武漢大學

课 程 设 计 报 告

矩阵分解

姓　　名：朱鹤然

学　　号：2021202120085

任课教师：王文伟

学　　院：电子信息学院

专　　业：信息与通信工程

二〇二一年十一月

Zhu Heran

说　　明

代码已经上传至 Github（https://github.com/HenryZhuHR/Matrix-Theory-Assignment），代码分为

- Matlab 版本: https://github.com/HenryZhuHR/Matrix-Theory-Assignment/tree/main/matlab
- C++ 版本：https://github.com/HenryZhuHR/Matrix-Theory-Assignment/tree/main/cpp

其中，C++ 采用开源的线性代数和科学计算库 Eigen3(3.4.0) 进行矩阵运算，C++ 代码采用 CMake(3.21.4) 构建项目，Clang(13.0.0) 作为编译器

目　　录

1 LU 分解

1.1 LU 分解简介

LU 分解能够将 $m \times m$ 的满矩阵或稀疏矩阵 A 分解成为一个 $m \times m$ 上三角矩阵 U 和一个 $m \times m$ 经过置换的下三角矩阵 L，满足

$$A = LU \tag{1.1}$$

另一种 LU 分解的形式是将 A 分解成为一个置换矩阵 P、一个 $m \times m$ 上三角矩阵 R 和一个 $m \times m$ 下单位三角矩阵 Q

$$A = P^T LU \tag{1.2}$$

LU 分解需要满足如下条件：

- 矩阵是方阵
- 矩阵是可逆的，即矩阵是满秩矩阵，每一行都是独立向量
- 消元过程中没有 0 主元出现，也就是消元过程中不能出现行交换的初等变换。

1.2 LU 分解的 Matlab 实现及结果验证

用 Matlab 自带的函数 `rand()` 创建一个大小为 5×5 的随机矩阵

```
>> A=rand(5,5)*20
A =
     14.2243    8.4833    0.5844    4.7457    4.6319
      4.4349   10.1572   18.5771    9.1770    9.7780
      2.3484    1.7103   14.6066   19.2618   12.4812
      5.9335    5.2496    9.7722   10.9361   13.5827
      6.3756   16.0203   11.5705   10.4227    7.9103
```

向下取整后作为项目测试用的矩阵 A

$$A = \begin{bmatrix} 14 & 8 & 0 & 4 & 4 \\ 4 & 10 & 18 & 9 & 9 \\ 2 & 1 & 14 & 19 & 12 \\ 5 & 5 & 9 & 10 & 13 \\ 6 & 16 & 11 & 10 & 7 \end{bmatrix} \tag{1.3}$$

随机产生列向量 b

```
>> b=rand(5,1)*20
b =
   12.9798
   16.0066
    9.0760
    8.6478
   16.5063
```

取 $b = [12, 16, 9, 8, 16]$

设置矩阵 A

```
>> A = [
    14, 8,  0,  4,  4;
    4, 10, 18,  9,  9;
    2,  1, 14, 19, 12;
    5,  5,  9, 10, 13;
    6, 16, 11, 10,  7
    ]
A =
    14     8     0     4     4
     4    10    18     9     9
     2     1    14    19    12
     5     5     9    10    13
     6    16    11    10     7
```

在进行矩阵分解之前，我们需要验证该矩阵 A 是否可逆

```
>> det(A)==0
ans =
  logical
   0
```

输出为逻辑 0，则表示矩阵 A 的行列式不为 0，矩阵是可逆的，满足 LU 分解的基本条件，可以进行分解

调用 Matlab 自带的函数 `lu()` 进行矩阵分解，`[L,U] = lu(A)` 函数可以将满秩矩阵 A 分解为一个上三角矩阵 U 和一个经过置换的下三角矩阵 L，使得

$$A = LU \tag{1.4}$$

```
>> [L,U] = lu(A)
L =

    1.0000         0         0         0         0
    0.2857    0.6136    0.7965    1.0000         0
    0.1429   -0.0114    1.0000         0         0
```

```
    0.3571    0.1705    0.5044    0.1823    1.0000
    0.4286    1.0000         0         0         0
U =
   14.0000    8.0000         0    4.0000    4.0000
         0   12.5714   11.0000    8.2857    5.2857
         0         0   14.1250   18.5227   11.4886
         0         0         0  -11.9799   -4.5366
         0         0         0         0    5.7024
```

为了验证结果，我们将上三角矩阵 U 和下三角矩阵 L 相乘

```
>> RES=L*U
RES =
   14.0000    8.0000         0    4.0000    4.0000
    4.0000   10.0000   18.0000    9.0000    9.0000
    2.0000    1.0000   14.0000   19.0000   12.0000
    5.0000    5.0000    9.0000   10.0000   13.0000
    6.0000   16.0000   11.0000   10.0000    7.0000
```

得到的结果矩阵 B 与待分解矩阵一致

此外，该函数 `[L,U,P] = lu(A)` 还可以返回一个置换矩阵 P，并满足 $A = P^T LU$。在此语法中，L 是单位下三角矩阵，U 是上三角矩阵。

```
>> [L,U,P] = lu(A)
L =
    1.0000         0         0         0         0
    0.4286    1.0000         0         0         0
    0.1429   -0.0114    1.0000         0         0
    0.2857    0.6136    0.7965    1.0000         0
    0.3571    0.1705    0.5044    0.1823    1.0000

U =
   14.0000    8.0000         0    4.0000    4.0000
         0   12.5714   11.0000    8.2857    5.2857
         0         0   14.1250   18.5227   11.4886
         0         0         0  -11.9799   -4.5366
         0         0         0         0    5.7024

P =
     1     0     0     0     0
     0     0     0     0     1
     0     0     1     0     0
     0     1     0     0     0
     0     0     0     1     0
```

验证分解的结果

```
>> RES=P'*L*U
RES =
   14.0000    8.0000         0    4.0000    4.0000
    4.0000   10.0000   18.0000    9.0000    9.0000
    2.0000    1.0000   14.0000   19.0000   12.0000
    5.0000    5.0000    9.0000   10.0000   13.0000
    6.0000   16.0000   11.0000   10.0000    7.0000
```

1.3 利用 LU 分解求解线性方程组的 Matlab 实现

假定需要求解的方程组为 $Ax = b$，即

$$\begin{bmatrix} 14 & 8 & 0 & 4 & 4 \\ 4 & 10 & 18 & 9 & 9 \\ 2 & 1 & 14 & 19 & 12 \\ 5 & 5 & 9 & 10 & 13 \\ 6 & 16 & 11 & 10 & 7 \end{bmatrix} x = \begin{bmatrix} 12 \\ 16 \\ 9 \\ 8 \\ 16 \end{bmatrix} \tag{1.5}$$

求解上述方程组的过程如下

```
>> b=[12;16;9;8;16];
>> [L,U,P] = lu(A);
>> y = L\(P*b)
y =
   12.0000
   10.8571
    7.4091
    0.0080
   -1.8752

>> x = U\y
x =
    0.7046
    0.3694
    0.6296
    0.1239
   -0.3288
```

验证结果

```
>> RES=A*x
RES =
   12.0000
```

```
   16.0000
    9.0000
    8.0000
   16.0000
```

1.4 LU 分解的 C++ 实现及结果验证

```cpp
#include <iostream>
#include <Eigen/Dense>

using namespace std;
using namespace Eigen;
int main()
{
    Matrix<double, 5, 5> A;
    A
       <<14,8,0,4,4,4,10,18,9,9,2,1,14,19,12,5,5,9,10,13,6,16,11,10,7;

    cout << "matrix A:" << endl << A << endl << endl;

    FullPivLU<Matrix<double, 5, 5>> lu(A);

    Matrix<double, 5, 5> L = Matrix<double, 5, 5>::Identity();
    L.block<5,5>(0,0).triangularView<StrictlyLower>()=lu.matrixLU
       ();
    cout << "matrix L:" << endl << L << endl << endl;

    Matrix<double, 5, 5> U = lu.matrixLU().triangularView<Upper
       >();
    cout << "matrix U:" << endl << U << endl << endl;

    cout << "reconstruct the original matrix A:" << endl;
    auto reconstruct = lu.permutationP().inverse() * L * U
                     * lu.permutationQ().inverse();
    cout << reconstruct << endl;
    cout << () << endl;

    return 0;
}
```

编译代码

```
mkdir build
cd build
```

```
cmake -DCMAKE_BUILD_TYPE=Release ..
make -j8
cd ../bin
```

运行二进制文件，得到输出为

```
E:\Projects\Matrix-Theory-Assignment\cpp\bin>lu.exe
matrix A:
14  8  0  4  4
4  10 18  9  9
2   1 14 19 12
5   5  9 10 13
6  16 11 10  7

matrix L:
           1           0           0           0           0
    0.526316           1           0           0           0
    0.210526    0.503401           1           0           0
    0.473684    0.615646 0.000613497           1           0
    0.526316    0.289116    0.226994    0.182325           1

matrix U:
       19         1         2        14        12
        0   15.4737   4.94737   3.63158  0.684211
        0         0   11.0884  -4.77551   1.12925
        0         0         0   9.13558   2.89387
        0         0         0         0   5.70244

reconstruct the original matrix A:
14  8  0  4  4
4  10 18  9  9
2   1 14 19 12
5   5  9 10 13
6  16 11 10  7
```

2 QR 分解

2.1 QR 分解简介

QR 分解能够将 $m \times n (m \geq n)$ 的矩阵 A 分解成为一个 $m \times n$ 上三角矩阵 R 和一个 $m \times m$ 正交矩阵 Q

$$A = QR \tag{2.1}$$

另一种 QR 分解的形式是引入 $n \times n$ 置换矩阵 P 使得 AP 分解成为一个 $m \times n$ 上三角矩阵 R 和一个 $m \times m$ 正交矩阵 Q

$$AP = QR \tag{2.2}$$

2.2 QR 分解的 Matlab 实现及结果验证

用 Matlab 自带的函数 `rand()` 创建一个大小为 3×5 的随机矩阵

```
>> A=rand(3,5)*20
A =
    16.2945    18.2675     5.5700    19.2978    19.1433
    18.1158    12.6472    10.9376     3.1523     9.7075
     2.5397     1.9508    19.1501    19.4119    16.0056
```

向下取整后作为项目测试用的矩阵 A

$$A = \begin{bmatrix} 16 & 18 & 5 & 19 & 19 \\ 18 & 12 & 10 & 3 & 9 \\ 2 & 1 & 19 & 19 & 16 \end{bmatrix} \tag{2.3}$$

设置矩阵 A

```
>> A = [16, 18, 5, 19, 19;18, 12, 10, 3, 9;2, 1, 19, 19, 16];
```

调用 Matlab 自带的函数 `qr()` 进行矩阵分解，`[Q,R] = qr(A)` 函数可以对 3×5 矩阵 A 进行 QR 分解，满足式2.1

```
>> [Q, R] = qr(A)
Q =
   -0.6621    0.7481    0.0449
   -0.7448   -0.6502   -0.1497
   -0.0828   -0.1325    0.9877
R =
```

```
  -24.1661   -20.9384   -12.3313   -16.3866   -20.6074
         0     5.5301    -5.2799     9.7449     6.2410
         0          0    17.4946    19.1707    15.3096
```

为了验证结果，我们将 3×3 的正交矩阵 Q 和 3×5 的上三角矩阵 R 相乘得到结果矩阵 RES

```
>> RES = Q * R
RES =
        16.0000    18.0000     5.0000    19.0000    19.0000
        18.0000    12.0000    10.0000     3.0000     9.0000
         2.0000     1.0000    19.0000    19.0000    16.0000
```

得到的结果矩阵 RES 与待分解矩阵 A 一致

此外，该函数 `[Q,R,P] = qr(A)` 还会额外返回一个 5×5 的置换矩阵 P，并满足

$$AP = QR \tag{2.4}$$

```
>> [Q, R, P] = qr(A)
Q =
   -0.7027   -0.2969   -0.6465
   -0.1110   -0.8519    0.5118
   -0.7027    0.4314    0.5657
R =
  -27.0370   -14.6466   -17.9754   -14.6836   -25.5945
         0   -19.2218    -1.8064   -15.1357    -6.4056
         0          0    12.6342    -4.9298     1.3739
P =
     0     1     0     0     0
     0     0     0     1     0
     0     0     1     0     0
     1     0     0     0     0
     0     0     0     0     1
```

为了验证结果，我们将 3×5 的待分解矩阵 A 和 5×5 的置换矩阵 P 相乘得到结果矩阵 RES_1，并且将 3×3 的正交矩阵 Q 和 3×5 的上三角矩阵 R 相乘得到结果矩阵 RES_2

```
>> RES1 = A * P
RES2 = Q * R
RES1 =
    19    16     5    18    19
     3    18    10    12     9
    19     2    19     1    16
RES2 =
   19.0000   16.0000    5.0000   18.0000   19.0000
```

```
    3.0000    18.0000    10.0000    12.0000     9.0000
   19.0000     2.0000    19.0000     1.0000    16.0000
```

矩阵 $RES1$ 和矩阵 $RES2$ 的结果一致

2.3 QR 分解的 C++ 实现及结果验证

```cpp
#include <iostream>
#include <Eigen/Dense>

using namespace std;
using namespace Eigen;
int main()
{
    MatrixXf A(3, 5);
    A << 16, 18,  5, 19, 19,
         18, 12, 10,  3,  9,
          2,  1, 19, 19, 16;
    cout << "matrix A:" << endl << A << endl << endl;

    HouseholderQR<MatrixXf> qr(A);
    qr.compute(A);
    MatrixXf R = qr.matrixQR().triangularView<Upper>();
    MatrixXf Q = qr.householderQ();
    cout << "matrix Q:" << endl << Q << endl << endl;
    cout << "matrix R:" << endl << R << endl << endl;

    return 0;
}
```

编译运行后

```
E:\Projects\Matrix-Theory-Assignment\cpp\bin>qr.exe
matrix A:
16 18  5 19 19
18 12 10  3  9
 2  1 19 19 16

matrix Q:
 -0.662085   0.748083  0.0448963
 -0.744845  -0.650238  -0.149654
-0.0827606  -0.132525   0.987719

matrix R:
-24.1661 -20.9384 -12.3313 -16.3866 -20.6074
```

```
         0   5.53012  -5.27993   9.74489   6.24104
         0         0   17.4946   19.1707   15.3096
```

该结果与 Matlab 运行结果一致

3 SVD 分解

3.1 SVD 分解简介

SVD (Singular Value Decomposition) 分解 usv 能够将 $m \times n$ 矩阵 A 分解成为 $m \times m$ 的酉矩阵 U 、$m \times n$ 的矩阵 Σ 和 $n \times n$ 的酉矩阵 V ，满足

$$A = U\Sigma V^T \tag{3.1}$$

其中，酉矩阵 U 和 V 满足 $U^TU = I, V^TV = I$ ，Σ 是一个对角矩阵，主对角线上的值就是奇异值

3.2 SVD 分解的 Matlab 实现及结果验证

用 Matlab 自带的函数 `rand()` 创建一个大小为 4×5 的随机矩阵

```
>> A=rand(4,5)*20
A =
    2.8377   19.1898   18.6799    7.8445    0.6367
    8.4352   13.1148   13.5747   13.1096    5.5385
   18.3147    0.7142   15.1548    3.4237    0.9234
   15.8441   16.9826   14.8626   14.1209    1.9426
```

向下取整后作为项目测试用的矩阵 +A

$$A = \begin{bmatrix} 2 & 19 & 18 & 7 & 0 \\ 8 & 13 & 13 & 13 & 5 \\ 18 & 0 & 15 & 3 & 0 \\ 15 & 16 & 14 & 14 & 1 \end{bmatrix} \tag{3.2}$$

设置矩阵 A

```
>> A = [2, 19, 18, 7, 0; 8, 13, 13, 13, 5;
       18, 0, 15, 3, 0; 15, 16, 14, 14, 1]
```

调用 Matlab 自带的函数 `svd()` 进行矩阵分解，`[U, S, V] = svd(A)` 函数可以对 4×5 矩阵 A 进行 SVD 分解，满足式3.1

```
>> [U, S, V] = svd(A)
U =
  -0.5101   -0.5302   -0.6739    0.0672
  -0.4906   -0.1470    0.4118   -0.7538
  -0.3720    0.8332   -0.3872   -0.1319
```

```
    -0.6006     0.0542     0.4758     0.6402
S =
    48.4506          0          0          0          0
          0    17.8100          0          0          0
          0          0     9.1462          0          0
          0          0          0     4.2053          0
V =
    -0.4262     0.7622     0.2312     0.3172     0.2886
    -0.5300    -0.6242     0.0177     0.4095     0.4018
    -0.6099     0.1012    -0.6477    -0.3814    -0.2300
    -0.4019    -0.1327     0.6708    -0.1810    -0.5815
    -0.0630    -0.0382     0.2771    -0.7440     0.6035
```

分解之后的奇异值为 48.4506, 17.8100, 9.1462, 4.2053

首先验证 U、V 是否是酉矩阵

```
>> RES=U*U'
RES =
     1.0000     0.0000     0.0000     0.0000
     0.0000     1.0000     0.0000          0
     0.0000     0.0000     1.0000     0.0000
     0.0000          0     0.0000     1.0000
```

```
>> RES=V*V'
RES =
     1.0000    -0.0000     0.0000     0.0000     0.0000
    -0.0000     1.0000     0.0000    -0.0000     0.0000
     0.0000     0.0000     1.0000     0.0000     0.0000
     0.0000    -0.0000     0.0000     1.0000    -0.0000
     0.0000     0.0000     0.0000    -0.0000     1.0000
```

求 U^TU 和 V^TV 之后，得到的结果均为单位阵 I，符合酉矩阵的定义。验证分解的结果，求 $RES = USV^T$:

```
>> RES=U*S*V'
RES =
     2.0000    19.0000    18.0000     7.0000     0.0000
     8.0000    13.0000    13.0000    13.0000     5.0000
    18.0000     0.0000    15.0000     3.0000     0.0000
    15.0000    16.0000    14.0000    14.0000     1.0000
```

3.3 SVD 分解的 Matlab 实现及应用

对矩阵进行 SVD 分解之后，可以根据奇异值分解结果来确定矩阵的秩、列空间和零空间。

我们可以根据 SVD 分解的结果求解矩阵的秩（我们使用在上一部分中分解的数据进行）

```
>> s = diag(S)
rank_A = nnz(s)
s =
   48.4506
   17.8100
    9.1462
    4.2053
rank_A =
     4
```

我们可以根据 SVD 分解的结果求解矩阵的列空间，使用 U 中有对应的非零奇异值的列来计算 A 的列空间的标准正交基。

```
>> column_basis = U(:,logical(s))
column_basis =
   -0.5101   -0.5302   -0.6739    0.0672
   -0.4906   -0.1470    0.4118   -0.7538
   -0.3720    0.8332   -0.3872   -0.1319
   -0.6006    0.0542    0.4758    0.6402
```

我们可以根据 SVD 分解的结果求解矩阵的零空间，使用 V 中有对应的零奇异值的列来计算 A 的零空间的标准正交基。

```
>> null_basis = V(:,~s)

null_basis =

  空的 5×0 double 矩阵
```

3.4 SVD 分解的 C++ 实现及结果验证

```
#include <iostream>
#include <Eigen/Dense>

using namespace std;
using namespace Eigen;
int main()
{
    MatrixXf A(4, 5);
    A << 2 , 19 , 18 ,  7 ,  0 ,
        8 , 13 , 13 , 13 ,  5 ,
        18 ,  0 , 15 ,  3 ,  0 ,
```

```
        15 , 16 , 14 , 14 ,  1;
    cout << "matrix A:" << endl << A << endl << endl;


    JacobiSVD<MatrixXf> svd(A, ComputeThinU | ComputeThinV);
    cout << "Its singular values are:" << endl << svd.
       singularValues() << endl<< endl;
    cout << "matrix U: " << endl << svd.matrixU() << endl<< endl;
    cout << "matrix V: " << endl << svd.matrixV() << endl<< endl;

    return 0;
}
```

编译运行后

```
E:\Projects\Matrix-Theory-Assignment\cpp\bin>svd.exe
matrix A:
2   19 18  7  0
8   13 13 13  5
18   0 15  3  0
15  16 14 14  1

Its singular values are:
48.4506
  17.81
9.14625
4.20531

matrix U:
-0.510071 -0.530243 -0.673905 0.0672442
-0.490559 -0.147009  0.411754 -0.753789
-0.372042   0.83324 -0.387175 -0.131875
-0.600636 0.0542391  0.475822  0.640226

matrix V:
 -0.426226     0.76223    0.231176    0.317171
 -0.529999    -0.62425   0.0176852     0.40948
  -0.60986    0.101204   -0.647655   -0.381384
  -0.40191    -0.13272    0.670816   -0.180966
-0.0630215  -0.0382261    0.277118   -0.743994
```

得到的奇异值为 48.4506, 17.81, 9.14625, 4.20531，与 Matlab 中结果一致

附录 A　Matlab 代码

A.1　LU 分解

lu_test.m

```
A = [
    14, 8, 0, 4, 4;
    4, 10, 18, 9, 9;
    2, 1, 14, 19, 12;
    5, 5, 9, 10, 13;
    6, 16, 11, 10, 7
    ]

det(A)==0

[L,U] = qr(A);
RES=L*U;


[L,U,P] = qr(A);
RES=P'*L*U;


b=[12;16;9;8;16]
[L,U,P] = qr(A);
y = L\(P*b);
x = U\y;

RES=A*x
```

A.2　QR 分解

qr_test.m

```
A = [
    16, 18, 5, 19, 19;
    18, 12, 10, 3, 9;
    2, 1, 19, 19, 16
    ]

[Q, R] = qr(A)
```

```
A
RES = Q * R

[Q, R, P] = qr(A)
RES1 = A * P
RES2 = Q * R
```

A.3 SVD 分解

svd_test.m

```
A = [2, 19, 18, 7, 0; 8, 13, 13, 13, 5;
    18, 0, 15, 3, 0; 15, 16, 14, 14, 1];

[U, S, V] = svd(A);
RES=U*U';
RES=V*V';

RES=U*S*V';

s = diag(S)
rank_A = nnz(s)

column_basis = U(:,logical(s))

null_basis = V(:,~s)
```

附录 B　C++ 代码

B.1　LU 分解

lu.cpp

```cpp
#include <iostream>
#include <Eigen/Dense>

using namespace std;
using namespace Eigen;
int main()
{
    Matrix<double, 5, 5> A;
    A << 14, 8, 0, 4, 4, 4, 10, 18, 9, 9, 2, 1, 14, 19, 12, 5, 5, 9, 10, 13, 6, 16, 11, 10, 7;
    cout << "matrix A:" << endl << A << endl << endl;

    FullPivLU<Matrix<double, 5, 5>> lu(A);

    Matrix<double, 5, 5> L = Matrix<double, 5, 5>::Identity();
    L.block<5, 5>(0, 0).triangularView<StrictlyLower>() = lu.matrixLU();
    cout << "matrix L:" << endl << L << endl << endl;

    Matrix<double, 5, 5> U = lu.matrixLU().triangularView<Upper>();
    cout << "matrix U:" << endl << U << endl << endl;

    cout << "reconstruct the original matrix A:" << endl;
    auto reconstruct = lu.permutationP().inverse() * L * U * lu.permutationQ().inverse();
    cout << reconstruct << endl;


    return 0;
}
```

B.2 QR 分解

qr.cpp

```
#include <iostream>
#include <Eigen/Dense>

using namespace std;
using namespace Eigen;
int main()
{
    MatrixXf A(3, 5);
    A << 16, 18, 5, 19, 19, 18, 12, 10, 3, 9, 2, 1, 19, 19, 16;
    cout << "matrix A:" << endl << A << endl << endl;


    HouseholderQR<MatrixXf> qr(A);
    qr.compute(A);
    MatrixXf R = qr.matrixQR().triangularView<Upper>();
    MatrixXf Q = qr.householderQ();
    cout << "matrix Q:" << endl << Q << endl << endl;
    cout << "matrix R:" << endl << R << endl << endl;

    return 0;
}
```

B.3 SVD 分解

svd.cpp

```
#include <iostream>
#include <Eigen/Dense>

using namespace std;
using namespace Eigen;
int main()
{
    MatrixXf A(4, 5);
    A << 2, 19, 18, 7, 0, 8, 13, 13, 13, 5, 18, 0, 15, 3, 0, 15,
        16, 14, 14, 1;
    cout << "matrix A:" << endl << A << endl << endl;


    JacobiSVD<MatrixXf> svd(A, ComputeThinU | ComputeThinV);
```

```
    cout << "Its singular values are:" << endl << svd.
       singularValues() << endl<< endl;
    cout << "matrix U: " << endl << svd.matrixU() << endl<< endl;
    cout << "matrix V: " << endl << svd.matrixV() << endl<< endl;

    return 0;
}
```