实验八 矩阵特征值及特征向量计算

一、实验目的

1. 掌握求矩阵的主特征值（即按模最大的特征值）和主特征向量的幂法；
2. 初步了解幂法的加速。

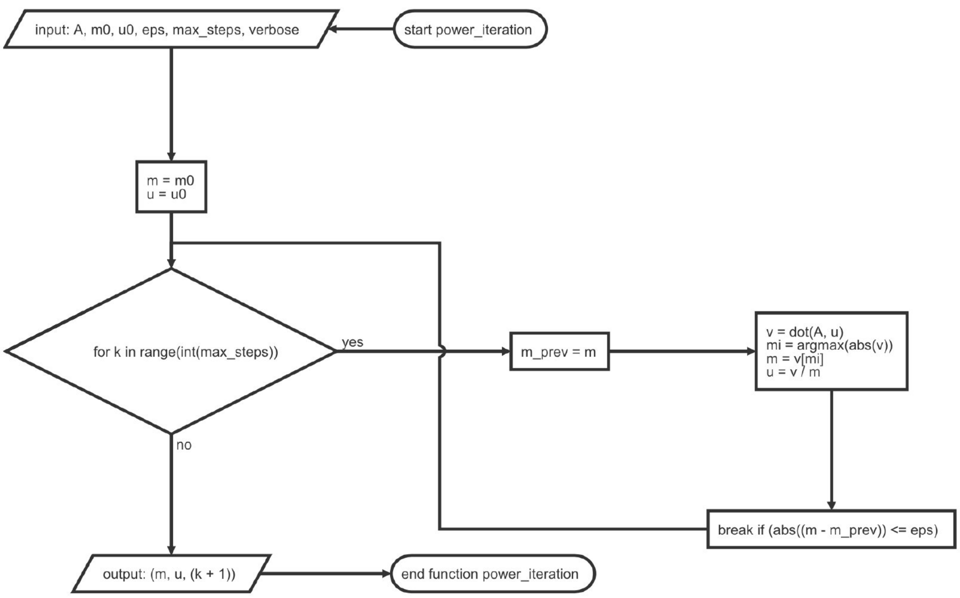
二、实验过程和结果

1. 正幂法的实现

```python
def power_iteration(A, m0=1, u0=None, eps=1e-8, max_steps=500, verbose=False):
    """正幂法 (power iteration a.k.a. the power method)
    计算矩阵 A 的按模最大特征值、特征向量。

    Args:
        A:  np_array_like 待求特征值的矩阵 (nxn)
        m0: float 初始特征值 default m0=1
        u0: np_array_like 初始特征向量 (n)  default u0=None: 取 u0 = (1, 1, ..., 1)
        eps: float 精度要求 default eps=1e-8
        max_steps: int 最大迭代次数 default max_steps=1000
        verbose: bool, 若为 True 则打印出每一步的结果 default verbose=False

    Returns:
        (m, u, k): 在 max_steps 次迭代以内得到第一组的满足 eps 的结果
            m: float 所求主特征值
            u: np.array 相应的特征向量
            k: int 迭代次数

    Raises:
        ValueError: 给定参数 A u0 尺寸不匹配
        Exception:  无法在max_steps 次迭代以内得到满足精度 eps 的结果
    """
    ...
```

（这里只保留函数的声明和文档注释，具体实现源码见：ex8/src/power_method.py）

程序流程图：



算一个课本上的例子（P159 例1）作为测试：

```
A = [[2, 3, 2], [10, 3, 4], [3, 6, 1]]
power_iteration(A, eps=1e-8, verbose=True)
```

```
0 1 [1. 1. 1.]
1 17.0 [0.41176471 1.         0.58823529]
2 9.470588235294118 [0.52795031 1.         0.82608696]
3 11.58385093167702 [0.49276139 1.         0.72600536]
4 10.831635388739945 [0.50200485 1.         0.75743775]
5 11.049799514875502 [0.49945569 1.         0.74783731]
6 10.985906091381928 [0.50014864 1.         0.75061668]
7 11.003953118800313 [0.49995948 1.         0.74982713]
8 10.998903290037243 [0.50001105 1.         0.75004801]
9 11.000302582696586 [0.49999699 1.         0.74998675]
10 10.999916852432536 [0.50000082 1.         0.75000364]
11 11.000022790833176 [0.49999978 1.         0.749999  ]
12 10.999993763594336 [0.50000006 1.         0.75000027]
13 11.000001704609195 [0.49999998 1.         0.74999993]
14 10.999999534421143 [0.5        1.         0.75000002]
15 11.000000127100696 [0.5        1.         0.74999999]
16 10.999999965313513 [0.5  1.   0.75]
17 11.00000000946407 [0.5  1.   0.75]
18 10.999999997418142 [0.5  1.   0.75]
result of power_iteration: 11.000000000704278 [0.5  1.   0.75] 19
```

计算一个比较大的随机矩阵，和 NumPy 求得得结果做比较：

```
A = np.random.random((100, 100))
# print(A)

r = power_iteration(A, eps=1e-5)
print('by power_iteration:', r[0])

a = np.linalg.eig(A)[0]
ar = [x for x in a if not np.iscomplex(x)]
i = np.argmax(np.abs(ar))
print('by   np.linalg.eig:', ar[i])
```

```
by power_iteration: 50.445839029418046
by   np.linalg.eig: (50.44583922496655+0j)
```

2. 反幂法的实现

反幂法的代码实现在框架上和正幂法相同，只有迭代中值更新的算法，以及最后输出时的取值有所不同：

```
def inverse_iteration(A, m0=1, u0=None, eps=1e-8, max_steps=500, verbose=False):
    """反幂法 (inverse iteration a.k.a. inverse power method)
    计算矩阵 A 的按模最小特征值、特征向量。
    """
    ...
    for k in range(int(max_steps)):
        ...
        v = np.linalg.solve(A, u)
        mi = np.argmax(np.abs(v))
        m = v[mi]
        u = v / m
        ...
    ...
    return 1/m, u, k+1
```

（`...`代表和正幂法相同的代码，完整实现见：ex8/src/inverse_power.py）

调用结果:

```python
A = [[2, 3, 2], [10, 3, 4], [3, 6, 1]]
inverse_iteration(A, eps=1e-5, max_steps=40, verbose=True)
```

```
0 1.0 [1. 1. 1.]
1 2.75 [-0.25        0.41666667  1.        ]
2 -1.7368421052631582 [-0.39473684 -0.25877193  1.        ]
3 -1.522306525037936 [-0.27587253 -0.28244815  1.        ]
4 -1.773137541073161 [-0.24854257 -0.33791831  1.        ]
5 -1.850713802809302 [-0.22932389 -0.36045702  1.        ]
6 -1.9067348353996336 [-0.21874548 -0.37508306  1.        ]
7 -1.9394212652843035 [-0.21209945 -0.38385382  1.        ]
8 -1.9604633754301686 [-0.20791023 -0.38945545  1.        ]
9 -1.973976092920489 [-0.20520426 -0.39306055  1.        ]
10 -1.9828015632828082 [-0.20343978 -0.3954137   1.        ]
11 -1.988599435522524 [-0.2022801  -0.39695986  1.        ]
12 -1.9924284508421255 [-0.20151431 -0.39798092  1.        ]
13 -1.9949649983827589 [-0.201007   -0.39865733  1.        ]
14 -1.996648958194817 [-0.20067021 -0.39910639  1.        ]
15 -1.997768464464223 [-0.20044631 -0.39940492  1.        ]
16 -1.998513415490028 [-0.20029732 -0.39960358  1.        ]
17 -1.999009434502497 [-0.20019811 -0.39973585  1.        ]
18 -1.9993398409805123 [-0.20013203 -0.39982396  1.        ]
19 -1.9995599908120016 [-0.200088   -0.39988266  1.        ]
20 -1.9997067035591078 [-0.20005866 -0.39992179  1.        ]
21 -1.9998044881537032 [-0.2000391  -0.39994786  1.        ]
22 -1.999869667263002 [-0.20002607 -0.39996524  1.        ]
23 -1.9999131152833085 [-0.20001738 -0.39997683  1.        ]
result of inverse_iteration: -1.999942078533036 [-0.20001158 -0.39998455  1.        ] 24
```

在反幂法中解了一个线性方程组，而且这个方程组是系数矩阵始终不变的，所以可以考虑调用实验6实现的 LU 分解法来求解:

```python
def inverse_iteration(A, m0=1, u0=None, eps=1e-8, max_steps=500, verbose=False):
    """反幂法 (inverse iteration a.k.a. inverse power method)
    计算矩阵 A 的按模最小特征值、特征向量。
    """
    ...
    lupA = lu(A)  # lupA = (l, u, p)
    for k in range(int(max_steps)):
        ...
        v = solve_lu(u, *lupA)
        ...
    ...
```

（同样略去了相同代码，完整实现见：ex8/src/inverse_power.py ，其中 LU 分解的程序来自实验6： [ex6/src/lu.py](ex6/src/lu.py)）

调用测试:

```python
A = [[2, 3, 2], [10, 3, 4], [3, 6, 1]]
inverse_iteration_lu(A, eps=1e-5, max_steps=40, verbose=True)
```

```
0 1.0 [1. 1. 1.]
1 2.7499999999999996 [-0.25        0.41666667  1.        ]
2 -1.7368421052631577 [-0.39473684 -0.25877193  1.        ]
3 -1.5223065250379362 [-0.27587253 -0.28244815  1.        ]
4 -1.773137541073161 [-0.24854257 -0.33791831  1.        ]
5 -1.8507138028093015 [-0.22932389 -0.36045702  1.        ]
6 -1.906734835399633 [-0.21874548 -0.37508306  1.        ]
7 -1.939421265284303 [-0.21209945 -0.38385382  1.        ]
8 -1.9604633754301677 [-0.20791023 -0.38945545  1.        ]
9 -1.9739760929204886 [-0.20520426 -0.39306055  1.        ]
10 -1.9828015632828078 [-0.20343978 -0.3954137   1.        ]
11 -1.9885994355225236 [-0.2022801  -0.39695986  1.        ]
12 -1.992428450842125 [-0.20151431 -0.39798092  1.        ]
13 -1.994964998382758 [-0.201007   -0.39865733  1.        ]
14 -1.9966489581948166 [-0.20067021 -0.39910639  1.        ]
15 -1.997768464464223 [-0.20044631 -0.39940492  1.        ]
16 -1.998513415490028 [-0.20029732 -0.39960358  1.        ]
17 -1.9990094345024974 [-0.20019811 -0.39973585  1.        ]
18 -1.9993398409805128 [-0.20013203 -0.39982396  1.        ]
19 -1.9995599908120016 [-0.200088   -0.39988266  1.        ]
20 -1.9997067035591083 [-0.20005866 -0.39992179  1.        ]
21 -1.9998044881537032 [-0.2000391  -0.39994786  1.        ]
22 -1.999869667263002 [-0.20002607 -0.39996524  1.        ]
23 -1.9999131152833085 [-0.20001738 -0.39997683  1.        ]
result of inverse_iteration_lu: -1.9999420785330355 [-0.20001158 -0.39998455  1.        ] 24
```

事实上，由于这里调用的 LU 分解算法相当原始，没有做任何计算上的优化，所以计算效率并不高。在不严谨的测试中可以看到，这里对于同一个问题，调用 LU 分解要远慢于使用 NumPy 来解方程：

```
call_numpy_linalg_eig executed in  0.0003 s
(-0.034991233156967005+0j)
inverse_iter_np_solve executed in  0.0005 s
-0.034991232623709595
inverse_iter_lu_solve executed in  0.0031 s
-0.03499123262370956
```

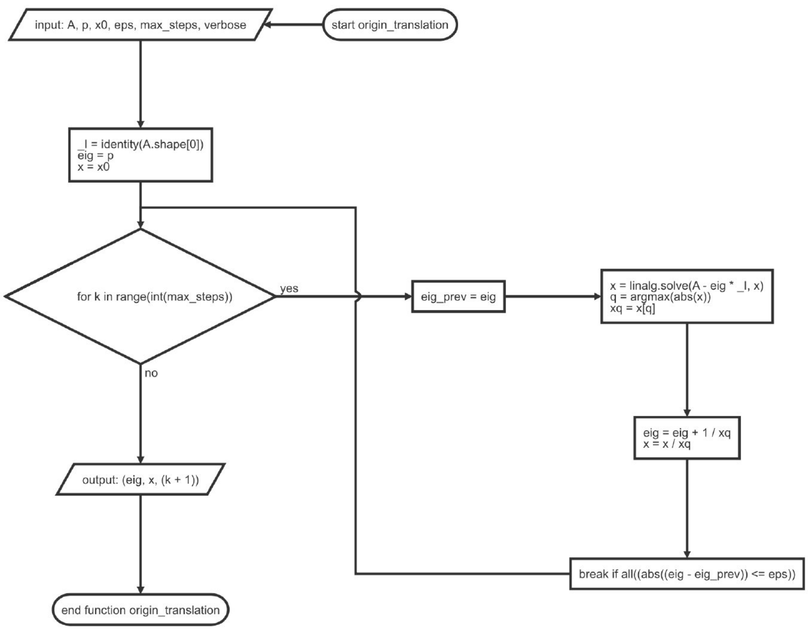
（这个测试的具体方法可以在这个 Jupyter Notebook 中找到： ex8/src/ex8.ipynb）

3. 原点位移法的实现

```python
def origin_translation(A, p, x0=None, eps=1e-8, max_steps=1000, verbose=False):
    """原点位移算法 (没搜到这个方法英文是什么，随便写的 origin_translation)

    Args:
        A:  np_array_like, 待求特征值的矩阵 (nxn)
        p:  float 位移因子
        x0: np_array_like, 初始向量 (n) , 要求 max(abs(x0)) == 1
                default x0=None: 取 x0 = (1, 1, ..., 1)
        eps: float, 控制精度 default eps=1e-8
        max_steps: 最大迭代次数 default max_steps=1000
        verbose: 打印出每步的结果，default verbose=False

    Returns:
        (eig, x, k)
            eig: float, A 的靠近 p 的特征值
            x: np.array, 单位化的特征向量
            k: int, 迭代次数

    Raises:
        ValueError: 给定参数 A 和 x0 尺寸不匹配，或 x0 = 0
        Exception:  无法在max_steps 次迭代以内得到满足精度 eps 的结果
    """
    ...
```

（完整实现见：ex8/src/origin_translation.py）

算法流程图：



调用测试：

```
A = [[2, 3, 2], [10, 3, 4], [3, 6, 1]]
origin_translation(A, 10, verbose=True)
```

```
0 10 [1. 1. 1.]
1 10.709090909090909 [0.47272727 1.         0.74545455]
2 11.005485349227762 [0.50058964 1.         0.74989723]
3 11.000001815769398 [0.50000023 1.         0.74999988]
4 11.000000000000188 [0.5   1.    0.75]
result of origin_translation: 11.0 [0.5   1.    0.75] 5
```

4. 加速法

用动态原点位移算法加速求解按模最大特征值：

```python
def accelerating_max_val_eig(A, x0=None, eps_pi=1e-1, eps_ot=1e-8, max_steps_pi=1000, max_steps_ot=1000, verbose=False):
    """动态原点位移算法求按模最大特征值

    Args:
        A:  np_array_like, 待求特征值的矩阵 (nxn)
        x0: np_array_like, 初始向量 (n) , default x0=None: 取 x0 = (1, 1, ..., 1)
        eps_pi: float,「正幂法」迭代的控制精度 default eps_pi=1e-1
        eps_ot: float,「原点位移」算法的控制精度, 须满足 eps > eps_ot, default eps_ot=1e-8
        max_steps_pi: int,「正幂法」迭代的最大迭代次数 default max_steps_pi=1000
        max_steps_ot: int,「原点位移」算法的最大迭代次数 default max_steps_ot=1000
        verbose: bool: True 则打印出「正幂法」、「原点位移」的完整迭代过程 default verbose=False

    Returns:
        (eig, eigv, k)
            eig: float, A 的按模最大特征值
            eigv: np.array, 单位化的特征向量
            k: 迭代次数

    Raises:
        ValueError: 参数不合法
        Exception:  无法在max_steps 次迭代以内得到满足精度 eps 的结果
    """
    eig, eigv, k_pi = power_iteration(A, u0=x0,
                                      eps=eps_pi, max_steps=max_steps_pi,
                                      verbose=verbose)
    eig, eigv, k_ot = origin_translation(A, eig, x0=eigv,
                                         eps=eps_ot, max_steps=max_steps_ot,
```

```
                                        verbose=verbose)
    return eig, eigv, k_pi + k_ot
```

（完整实现见：ex8/src/accelerate.py）

调用测试：

```
A = [[2, 3, 2], [10, 3, 4], [3, 6, 1]]
accelerating_max_val_eig(A, verbose=True)
```

```
    power_iteration:
0 1 [1. 1. 1.]
1 17.0 [0.41176471 1.         0.58823529]
2 9.470588235294118 [0.52795031 1.         0.82608696]
3 11.58385093167702 [0.49276139 1.         0.72600536]
4 10.831635388739945 [0.50200485 1.         0.75743775]
5 11.049799514875502 [0.49945569 1.         0.74783731]
result of power_iteration: 10.985906091381928 [0.50014864 1.         0.75061668] 6
    origin_translation:
0 10.985906091381928 [0.50014864 1.         0.75061668]
1 10.999996033673042 [0.49999985 1.         0.74999938]
2 11.000000000001119 [0.5  1.   0.75]
result of origin_translation: 11.0 [0.5  1.   0.75] 3
```

```
(11.0, array([0.5 , 1.  , 0.75]), 9)
```

用动态原点位移算法加速求解按模最小特征值（完整实现见：ex8/src/accelerate.py）只需把上述代码中的 `power_iteration` 改为 `inverse_iteration`。调用测试：

```
A = [[2, 3, 2], [10, 3, 4], [3, 6, 1]]
accelerating_min_val_eig(A, verbose=True)
```

```
    inverse_iteration:
0 1.0 [1. 1. 1.]
1 2.75 [-0.25        0.41666667  1.        ]
2 -1.7368421052631582 [-0.39473684 -0.25877193  1.        ]
result of inverse_iteration: -1.522306525037936 [-0.27587253 -0.28244815  1.        ] 3
    origin_translation:
0 -1.522306525037936 [-0.27587253 -0.28244815  1.        ]
1 -1.8991931095340484 [-0.22075327 -0.37282221  1.        ]
2 -1.9913582399414025 [-0.20172412 -0.39769765  1.        ]
3 -1.9999267327390964 [-0.20001466 -0.39998046  1.        ]
4 -1.9999999946318703 [-0.2 -0.4  1. ]
result of origin_translation: -2.0 [-0.2 -0.4  1. ] 5
```

```
(-2.0, array([-0.2, -0.4,  1. ]), 8)
```

5. 实验内容题目

1. 已知矩阵

$$A=\begin{pmatrix}4 & -1 & 1\\ -1 & 3 & -2\\ 1 & -2 & 3\end{pmatrix}$$

用幂法计算该矩阵主特征值和相应的特征向量。(容许误差10^{-6})

(其特征值为：6，3，1，对应特征向量为：$(1,-1,1)^T,(2,1,-1)^T,(0,1,1)^T$)

(1) 编写规范化幂法的程序，求上述矩阵的主特征值和相应特征向量，

分别选择初值：$(1,1,1)^T,(2.001,1.999,0)^T,(0,1,1)^T$，观察所需的迭代次数和迭代结果，试说明幂法对初值的选择有什么要求。

(2) 基于原点平移加速的思想，通过对矩阵 $B=A-\alpha E$（α 的值自己给定!）运用幂法求矩阵 A 的主特征值和主特征向量，在相同精度要求下，和（1）的结果进行比较，分析其加速效果。

2. 已知矩阵

$$A=\begin{pmatrix}4 & -1 & 1\\ -1 & 3 & -2\\ 1 & -2 & 3\end{pmatrix}$$

用反幂法计算该矩阵按模最小的特征值和相应的特征向量。(容许误差10^{-6})

(1) 将幂法的计算程序修改为反幂法的计算程序，对问题进行求解；

(2) 已知矩阵 A 某个特征值的近似值为 2.5，试计算该特征值!

解:

```
A = [[4, -1, 1],
     [-1, 3, -2],
     [1, -2, 3]]

for u0 in [[1, 1, 1], [2.001, 1.999, 0], [0, 1, 1]]:
    print(f"\n-- 取初值 u0 = {u0}")

    print("\n正幂法: ", end='')
    maxlp = power_iteration(A, u0=u0, eps=1e-6)
    print(maxlp)

    print("加速法: ", end='')
    maxla = accelerating_max_val_eig(A, x0=u0, eps_pi=1e-3, eps_ot=1e-6)
    print(maxla)

    print("\n反幂法: ", end='')
    minli = inverse_iteration(A, u0=u0, eps=1e-6)
    print(minli)

    print("加速法", end='')
    minla = accelerating_min_val_eig(A, x0=u0, eps_ii=1e-3, eps_ot=1e-6)
    print(minla)

print("\n-- 2.5 附近特征值: ")
r = origin_translation(A, 2.5)
print("原点位移: ", r)
```

得到结果:

```
-- 取初值 u0 = [1, 1, 1]

正幂法: (5.999999284744433, array([ 1.        , -0.99999982,  0.99999982]), 24)
加速法: (6.000000333333467, array([ 1.        , -0.99999947,  0.99999987]), 20)

反幂法: (1.0000004180226194, array([4.18175751e-07, 1.00000000e+00, 9.99999582e-01]), 13)
加速法(1.0000000000000009, array([-1.00994543e-15,  1.00000000e+00,  1.00000000e+00]), 9)

-- 取初值 u0 = [2.001, 1.999, 0]

正幂法: (5.999999476043831, array([ 1.        , -0.99999987,  0.99999987]), 35)
加速法: (5.999999666666533, array([ 1.        , -0.99999947,  0.99999987]), 30)

反幂法: (1.0000004184290925, array([4.18429144e-07, 1.00000000e+00, 9.99999582e-01]), 14)
加速法(1.0000000000000009, array([-9.81794114e-16,  1.00000000e+00,  1.00000000e+00]), 10)

-- 取初值 u0 = [0, 1, 1]

正幂法: (1, array([0., 1., 1.]), 1)
加速法: (0.9999999999999999, array([0., 1., 1.]), 4)

反幂法: (1.0, array([2.77555756e-17, 1.00000000e+00, 1.00000000e+00]), 1)
加速法(0.9999999999999999, array([0., 1., 1.]), 4)

-- 2.5 附近特征值:
原点位移:  (3.0, array([ 1. ,  0.5, -0.5]), 5)
```

三、思考题分析解答

> 幂法收敛速度取决于什么？为什么？

从幂法的推导中最后得到有 $\lambda_1 \approx \frac{x_i^{(k+1)}}{x_i^{(k)}}$，所以幂法的收敛速度取决于主特征值和第二大特征值的比值大小： $|\frac{\lambda_2}{\lambda_1}|$ ，这个值越小收敛越快。当这个比值接近 1 的时候，收敛的速度就比较慢了。

四、重点难点分析

1. 求矩阵的按模最大的主特征值和主特征向量的幂法；
2. 幂法的动态原点位移加速。