矩阵分析与应用大作业-梁奥-202128014728021

2021 年 11 月 29 日

目录

1 引言

本项目严格依据 Python 库文件的编写要求编写，所有功能实现的程序都储存在 factorization 文件夹中，实例的所有功能都封装在 *Matrix* 对象中。从外部调用可实现程序的功能，封装的矩阵功能有：矩阵行阶梯表示、矩阵的秩、矩阵的零空间、矩阵的值空间、矩阵的 (P)LU 分解、矩阵的逆、Gram-Schmidt 正交化、Householder 正交约简、Givens 约简、URV 分解还有基于这些功能实现的矩阵的行列式和 $Ax = b$ 的线性系统的求解。

实现过程中，除了必要的如 “矩阵表达->numpy”，“数据拷贝->copy” 用了相应的第三方库外，其余均从头实现。在程序文件中，也都清楚的注释了各个功能功能介绍和实现方式。对于项目中各对象的继承过程，功能的传递过程详情见./factorization/___init___.py。

为更进一步的完善项目，增加调试的便捷性，同时设计了项目实现的 GUI，实现程序为 MatrixGUI.py，在 GUI 中可以更灵活的进行功能测试。下面将具体阐述项目的环境要求和测试过程，最后将使用 numpy，sympy 中成熟的 Python 运算函数进行同样的运算，对项目中的设计的功能进行准确度检验，确保设计程序的正确性和精度。

1.1 运行环境及编译工具

- Windows
- VS Code

1.2 编程语言及库版本

库	版本
Python	3.7.0
copy	无
numpy	1.19.2
PyQt5	5.15.4

1.3 文件树

1.3.1 程序库

- factorization
  - ___init___.py
  - lu_factorization.py
  - matrix.py
  - matrixtools.py
  - qr_factorization.py

– urv_factorization.py

1.3.2 测试程序

任选下其一测试，其中 test.ipynb 可能会更清楚明白一些

- test.py
- test.ipynb

1.3.3 GUI

- MatrixGUI.py

2 功能测试

2.1 矩阵的行阶梯表示->rref()

2.1.1 设计思路

主要是运用高斯约旦法，将主元置 1，并将主元所在列的其他元素置零，实现过程运用了矩阵基本运算的后两种。需要注意的有

- 在进行基本运算时，要杜绝分母是零的情况
- 由于计算要求，矩阵格式应是 float 类型，但由于 numpy 对浮点数计算精度过高经常出现极小但不为零的情况，如 $1.230512e-16$，此时应判断后置零，否则会极大的影响后面对于矩阵秩的判断

2.1.2 测试

```
In [15]: # 所有功能均封装在 Matrix 中
         from factorization import Matrix
         import numpy as np
         a = np.array([[1,1,2,2,1],
                       [2,2,4,4,3],
                       [2,2,4,4,2],
                       [3,5,8,6,5]])
         # 实例化对象
         test = Matrix(a)
         # 调用 rref()
         test.rref()
```

```
Out[15]: array([[ 1.,  0.,  1.,  2.,  0.],
                [ 0.,  1.,  1.,  0.,  0.],
                [-0., -0., -0., -0.,  1.],
                [ 0.,  0.,  0.,  0.,  0.]])
```

2.2 矩阵的秩->rank()

2.2.1 基本思路

由矩阵的行阶梯表示可以很容易的得到矩阵的秩，只用判断每行中是否所有元素都是 0，所以上述对极小的判断就非常重要

2.2.2 测试

```
In [16]: # 接着使用上面的实例化对象
         # 调用 rank()
         test.rank()
```

```
Out[16]: 3
```

2.3 矩阵的零空间->nullspace()

2.3.1 基本思路

零空间即为 $Ax = 0$ 中的 x，由于得出了矩阵的行阶梯表示和矩阵的秩，当矩阵是满秩阵时零空间只有零元素，非满秩矩阵时需找出自由变量，并依据行阶梯表示进行计算

2.3.2 测试

```
In [17]: # 接着使用上面的实例化对象
         # 调用 nullspace()
         test.nullspace()
```

```
Out[17]: array([[-1., -2.],
                [-1., -0.],
                [ 1.,  0.],
                [ 0.,  1.],
                [ 0.,  0.]])
```

2.4 矩阵的值空间->columnspace()

2.4.1 基本思路

值空间是矩阵的列空间，是主元所在的列，当矩阵是满秩矩阵时值空间就是它本身，否则由行阶梯表示得到的主元所在的列也可求出

2.4.2 测试

```
In [18]: # 接着使用上面的实例化对象
         # 调用 columnspace()
         test.columnspace()
```

```
Out[18]: array([[2., 2., 1.],
                [4., 4., 3.],
                [4., 4., 2.],
                [8., 8., 5.]])
```

2.5 矩阵的 (P)LU 分解 ->PLU_factorization()

2.5.1 基本思路

矩阵的 LU 分解主要是高斯约简的过程，PLU 的不同之处是将过程中的最大元素挪到上方作为主元，并在 P 矩阵中存储移动过程，在项目中将 P 作为缺省参数，默认进行 PLU 分解，当 P=False 时进行 LU 分解，需要注意有

- 分解的矩阵需要是非奇异阵，否则分母元素可能为零

2.5.2 测试

```
In [19]: # 在上面已经调用过库，在这里只更改测试矩阵 a
         # from factorization import Matrix
         # import numpy as np
         a = np.array([[1,2,-3,4],
                       [4,8,12,-8],
                       [2,3,2,1],
                       [-3,-1,1,-4]])
         # 实例化对象
         test = Matrix(a)
         # PLU_factorization()()
         test.PLU_factorization()
```

```
Out[19]: (array([[0., 1., 0., 0.],
                 [0., 0., 0., 1.],
```

```
        [1., 0., 0., 0.],
        [0., 0., 1., 0.]], dtype=float32),
array([[ 1.        ,  0.        ,  0.        ,  0.        ],
       [-0.75      ,  1.        ,  0.        ,  0.        ],
       [ 0.25      ,  0.        ,  1.        ,  0.        ],
       [ 0.5       , -0.2       ,  0.33333334,  1.        ]],
      dtype=float32),
array([[  4.,   8.,  12.,  -8.],
       [  0.,   5.,  10., -10.],
       [  0.,   0.,  -6.,   6.],
       [  0.,   0.,   0.,   1.]], dtype=float32))
```

2.6 矩阵的逆->get_inversion()

2.6.1 基本思路

由矩阵的 PLU 分解可以有效处理 $Ax = b$ 的情况，当 b 分别是单位阵的每一列时，我们求得的 x 就是矩阵 A^{-1} 的每一列

2.6.2 测试

```
In [22]: # 接着使用上面的实例化对象
         # get_inversion()
         test.get_inversion()
```

```
Out[22]: array([[ 0.16666669,  0.25833336, -1.        , -0.6       ],
                [ 0.3333333 ,  0.06666665,  0.        ,  0.2       ],
                [-0.5       , -0.22499998,  1.        ,  0.2       ],
                [-0.33333334, -0.26666665,  1.        ,  0.2       ]],
               dtype=float32)
```

2.7 Gram-Schmidt 正交化->Gram_Schmidt_orthogonalization()

2.7.1 基本思想

$$\mathbf{q}_1 = \frac{\mathbf{a}_1}{\nu_1} \quad \text{and} \quad \mathbf{q}_k = \frac{\mathbf{a}_k - \sum_{i=1}^{k-1} \langle \mathbf{q}_i \mid \mathbf{a}_k \rangle \, \mathbf{q}_i}{\nu_k} \quad \text{for } k = 2, 3, \ldots, n$$

$$\nu_1 = \|\mathbf{a}_1\| \qquad \nu_k = \left\| \mathbf{a}_k - \sum_{i=1}^{k-1} \langle \mathbf{q}_i \mid \mathbf{a}_k \rangle \, \mathbf{q}_i \right\| \qquad k > 1$$

$$(\mathbf{a}_1 \,|\mathbf{a}_2|\cdots|\, \mathbf{a}_n) = (\mathbf{q}_1 \,|\mathbf{q}_2|\cdots|\, \mathbf{q}_n) \begin{pmatrix} \nu_1 & \langle \mathbf{q}_1 \mid \mathbf{a}_2 \rangle & \langle \mathbf{q}_1 \mid \mathbf{a}_3 \rangle & \cdots & \langle \mathbf{q}_1 \mid \mathbf{a}_n \rangle \\ 0 & \nu_2 & \langle \mathbf{q}_2 \mid \mathbf{a}_3 \rangle & \cdots & \langle \mathbf{q}_2 \mid \mathbf{a}_n \rangle \\ 0 & 0 & \nu_3 & \cdots & \langle \mathbf{q}_3 \mid \mathbf{a}_n \rangle \\ \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & \nu_n \end{pmatrix}$$

需要注意的是：Gram-Schmidt 正交化中的矩阵应是列线性无关的

2.7.2 测试

```
In [23]: # 在上面已经调用过库，在这里只更改测试矩阵 a
         # from factorization import Matrix
         # import numpy as np
         a = np.array([[0,-20,-14],
                       [3,27,-4],
                       [4,11,-2]])
         # 实例化对象
         test = Matrix(a)
         # Gram_Schmidt_orthogonalization()
         test.Gram_Schmidt_orthogonalization()

Out[23]: (array([[ 0.  , -0.8 , -0.6 ],
                 [ 0.6 ,  0.48, -0.64],
                 [ 0.8 , -0.36,  0.48]]), array([[ 5., 25., -4.],
                 [ 0., 25., 10.],
                 [ 0.,  0., 10.]]))
```

2.8 Householder reduction–>Householder_reduction()

2.8.1 基本思想

基于反射矩阵的思想，逐步的将每一列（子矩阵）反射到第一坐标轴上

2.8.2 测试

```
In [24]: # 接着使用上面的实例化对象
         # Householder_reduction()
         test.Householder_reduction()
```

```
Out[24]: (array([[ 0.  ,  0.6 ,  0.8 ],
                 [-0.8 ,  0.48, -0.36],
                 [-0.6 , -0.64,  0.48]]),
          array([[ 5.00000000e+00,  2.50000000e+01, -4.00000000e+00],
                 [ 0.00000000e+00,  2.50000000e+01,  1.00000000e+01],
                 [ 0.00000000e+00, -1.33226763e-15,  1.00000000e+01]]))
```

2.9 Givens reduction–>Givens_reduction()

2.9.1 基本思想

利用旋转矩阵的思想，逐步将元素（子矩阵）旋转到第一坐标轴上

$$\mathbf{P}_{ij} = \begin{pmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & c & & s & & & \\ & & & 1 & & & & \\ & & & & \ddots & & & \\ & & -s & & c & & & \\ & & & & & 1 & & \\ & & & & & & \ddots & \\ & & & & & & & 1 \end{pmatrix}$$

2.9.2 测试

```
In [25]: # 接着使用上面的实例化对象
         # Givens_reduction()
         test.Givens_reduction()
```

```
Out[25]: (array([[ 0.  ,  0.6 ,  0.8 ],
                 [-0.8 ,  0.48, -0.36],
                 [-0.6 , -0.64,  0.48]]),
          array([[ 5.00000000e+00,  2.50000000e+01, -4.00000000e+00],
                 [ 2.66453526e-16,  2.50000000e+01,  1.00000000e+01],
                 [-3.55271368e-16, -3.10862447e-16,  1.00000000e+01]]))
```

2.10 URV 分解–>URV_factorization()

2.10.1 基本思想

Solution: Apply Householder (or Givens) reduction to produce an orthogonal matrix $\mathbf{P}_{m\times m}$ such that $\mathbf{PA} = \begin{pmatrix}\mathbf{B}\\ \mathbf{0}\end{pmatrix}$, where $\mathbf{B}$ is $r \times n$ of rank r. Householder (or Givens) reduction applied to $\mathbf{B}^T$ results in an orthogonal matrix $\mathbf{Q}_{n\times n}$ and a nonsingular upper-triangular matrix $\mathbf{T}$ such that

$$\mathbf{QB}^T = \begin{pmatrix}\mathbf{T}_{r\times r}\\ \mathbf{0}\end{pmatrix} \implies \mathbf{B} = (\mathbf{T}^T \,|\, \mathbf{0})\mathbf{Q} \implies \begin{pmatrix}\mathbf{B}\\ \mathbf{0}\end{pmatrix} = \begin{pmatrix}\mathbf{T}^T & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{pmatrix}\mathbf{Q},$$

so $\mathbf{A} = \mathbf{P}^T\begin{pmatrix}\mathbf{B}\\ \mathbf{0}\end{pmatrix} = \mathbf{P}^T\begin{pmatrix}\mathbf{T}^T & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{pmatrix}\mathbf{Q}$ is a URV factorization.

2.10.2 测试

```
In [26]: # 接着使用上面的实例化对象
         # URV_factorization()
         test.URV_factorization()
```

```
Out[26]: (array([[ 0.  , -0.8 , -0.6 ],
                 [ 0.6 ,  0.48, -0.64],
                 [ 0.8 , -0.36,  0.48]]),
          array([[ 2.58069758e+01,  1.36696280e-16, -8.77596167e-16],
                 [ 2.26682896e+01,  1.45309548e+01, -1.43603192e-16],
                 [-1.54996852e+00,  9.29981110e+00,  3.33333333e+00]]),
          array([[ 0.19374606,  0.96873032, -0.15499685],
                 [-0.30224386,  0.20924575,  0.92998111],
                 [ 0.93333333, -0.13333333,  0.33333333]]))
```

2.11 矩阵的行列式->det()

2.11.1 基本思想

由于只有方阵才有行列式，所以我们可以依靠 PLU 分解来实现求解矩阵的行列式。在计算之前判断矩阵是否满秩，是则用 PLU 分解求解，不是时返回 0，因为非奇异阵的行列式为 0.

2.11.2 测试

```
In [27]: # 接着使用上面的实例化对象
         # det()
         test.det()
```

```
Out[27]: 1250.0
```

2.12 $Ax = b$ 求解

2.12.1 基本思想

- 当 $b = 0$ 时
  - 当 A 为满秩矩阵时，返回零元素
  - 当 A 非满秩时，有无数解
  - 可以调用上面的 nullspace() 函数返回 A 的零空间
- 当 $b \neq 0$
  - 当 $rank(A) \neq rank([A|b])$ 时，无解
  - 当 $rank(A) = rank([A|b])$ 时
    - 当 $rank(A) = rank([A|b]) = ncol$ 时，有唯一解，用 PLU 分解求解
    - 否则，有无数解，由通解和特解组成

2.12.2 测试

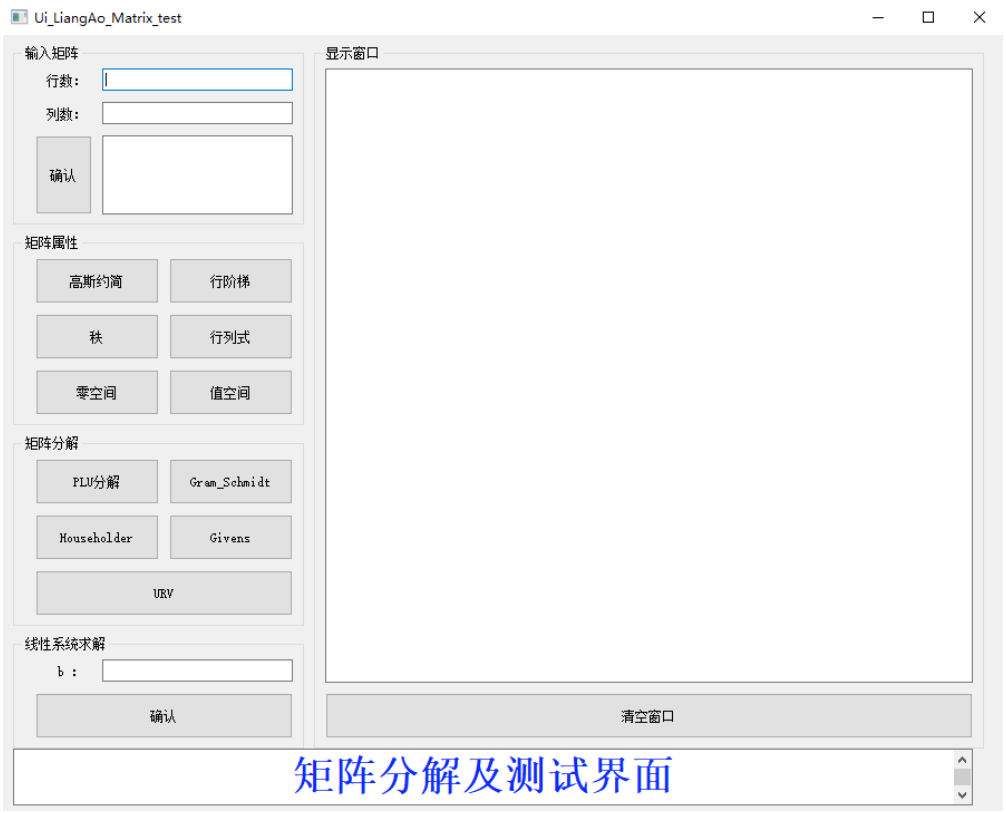
```
In [28]: # 在上面已经调用过库，在这里只更改测试矩阵 a
         # from factorization import Matrix
         # import numpy as np
         a = np.array([[1,1,2,2,1],
                       [2,2,4,4,3],
                       [2,2,4,4,2],
                       [3,5,8,6,5]])
         b = np.array([1,1,2,3])
         # 实例化对象
         test = Matrix(a)
         # linear_equation()
         test.linear_equation(b)
         # 返回前半部分是特解，后半部分通解
```

```
Out[28]: [array([ 1.,  1.,  0.,  0., -1.]), array([[-1., -2.],
                 [-1., -0.],
                 [ 1.,  0.],
                 [ 0.,  1.],
                 [ 0.,  0.]])]
```

3 GUI 测试–>MatrixGUI.py

3.1 启动

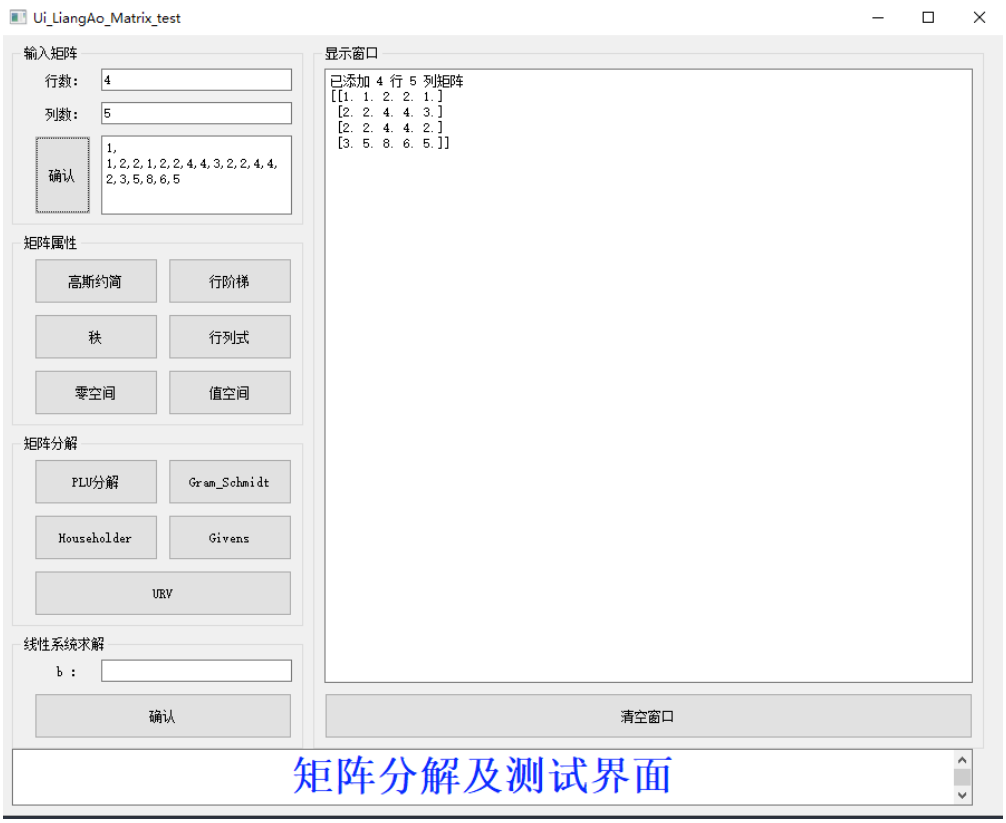
```
In [35]: # jupyter notebook 中启动
         %run MatrixGUI.py
         # 外部启动
         # python MatrixGUI.py
```



图一：GUI 界面

3.2 输入矩阵

矩阵采用输入行，列，和矩阵元素进行输入，如图二所示



图二：输入矩阵

3.3 功能测试

点击相应按钮进行功能测试，如图三所示

图三：功能测试

4 功能验证

为保证程序功能的正确性，本项目也使用 numpy.linalg，sympy 等第三方库进行了对比验证，由于验证的篇幅较大，本报告中仅作部分展示，完整验证过程将放在 test.py 和 test.ipynb 文件中。

4.1 矩阵的秩

随机生成 5000 个 5x4 的 0/1 矩阵求秩，用项目中的函数与 numpy 库中功能函数的做对比

```
In [41]: error = 0
for i in range(5000):
#     随机生成 0/1 矩阵
mat = np.random.randint(0,2,(5,4))
#     实例化对象
test = Matrix(mat)
#     调用 rank() 函数
rank1 = test.rank()
#     调用 numpy 中的函数
```

```
rank2 = np.linalg.matrix_rank(mat)
if rank1 != rank2:
print("出现错误，错误的矩阵是\n{}".format(mat))
error += 1
if error == 0:
print("未出现错误，测试样本{}个".format(5000))
```

```
未出现错误，测试样本 5000 个
```

4.2 URV 分解

随机生成 5000 个 5x3 矩阵进行 URV 分解，并计算是否 $UU^T = I, RR^T = I, A = URV$，是则认为程序正确

```
In [33]: num = 0
error = 0
while(num < 5000):
#     随机生成矩阵
mat = np.random.randint(0,11,(5,3))
#     实例化对象
test = Matrix(mat)
#     Givens_reduction() 函数
U,R,V = test.URV_factorization()
#     设置四位有效数字的精度
if np.any(np.round(U@U.T,decimals=3) != np.eye(5)) and np.any(np.round(V@V.T,decimals=
print("mat = {}".format(mat))
print("Error!!!")
error += 1
num += 1
if error == 0:
print("未出现错误，测试样本{}个".format(5000))
```

```
未出现错误，测试样本 5000 个
```

4.3 矩阵的行列式

随机生成 5000 个 5x5 的矩阵求行列式，用项目中的函数与 numpy 库中功能函数的做对比，展示前十个矩阵的计算结果

```
In [46]: error = 0
for i in range(5000):
#     随机生成矩阵
mat = np.random.randint(-10,10,(5,5))
#     实例化对象
test = Matrix(mat)
#     调用 det() 函数
det1 = test.det()
#     调用 numpy 中的函数
det2 = np.linalg.det(mat)
if np.round(det1) != np.round(det2):
print("出现错误，错误的矩阵是\n{}".format(mat))
error += 1
if i < 10:
print("====================={}====================".format(i))
print("本项目函数的计算结果：{}".format(det1))
print(" numpy 库的计算结果：{}".format(det2))
if error == 0:
print("未出现错误，测试样本{}个".format(5000))
```

```
====================0====================
本项目函数的计算结果：41113.998479636626
numpy 库的计算结果：41113.99999999996
====================1====================
本项目函数的计算结果：-81923.0020582776
numpy 库的计算结果：-81922.99999999994
====================2====================
本项目函数的计算结果：-19559.99929034382
numpy 库的计算结果：-19559.999999999996
====================3====================
本项目函数的计算结果：-23727.000239994315
numpy 库的计算结果：-23727.00000000002
====================4====================
本项目函数的计算结果：7476.000406625426
numpy 库的计算结果：7476.00000000001
====================5====================
本项目函数的计算结果：-6804.000084235043
```

```
numpy 库的计算结果：-6803.9999999999945
====================6==================
本项目函数的计算结果：-387976.0106603415
numpy 库的计算结果：-387975.99999999994
====================7==================
本项目函数的计算结果：-10193.000293473218
numpy 库的计算结果：-10193.000000000004
====================8==================
本项目函数的计算结果：4458.999891225982
numpy 库的计算结果：4458.999999999992
====================9==================
本项目函数的计算结果：-9183.000072946019
numpy 库的计算结果：-9182.999999999989
未出现错误，测试样本 5000 个
```

5 致谢

谢谢李保滨老师这两个多月的陪伴，让我们学到了很多，最后的大作业希望能够给《矩阵分析与应用》画上圆满的句号，祝老师身体健康，工作顺利。如果看这篇文档是哪位师兄或师姐的话也祝师兄师姐学习顺利多发文章!!

@Time : 2021/11/26 22:49
@Author : Liang Ao
@Software: VS Code
@Github: https://github.com/AlanLiangC
@Blog : https://AlanLiangC.github.io