\mathbf{B}_k\left(\mathbf{I} - \frac{\mathbf{q}_k\mathbf{p}_k^T}{\mathbf{p}_k^T\mathbf{q}_k}\right) + \frac{\mathbf{p}_k\mathbf{p}_k^T}{\mathbf{p}_k^T\mathbf{q}_k} \\ &= \mathbf{V}_k^T\mathbf{B}_k\mathbf{V}_k + \rho_k\mathbf{p}_k\mathbf{p}_k^T,\end{aligned}$$

其中 $\rho_k = 1/\mathbf{p}_k^T\mathbf{q}_k$ 且 $\mathbf{V}_k = \mathbf{I} - \rho_k\mathbf{q}_k\mathbf{p}_k^T$

有了算法原理便能进行编程实现。程序如下：

```
Matrix BFGS(complex_matrix H_hat,complex_matrix y_hat,double s)//BFGS算法 s为方差
{
    int N = 2 * y_hat.rows_num;
    int K = 2 * H_hat.cols_num;
    //complex_matrix temp_y(2 * y.rows_num, 1);
    Matrix R_y = y_hat.realmatrix();//实部矩阵
    Matrix I_y = y_hat.imagmatrix();//虚部矩阵

    Matrix R_H = H_hat.realmatrix();
    Matrix I_H = H_hat.imagmatrix();

    Matrix y=R_y.combine_rows(R_y,I_y);

    Matrix H = R_H.combine_rows(R_H, I_H);
    Matrix temp_imagpart = Matrix(H_hat.rows_num, H_hat.cols_num) - I_H;//负的H的虚部矩阵
    Matrix temp = temp_imagpart.combine_rows(temp_imagpart, R_H);
    H= H.combine_cols(H, temp);//最终得到H

    Matrix A = Matrix::T(H) * H + Matrix::eye(K) * s;
    Matrix b = Matrix::T(H) * y;

    Matrix s0 = generate_signal(K).realmatrix();//产生一个随机的s0
    Matrix B = Matrix::eye(s0.rows_num);//初始时可以为任意对称正定矩阵
    Matrix g=get_negetive_gradient(*f_s,s0,A,b);//notice that g is negative

    Matrix d;
    d = B * g;

    Matrix p;
    Matrix q;
```

```
    Matrix temps;
    Matrix tempg;

    double alpha = (Matrix::T(g) * d).p[0][0] / ((
         Matrix::T(d) * A * d)).p[0][0];
    //因为求得的g为负梯度，因此不需要加负号
    for (size_t i = 0; i < 5; i++)
    {
        temps = s0;//save s0

        s0 = s0 + d*alpha;//迭代

        tempg = g;
        g = get_negetive_gradient(*f_s, s0, A, b);//
             update g, notice that g is negative

        //更新B
        p = s0 - temps;
        q = tempg-g;//because g is negative

        //q = Matrix(q.rows_num, q.cols_num) - q;//
             because g is negative. make it positive
        double ro = 1.0 / ((Matrix::T(p) * q).p
             [0][0]);
        Matrix V = Matrix::eye(s0.rows_num) - q *
             Matrix::T(p) * ro;
        B = Matrix::T(V) * B * V + p * Matrix::T(p)
             * ro;

        d = B * g;//更新dk

        alpha= (Matrix::T(g) * d).p[0][0] / ((Matrix
             ::T(d) * A * d)).p[0][0];//更新

    }
    return s0;
}
```

1.2 LBFGS 算法

由于随着迭代的进行，B 矩阵变得稠密。因此 BFGS 方法的存储和计算代价将会变得很大，这对于大规模 MIMO 系统来说是不现实的。在此条件下，LBFGS 算法被提出，其并不显式地保存 B 矩阵而通过储存一定数目的 p 矩阵和 q 矩阵来得到 B 矩阵。

在 LBFGS 算法中我们通过一个双向循环来得到每次迭代过程中的 B 矩阵，算法循环的算法如下：

Algorithm 1 L-BFGS two-loop recursion

Input: a certain number m of $\{\mathbf{p}_i, \mathbf{q}_i\}$, and $\rho_i = 1/\left(\mathbf{p}_i^T \mathbf{q}_i\right)$ for all $i \in k-m+1, \cdots, k$, the current gradient $\mathbf{g}_{k+1}$;

Output: new search direction $\mathbf{d}_{k+1} = -\mathbf{B}_{k+1}\mathbf{g}_{k+1}$;

$\mathbf{r} = \mathbf{g}_{k+1}$;
for $i = k$ to $k-m+1$ **do**
  $\alpha_i = \rho_i \mathbf{p}_i^T \mathbf{r}$;
  $\mathbf{r} = \mathbf{r} - \alpha_i \mathbf{q}_i$;
end for
$\mathbf{r} = \mathbf{B}_k^0 \mathbf{r}$;
for $i = k-m+1$ to k **do**
  $\beta = \rho_i \mathbf{q}_i^T \mathbf{r}$;
  $\mathbf{r} = \mathbf{r} + \mathbf{p}_i\left(\alpha_i - \beta\right)$;
end for
Return: $\mathbf{d}_{k+1} = -\mathbf{r}$.

而 LBFGS 算法的主算法如下：

Algorithm 2 L-BFGS method for MMSE detection

Input: $\mathbf{H}, \mathbf{y}, \sigma_n^2$;

Output: $\hat{\mathbf{s}}$;

Initialization:

Initialize starting vector $\mathbf{s}_0$, inverse Hessian approximation $\mathbf{B}_0$, accuracy threshold ϵ, iteration number L, correction number m; Compute $\mathbf{A} = \mathbf{H}^T\mathbf{H} + \sigma_n^2\mathbf{I}_K, \mathbf{b} = \mathbf{H}^T\mathbf{y}, \mathbf{g}_0 = \mathbf{A}\mathbf{s}_0 - \mathbf{b}$, $\mathbf{d}_0 = -\mathbf{B}_0\mathbf{g}_0$; Set $k = 0$;

Step 1: If $\|\mathbf{g}_k\| \le \epsilon$ or $k = L$, stop and return $\hat{\mathbf{s}} = \mathbf{s}_k$;

Step 2: Find the step length α_k with the formula (10);

Step 3: Compute the new iteration $\mathbf{s}_{k+1} = \mathbf{s}_k + \alpha_k \mathbf{d}_k$;

Step 4: Compute the new gradient $\mathbf{g}_{k+1} = \mathbf{A}\mathbf{s}_{k+1} - \mathbf{b}$;

Step 5: Update $\mathbf{p}_k = \mathbf{s}_{k+1} - \mathbf{s}_k$ and $\mathbf{q}_k = \mathbf{g}_{k+1} - \mathbf{g}_k$;

Step 6: If $k > m$, discard vector pair $\mathbf{p}_{k-m}, \mathbf{q}_{k-m}$ from storage;

Step 7: Use Algorithm 1 to compute the new search direction $\mathbf{d}_{k+1} = -\mathbf{B}_{k+1}\mathbf{g}_{k+1}$;

Step 8: $k := k+1$ and go to Step **1**.

针对算法编写程序如下：

主函数部分：

```
Matrix LBFGS(complex_matrix H_hat,complex_matrix y_hat,double s)
{
    deque<Matrix> deque_of_p;
    deque<Matrix> deque_of_q;
    deque<double> deque_of_ro;
    int m = 1;
    double eps = 1e-6;
    int N = 2 * y_hat.rows_num;
    int K = 2 * H_hat.cols_num;
    //complex_matrix temp_y(2 * y.rows_num, 1);
    Matrix R_y = y_hat.realmatrix();//实部矩阵
    Matrix I_y = y_hat.imagmatrix();//虚部矩阵

    Matrix R_H = H_hat.realmatrix();
    Matrix I_H = H_hat.imagmatrix();

    Matrix y = R_y.combine_rows(R_y, I_y);

    Matrix H = R_H.combine_rows(R_H, I_H);
    Matrix temp_imagpart = Matrix(H_hat.rows_num, H_hat.cols_num) - I_H;//负的H的虚部矩阵
    Matrix temp = temp_imagpart.combine_rows(temp_imagpart, R_H);
    H = H.combine_cols(H, temp);//最终得到H
    //H.Show();
    //y.Show();


    Matrix A = Matrix::T(H) * H + Matrix::eye(K) * s;
    Matrix b = Matrix::T(H) * y;




    Matrix s0 = generate_signal(K).realmatrix();//产生一个随机的s0
    Matrix g0 = A * s0 - b;
    Matrix B0 = Matrix::eye(s0.rows_num);//初始时可以为任意对称正定矩阵
    Matrix B;
    Matrix d0 = B0 * g0;
    d0 = Matrix(d0.rows_num, d0.cols_num) - d0;
    Matrix temps;
    Matrix tempg;
    Matrix p;
    Matrix q;
    for (int k = 0; k < 100; k++)
    {
        if (g0.norm2(0)<=eps)
        {
            break;
        }
        else
        {
            double alpha = -(Matrix::T(g0) * d0).p[0][0] / ((Matrix::T(d0) * A * d0)).p[0][0];

            temps = s0;//save s0

            s0 = s0 + d0 * alpha;//迭代

            tempg = g0;
            g0 = A * s0 - b;

            //更新B
            p = s0 - temps;
            q = g0-tempg;
            double gamma = (Matrix::T(q)*p).p[0][0] / (Matrix::T(q)*q).p[0][0];
            B = B0 * gamma;
            //使用双端队列来保存pi和qi
            deque_of_p.push_back(p);
            deque_of_q.push_back(q);
            deque_of_ro.push_back(1.0 / (Matrix::T(p) * q).p[0][0]);
            if (k > m)
            {
                //k>m删除前端元素
                deque_of_p.pop_front();
                deque_of_q.pop_front();
                deque_of_ro.pop_front();
            }
             d0 = LBFGS_two_loop(m,k,deque_of_p,deque_of_q
            ,deque_of_ro,g0,B);
        }
    }
    return s0;
}
```

双向循环部分：

```
Matrix LBFGS_two_loop(int m, int k, deque<Matrix> deque_of_p
    , deque<Matrix> deque_of_q, deque<double> deque_of_ro, Matrix gk_1, Matrix Bk)
{
    int delta;
    int L;
    if (k <= m)
    {
        //delta = 0;
        L = k;
    }
    else
    {
        //delta = k - m;
        L = m;
    }
    double* alpha = new double[L];
```

```
    Matrix r = gk_1;
    //int j;
    for (int i = L-1; i >=0; i--)
    {
        //j = i + delta;
        alpha[i] = (Matrix::T(deque_of_p[i]) * r *
             deque_of_ro[i]).p[0][0];
        r = r - deque_of_q[i]* alpha[i];
    }
    r = Bk * r;
    double beta;
    for (int i = 0; i < L-1; i++)
    {
        beta = (Matrix::T( deque_of_q[i]) * r *
             deque_of_ro[i]).p[0][0];
        r = r + deque_of_p[i] * (alpha[i] - beta);
    }
    return Matrix(r.rows_num, r.cols_num) - r;
}
```

算法实现过程中通过三个 deque 双向队列实现 p 和 q 向量的保存与删除，通过 deque 队列能够较为方便地保存一定数目的 pq 向量并能够实现容器内任意元素的访问。

在双向循环函数中，当 k<m 时队列会一直压入 pq 向量，而当 k>m 时则会弹出最前端的元素以便减少算法内存使用量。

1.3 LBFGS 的改进

为了充分利用大规模 MIMO 系统的特性，进一步降低迭代成本，我们可以采用基于 L-BFGS 方法的 MMSE 检测的三种改进策略，包括减少校正向量对的个数、改进初始化矩阵和选择合适的步长。在本次作业中，我们改进了 pq 向量对的个数即搜索方向以及采用了改进后的初始矩阵。

1.3.1 搜索方向改进

由于 m 越小 LBFGS 算法的计算效率越高，考虑 pq 向量对个数为 1 时的情形，通过推导我们可以得到

$$\mathbf{d}_k^{LBFGS} = \mathbf{d}_k^{BFGS}$$

这意味着当 $m = 1$以及$\mathbf{B}_0^k = \mathbf{B}_0$ 时双向循环被忽略而我们可以减少内存的消耗。

基于上述思想将 LBFGS 算法中 m 的值改为 1 即可得到改进后的程序。此处不再列出。

1.3.2 B 矩阵初始化改进

由于初始矩阵 B 的选择会影响 L-BFGS 算法的行为，所以我们可以通过研究 B 矩阵对算法的影响来改进算法。

一个简单的初始化 B 的方法是令 $\mathbf{B_k^0} = \mathbf{I}$，但是如此选择的收敛速度往往较慢. 由于双循环递归使得 B0 与其他算法独立，因此可以在每次迭代中任意选择 B0，所以，一种常见的有效的替代 LBFGS 方法是设置 $\mathbf{B}_k^0 = \gamma_k \mathbf{I}$，其中 γ_k:

$$\gamma_k = \frac{\mathbf{q}_k^T \mathbf{p}_k}{\mathbf{q}_k^T \mathbf{q}_k}$$

在此想法基础上，编写程序只需要在 LBFGS 算法中添加语句：

```
double gamma = (Matrix::T(q)*p).p[0][0] / (Matrix::
     T(q)*q).p[0][0];
          B = B0 * gamma;
```

语句即可。

1.4 算法比较

为了测试不同条件下各个算法的误码率，编写误码率测试程序。误码率测试程序的基本单元如下：

```
out.open("BER_of_BFGS.txt");
    int SNR_max = 13;
    double BER_of_BFGS;
    complex_matrix temp_H = generate_H(nR, nT);
    //outML << "Eb_N0" << " " << "BER" << endl;
    out.open("BER_of_BFGS.txt");
    for (int k = 1; k < SNR_max; k++)
    {
        srand((unsigned)time(NULL));
         BER_of_BFGS = 0;
        for (int i = 0; i < times; i++)
        {
            complex_matrix c = generate_signal(nT)
                 ;//发射信号
            c=quantization(c);
            complex_matrix miu = generate_noise(nR);
            miu =(miu / sqrt(k))*nT;
            complex_matrix temp_x = temp_H * c + miu
                 ;
```

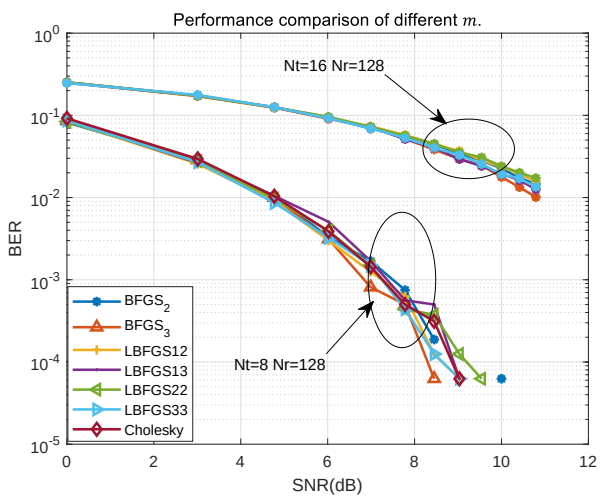

图 1: 不同的 m 下不同 BFGS 算法误码率表现

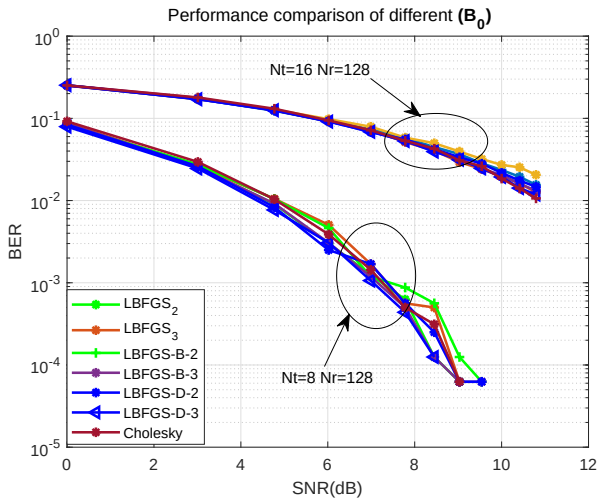

图 2: 不同的初始矩阵 $\mathbf{B}_0$LBFGS 算法误码率表现

```
18          complex_matrix result_of_BFGS =
                Matrix_to_complex(BFGS(temp_H,
                temp_x,0.5*double(k)));
19          //complex_matrix result_of_BFGS =SPCG(
                temp_H, temp_x, 0.5 * double(k));
20          BER_of_BFGS += BitsErrorRate(c,
                result_of_BFGS);
21      }
22      cout << BER_of_BFGS << endl;
23      out << fixed << setprecision(10) << double(k
            ) << "   " << BER_of_BFGS / times << "
            " << endl;;
24  }
25  out.close();
```

测试不同算法和在不同条件下的误码率是改变输出的文件以及对应的算法函数名称即可。

分别测试在不同情况下不同算法的表现，得到的结果绘制对应的曲线如图 1 所示：

结果分析 从误码率结果可以看出以下性质

- 误码率随着信噪比 SNR 的增大而减小
- 误码率随迭代次数增大而减小
- 误码率随 pq 向量对数 m 取值增大而减小

2 文献二

从文献一的讨论中我们知道还原信号即为函数：

$$f(\mathbf{s}) = \frac{1}{2}\mathbf{s}^T\mathbf{A}\mathbf{s} - \mathbf{b}^T\mathbf{s}$$

取最小值时的 s。在复数情况下该方程可化为：

$$f(\mathbf{s}) = \frac{1}{2}\mathbf{s}^H\mathbf{A}\mathbf{s} - \tilde{\mathbf{y}}\mathbf{s} + \frac{1}{2}\mathbf{y}^H\mathbf{y}$$

2.1 传统梯度下降法

寻找上述函数的最小值可使用传统的共轭梯度法，其基本的迭代格式如下所示：

$$\hat{\mathbf{s}}^{(k+1)} = \hat{\mathbf{s}}^{(k)} + \frac{\left(\tilde{\mathbf{y}}^{(k)} \cdot \tilde{\mathbf{y}}^{(k)}\right)}{\left(\mathbf{A}\tilde{\mathbf{y}}^{(k)} \cdot \tilde{\mathbf{y}}^{(k)}\right)}\tilde{\mathbf{y}}^{(k)}$$

$$\tilde{\mathbf{y}}^{(k+1)} = \tilde{\mathbf{y}}^{(k)} - \frac{\left(\tilde{\mathbf{y}}^{(k)} \cdot \tilde{\mathbf{y}}^{(k)}\right)}{\left(\mathbf{A}\tilde{\mathbf{y}}^{(k)} \cdot \tilde{\mathbf{y}}^{(k)}\right)}\mathbf{A}\tilde{\mathbf{y}}^{(k)}$$

在算法基础上编写程序如下：

```
1  complex_matrix SD(complex_matrix H_hat,
       complex_matrix y_hat,double s,int iteration)//
       传统最速梯度下降法 使用梯度下降法求最小值
2  {
3      complex_matrix y_1 = (!H_hat) * y_hat;
4      complex_matrix A = ((!H_hat) * H_hat +
           complex_matrix::eyes(H_hat.cols_num) * s);
5      double eps = 1e-5;
6      complex_matrix s0 = complex_matrix(H_hat.
           cols_num, 1);//generate_signal(H_hat.
           cols_num);//产生随机的s0
```

```
    for (size_t i = 0; i < iteration; i++)
    {
        s0 = s0 + y_1*(((!y_1)*y_1).p[0][0] / ((!(A*
             y_1))*y_1).p[0][0]);
        y_1 = y_1 - A * y_1 * (((!y_1) * y_1).p
             [0][0] / (((!(A * y_1))) * y_1).p
             [0][0]);
    }
    return s0;
}
```

2.2 Barzilai-Borwein 算法

与 SD 法不同的是 BB 算法为拟牛顿法，其在迭代过程中满足：

$$\mathbf{A}_k\mathbf{x}^{(k)} = \mathbf{z}^{(k)}$$

其中 $\mathbf{x}^{(k)} = \hat{\mathbf{s}}^{(k+1)} - \hat{\mathbf{s}}^{(k)}$ 且$\mathbf{z}^{(k)} = \nabla f\left(\hat{\mathbf{s}}^{(k+1)}\right) - \nabla f\left(\hat{\mathbf{s}}^{(k)}\right)$。Barzilai-Borwein 算法的基本迭代格式如下：

$$\begin{aligned}\hat{\mathbf{s}}^{(k+1)} &= \hat{\mathbf{s}}^{(k)} - \frac{1}{\alpha_{k-1}}\nabla f\left(\hat{\mathbf{s}}^{(k)}\right) \\ &= \hat{\mathbf{s}}^{(k)} - \frac{\left(\mathbf{x}^{(k-1)}\cdot\mathbf{x}^{(k-1)}\right)}{\left(\mathbf{A}\mathbf{x}^{(k-1)}\cdot\mathbf{x}^{(k-1)}\right)}\nabla f\left(\hat{\mathbf{s}}^{(k)}\right)\end{aligned}$$

初始时取 $\mathbf{B}_0 = \mathbf{0}$而$\mathbf{B}_1$ 任意。对于此问题中的函数，复数矩阵的梯度向量可表示为:

$$\nabla f\left(\hat{\mathbf{s}}^{(k)}\right) = \mathbf{A}\mathbf{s}\,\mathbf{k} - \tilde{\mathbf{y}}$$

有了迭代格式编写程序如下：

```
double eps = 1e-5;
    complex_matrix y_1 = (!H_hat) * y_hat;
    complex_matrix s0(H_hat.cols_num, 1);
    complex_matrix s1 =complex_matrix(s0.rows_num
         ,1);
    for (int i = 0; i < s0.rows_num;i++)
    {
        s1.p[i][0] = Complex(1, 0);
    }
    complex_matrix x0 = s1 - s0;
    complex_matrix A = (!H_hat) * H_hat +
         complex_matrix::eyes(H_hat.cols_num) * s;

    complex_matrix temps=s1;
    for (size_t i = 0; i < iteration; i++)
    {
        if (x0.norm2(0) <= eps&&i!=0)
        {
            break;
        }
        else
        {
            temps = s1;
            s1 = s1 -gradient_complex(A,s1,y_1)*(x0.
                 cdot() / (A*x0).cdot(x0));//已经为负
                 梯度
            x0 = s1 - temps;
        }
    }
    return s1;
```

其中求梯度函数：

```
complex_matrix gradient_complex(complex_matrix A,
     complex_matrix s, complex_matrix b)
{
    complex_matrix result;
    result = A * s - b;
    return result;
}
```

2.3 算法比较

测试在不同迭代次数下各个算法的表现情况。将得到的结果绘制曲线：

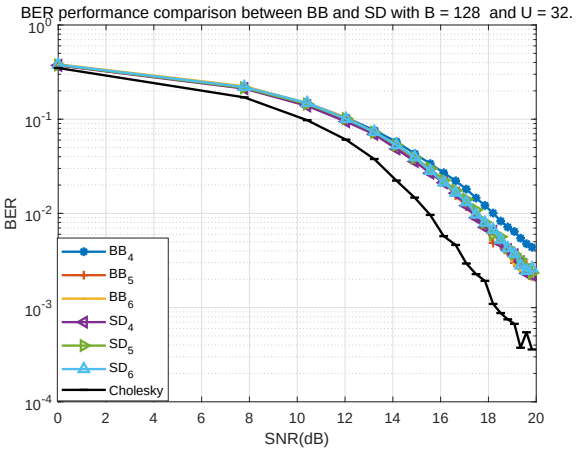

图 3: B=128,U=32 时 BB 与 SD 误码率

结果分析 从结果可以看出 SD 和 BB 算法的误码率接近而都随着迭代次数增大而减小。同时 Cholesky 分解算法的误码率比两者更低且在信噪比 SNR 较高时更为明显。

3 文献三

在前面的文献讨论中我们知道信号还原问题可以化为方程 1的求解问题，在此基础上共轭梯度法也可用于求解此方程。

3.1 共轭梯度 CG 法

共轭梯度法的基本迭代格式为：

$$\hat{\mathbf{s}}^{(i)} = \hat{\mathbf{s}}^{(i-1)} + \frac{\left(\tilde{\mathbf{y}}^{(i-1)} \cdot \tilde{\mathbf{y}}^{(i-1)}\right)}{\left(\mathbf{A}\tilde{\mathbf{p}}^{(i-1)} \cdot \tilde{\mathbf{p}}^{(i-1)}\right)}\tilde{\mathbf{p}}^{(i-1)}$$

$$\tilde{\mathbf{y}}^{(i)} = \tilde{\mathbf{y}}^{(i-1)} - \frac{\left(\tilde{\mathbf{y}}^{(i-1)} \cdot \tilde{\mathbf{y}}^{(i-1)}\right)}{\left(\mathbf{A}\tilde{\mathbf{p}}^{(i-1)} \cdot \tilde{\mathbf{p}}^{(i-1)}\right)}\mathbf{A}\tilde{\mathbf{p}}^{(i-1)}$$

$$\tilde{\mathbf{p}}^{(i)} = \tilde{\mathbf{y}}^{(i)} + \frac{\left(\tilde{\mathbf{y}}^{(i)} \cdot \tilde{\mathbf{y}}^{(i)}\right)}{\left(\tilde{\mathbf{y}}^{(i-1)} \cdot \tilde{\mathbf{y}}^{(i-1)}\right)}\tilde{\mathbf{p}}^{(i-1)}$$

初始化时可取

$$\hat{\mathbf{s}}^{(0)} = 0, \tilde{\mathbf{y}}^{(0)} = \tilde{\mathbf{y}}, \tilde{\mathbf{p}}^{(0)} = \tilde{\mathbf{y}}^{(0)}$$

在上述基础上编写程序如下：

```
complex_matrix CG(complex_matrix H_hat,
     complex_matrix y_hat, double s,int iteration)
{
    double eps = 1e-5;
    complex_matrix y_1 = (!H_hat) * y_hat;
    complex_matrix A = ((!H_hat) * H_hat +
         complex_matrix::eyes(H_hat.cols_num) * s);
    complex_matrix s0(H_hat.cols_num, 1);
    complex_matrix y0 = y_1;
    complex_matrix p0 = y0;
    complex_matrix tempy = y0;;
    for (size_t i = 0; i < iteration; i++)
    {
        if((tempy-y0).norm2(0)<=eps&&i!=0)
        {
            break;
        }
        else
        {
            tempy = y0;
            s0 = s0 + p0 * (((!y0) * y0).p[0][0] /
                 ((!(A * p0)) * p0).p[0][0]);
            y0 = y0 - A * p0 * (((!y0) * y0).p[0][0]
                  / ((!(A * p0)) * p0).p[0][0]);
            p0 = y0 + p0 * ((((!y0) * y0).p[0][0]) /
                  ((!tempy) * tempy).p[0][0]);
        }
    }
    return s0;
}
```

3.2 SPCG 法

为了提升在负载因子 ρ 较大的时候 CG 算法的表现情况以及减少算法对内存的需要。SPCG 算法被提出，SPCG 算法过程如下:

Algorithm 1 Proposed SPCG Algorithm

Input:

Matrix $\mathbf{A}$, vector $\tilde{\mathbf{y}}$

1: Compute B by Eq. .

2: $\hat{\mathbf{s}}^{(0)} = 0, \tilde{\mathbf{y}}^{(0)} = \mathbf{B}\tilde{\mathbf{y}}, \tilde{\mathbf{p}}^{(0)} = \tilde{\mathbf{y}}^{(0)}$

3: for $i = 1, \ldots, k$ **do**

4: $\hat{\mathbf{s}}^{(i)} = \hat{\mathbf{s}}^{(i-1)} + \frac{\left(\tilde{\mathbf{y}}^{(i-1)} \cdot \tilde{\mathbf{y}}^{(i-1)}\right)}{\left(\mathbf{BAB}\tilde{\mathbf{p}}^{(i-1)} \cdot \tilde{\mathbf{p}}^{(i-1)}\right)}\mathbf{B}\tilde{\mathbf{p}}^{(i-1)}$

5: $\tilde{\mathbf{y}}^{(i)} = \tilde{\mathbf{y}}^{(i-1)} - \frac{\left(\tilde{\mathbf{y}}^{(i-1)} \cdot \tilde{\mathbf{y}}^{(i-1)}\right)}{\left(\mathbf{BAB}\tilde{\mathbf{p}}^{(i-1)} \cdot \tilde{\mathbf{p}}^{(i-1)}\right)}\mathbf{BAB}\tilde{\mathbf{p}}^{(i-1)}$

6: $\tilde{\mathbf{p}}^{(i)} = \tilde{\mathbf{y}}^{(i)} + \frac{\left(\tilde{\mathbf{y}}^{(i)} \cdot \tilde{\mathbf{y}}^{(i)}\right)}{\left(\tilde{\mathbf{y}}^{(i-1)} \cdot \tilde{\mathbf{y}}^{(i-1)}\right)}\tilde{\mathbf{p}}^{(i-1)}$

7: **end for**

Output:

$\hat{\mathbf{s}} = \hat{\mathbf{s}}^{(k)}$

其中

$$\mathbf{B}_{(i,i)} = \frac{1}{\sqrt{\mathbf{A}_{(i,i)}}}$$

在算法基础上编写程序如下：

```
complex_matrix SPCG(complex_matrix H_hat,
     complex_matrix y_hat, double s,int iteration)
{
    double eps = 1e-5;
    complex_matrix y_1 = (!H_hat) * y_hat;
    complex_matrix A = ((!H_hat) * H_hat +
         complex_matrix::eyes(H_hat.cols_num) * s);
    complex_matrix s0(H_hat.cols_num, 1);
    complex_matrix B(A.rows_num, A.cols_num);
    for (size_t i = 0; i < B.rows_num; i++)
    {
        B.p[i][i] =Complex( 1.0 / sqrt(A.p[i][i].
             real),0);//HT*H虚部为零0
    }

    complex_matrix y0 = B*y_1;
    complex_matrix p0 = y0;
    complex_matrix tempy = y0;
    for (size_t i = 0; i < iteration; i++)
    {
        if ((tempy - y0).norm2(0) <= eps && i != 0)
        {
            break;
        }
        else{
        tempy = y0;
```

```
        s0 = s0 + B * p0 * (y0.cdot() / (B * A * B *
             p0).cdot(p0));
        y0 = y0 - B * A * B * p0 * (y0.cdot() / (B *
             A * B * p0).cdot(p0));
        p0 = y0 + p0 * (y0.cdot() / tempy.cdot());
        }
    }
    return s0;
}
```

3.3 算法比较

在不同迭代次数下比较 CG 算法和 SPCG 算法的误码率并绘制对应曲线，得到的图像如下:

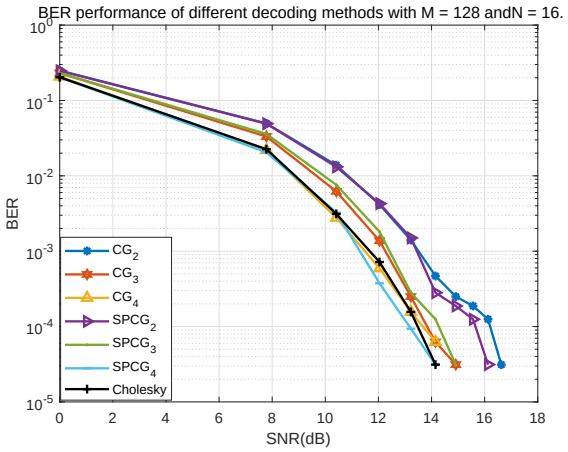

图 4: 当 M=128，N=16 时不同算法不同迭代次数下 CG 和 SPCG 算法的误码率

结果分析 从图中可以看出在不同迭代次数下 CG 和 SPCG 算法误码率相近，说明两个算法准确度相接近。而随着迭代次数增加，两个算法的误码率都下降，符合趋势。除此之外 Cholesky 分解的误码率在此条件下和三次迭代下的 CG 和 SPCG 算法接近，

更改发射天线数目至 N=32 进行测试，结果如下：

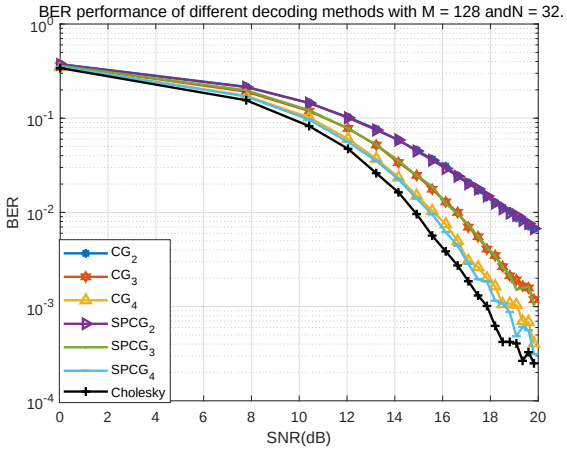

图 5: 当 M=128，N=32 时不同算法不同迭代次数下 CG 和 SPCG 算法的误码率

结果分析 从图中可以看出在 N=32 时结果和 N=16 时相近，不同迭代次数下 CG 和 SPCG 算法误码率相近，两个算法准确度相接近。而随着迭代次数增加，两个算法的误码率都下降，符合趋势。除此之外可以看出 Cholesky 分解的误码率低于另外两个算法。

4 文献四

对于方程线程方程组方程

$$\mathbf{Ax} = \mathbf{b}$$

文献四中给出了多种迭代算法，本作业取了如下的几种方法进行测试。

4.1 雅可比方法

在矩阵分裂 $\boldsymbol{A} = \boldsymbol{M} - \boldsymbol{N}$ 中取 $\boldsymbol{M}$ 为对角阵, 且其元素是 $\boldsymbol{A}$ 的对角元是一种最简单的方法. 设 D 是与 A 的对角元相同的对角阵,L 和 U 分别是 A 的严格下三角和严格上三角部分, 则

$$\boldsymbol{M} = \boldsymbol{D} \quad \boldsymbol{N} = -(\boldsymbol{L} + \boldsymbol{U})$$

在此条件下雅可比迭代的格式为：

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{D}^1 \left(\boldsymbol{b} - (\boldsymbol{L} + \boldsymbol{U})\boldsymbol{x}^{(k)}\right)$$

在此基础上编写程序如下所示：

```
Matrix Jacobi(complex_matrix H_hat, complex_matrix
    y_hat, double s=1.0, int iteration=1)
{
```

```
    double eps = 1e-6;
    int N = 2 * y_hat.rows_num;
    int K = 2 * H_hat.cols_num;
    //complex_matrix temp_y(2 * y.rows_num, 1);
    Matrix R_y = y_hat.realmatrix();//实部矩阵
    Matrix I_y = y_hat.imagmatrix();//虚部矩阵

    Matrix R_H = H_hat.realmatrix();
    Matrix I_H = H_hat.imagmatrix();

    Matrix y = R_y.combine_rows(R_y, I_y);

    Matrix H = R_H.combine_rows(R_H, I_H);
    Matrix temp_imagpart = Matrix(H_hat.rows_num,
        H_hat.cols_num) - I_H;//负的H的虚部矩阵
    Matrix temp = temp_imagpart.combine_rows(
        temp_imagpart, R_H);
    H = H.combine_cols(H, temp);//最终得到H
    Matrix A = Matrix::T(H) * H + Matrix::eye(K) *
        s;
    Matrix b = Matrix::T(H) * y;


    Matrix D(A.rows_num, A.cols_num);
    Matrix N_1(A.rows_num, A.cols_num);
    for (size_t i = 0; i <A.rows_num; i++)
    {
        D.p[i][i] = A.p[i][i];
    }
    N_1 = D - A;
    Matrix s0 = generate_signal(K).realmatrix();//
        产生一个随机的s0
    Matrix temps = s0;
    for (size_t i = 0; i < 100; i++)
    {
        if ((temps - s0).norm2(0) <= eps && i != 0)
        {
            break;
        }
        else
        {
            temps = s0;
            s0 = Matrix::inv(D) * (b + N_1 * s0);
        }
    }
    return s0;
}
```

程序中先将复数矩阵按照文献 1 中的方式转化为实数矩阵问题，随后通过雅可比迭代得到最终的结果，在迭代过程中通过迭代前后 s 向量相差得到的向量的范数大小来终止迭代。

4.2 高斯赛德尔方法

雅可比方法收敛速度缓慢，原因在于是它在迭代过程中并没有利用最新的信息，新的分量值只有在整个扫描过程全部完成才能被利用. 而高斯-塞德尔方法弥补了这一缺陷，一旦某个分量的新值计算出来马上将它利用。高斯赛德尔方法的迭代格式为：

$$\begin{aligned}\boldsymbol{x}^{(k+1)} &= \boldsymbol{D}^{-1}\left(\boldsymbol{b}-\boldsymbol{L}\boldsymbol{x}^{(k+1)}-\boldsymbol{U}\boldsymbol{x}^{(k)}\right)\\ &= (\boldsymbol{D}+\boldsymbol{L})^{1}\left(\boldsymbol{b}-\boldsymbol{U}\boldsymbol{x}^{(k)}\right)\end{aligned}$$

对应的迭代格式为：

$$\boldsymbol{M}=\boldsymbol{D}+\boldsymbol{L},\quad \boldsymbol{N}=-\boldsymbol{U}$$

对于高斯赛德尔方法编写程序如下：

```
Matrix Gauss(complex_matrix H_hat, complex_matrix
    y_hat, double s = 1.0, int iteration = 1)
{
    double eps = 1e-6;
    int N = 2 * y_hat.rows_num;
    int K = 2 * H_hat.cols_num;
    //complex_matrix temp_y(2 * y.rows_num, 1);
    Matrix R_y = y_hat.realmatrix();//实部矩阵
    Matrix I_y = y_hat.imagmatrix();//虚部矩阵

    Matrix R_H = H_hat.realmatrix();
    Matrix I_H = H_hat.imagmatrix();

    Matrix y = R_y.combine_rows(R_y, I_y);

    Matrix H = R_H.combine_rows(R_H, I_H);
    Matrix temp_imagpart = Matrix(H_hat.rows_num,
        H_hat.cols_num) - I_H;//负的H的虚部矩阵
    Matrix temp = temp_imagpart.combine_rows(
        temp_imagpart, R_H);
    H = H.combine_cols(H, temp);//最终得到H
    Matrix A = Matrix::T(H) * H + Matrix::eye(K) *
        s;
    Matrix b = Matrix::T(H) * y;

    Matrix D(A.rows_num, A.cols_num);
    Matrix L(A.rows_num, A.cols_num);
    Matrix U(A.rows_num, A.cols_num);
    Matrix s0 = generate_signal(K).realmatrix();//
        产生一个随机的s0
    Matrix temps = s0;
    for (size_t i = 0; i < A.rows_num; i++)
    {
        D.p[i][i] = A.p[i][i];
    }
    for (size_t i = 0; i < A.rows_num; i++)
    {
        for (size_t j = 0; j < A.cols_num; j++)
```

```
        {
            if (i > j)
            {
                L.p[i][j] = A.p[i][j];
            }
            else if (i < j)
            {
                U.p[i][j] = A.p[i][j];
            }
        }
    }
    for (size_t i = 0; i < 100; i++)
    {
        if ((temps - s0).norm2(0) <= eps && i != 0)
        {
            break;
        }
        else
        {
            temps = s0;
            s0 = Matrix::inv(D + L) * (b - U * s0);
        }
    }
    return s0;

}
```

程序与雅可比方法基本一致而只在迭代格式不同。高斯赛德尔方法收敛速度更快且不需要重复储存解响亮的值。

4.3 逐次超松弛迭代

逐次超松弛 (SOR) 技术可以加快高斯-塞德尔方法的收敛速度，这种方法以下一步高斯-塞德尔迭代的步长作为搜索方向, 加上一个固定的用 ω 表示的搜索参数. 具体做法是从初始解向量出发，首先用高斯-塞德尔方法计算下一步的迭代值 x_GS，然后取下一次迭代值为:

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \omega\left(\boldsymbol{x}_{\text{is}}^{(k+1)} - \boldsymbol{x}^{(k)}\right).$$

ω 相当于一个固定的起加速收敛作用的松弛参数.ω>1 为超松弛,ω<1 为低松弛 (ω=1 就是高斯-塞德尔方法), 通常总是取 0<w<2(否则方法发散)。

在矩阵形式下超松弛迭代可以写为

$$\begin{aligned}\boldsymbol{x}^{(k+1)} &= \boldsymbol{x}^{(k)} + \omega\left(\boldsymbol{D}^{-1}\left(\boldsymbol{b} - \boldsymbol{L}\boldsymbol{x}^{(k+1)} - \boldsymbol{U}\boldsymbol{x}^{(k)}\right) - \boldsymbol{x}^{(k)}\right)\\ &= (\boldsymbol{D}+\omega\boldsymbol{L})^{-1}((1-\omega)\boldsymbol{D} - \omega\boldsymbol{U})\boldsymbol{x}^{(k)}\\ &\quad + \omega(\boldsymbol{D}+\omega\boldsymbol{L})^{-1}\boldsymbol{b},\end{aligned}$$

相应的分裂为

$$\boldsymbol{M} = \frac{1}{\omega}\boldsymbol{D} + \boldsymbol{L}, \quad \boldsymbol{N} = \left(\frac{1}{\omega} - 1\right)\boldsymbol{D} - \boldsymbol{U}.$$

对应超松弛迭代，只需在上述程序中迭代格式改为：

```
s0 = Matrix::inv(D + L * omega) * (D * (1 - omega)
    - U * omega) * s0 + Matrix::inv(D + L * omega)
     * b * omega;
```

即可。

4.4 算法测试

4.4.1 SOR

测试 ω 对误码率影响 对于 SOR 方法，其松弛参数会影响算法的收敛速度以及误码率。我们可以在 SNR=10dB 的条件下测试不同 ω 对算法误码率的影响，其结果如下所示结果如下：图中的 n 为迭代

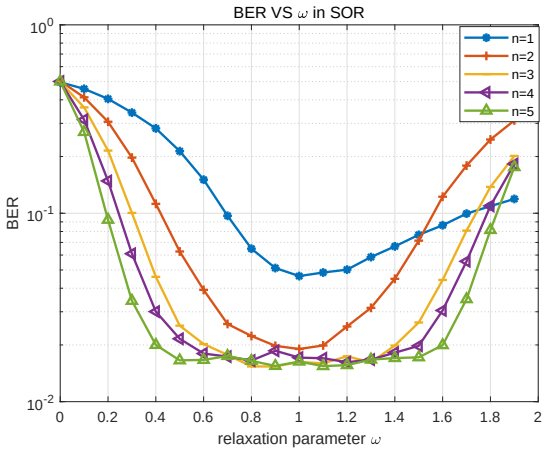

图 6: 当 M=128，N=16，SNR=10dB 时 ω 影响

次数，系统大小为 16*128. 由于每次测试都重新生成了信道矩阵 H，因此最佳松弛参数有所不同。但是都能从图中看到 SOR 方法存在一个最佳松弛参数可以使得误码最低同时收敛速度快。

4.4.2 高斯赛德尔及雅可比方法

对于高斯赛德尔方法和雅可比方法我们可以测试其在不同误码率以及不同迭代次数下的误码率情形。其结果如下 从图中可以看出两种方法的表现

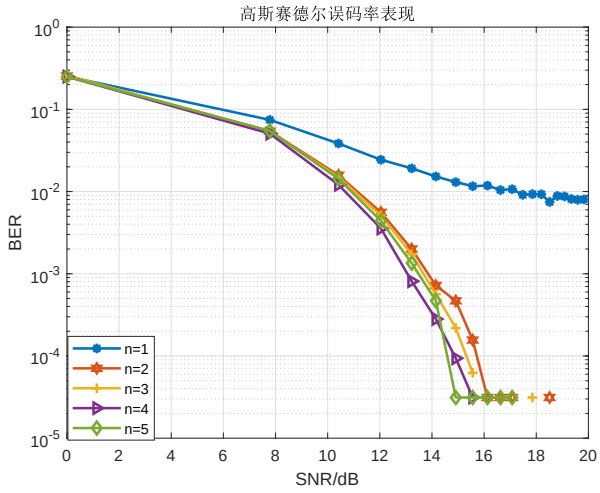

图 7: 不同 SNR 及迭代次数下高斯法误码率表现

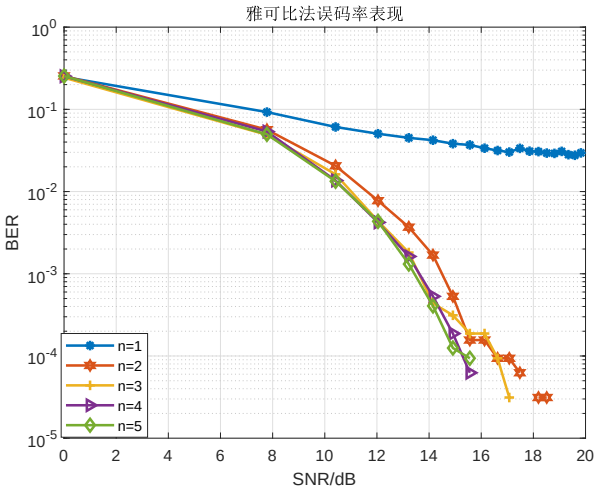

图 8: 不同 SNR 及迭代次数下雅可比法误码率表现

十分类似，两者的误码率随着迭代次数和信噪比增加了减小。