Humboldt Universität zu Berlin

Seminar Paper

Numerical Methods for solving Eigenvalue-Problems

Thomas Siskos (580726)

Numerical Introductory Course

Supervised by:

Prof. Dr. Brenda López Cabrera

June 29, 2018

Contents

List of Tables

List of Figures

List of Algorithms

1 Motivation

Abstract
Eigenvalues and eigenvectors are often the solution to multidimensional optimization problems, however computing them by hand for anything but trivial matrices is most of the time infeasible or inpractical. To this extend we would like to deploy an automated procedure which yields the correct eigenvectors and eigenvalues. We demonstrate the relevance of eigenvalues and eigenvectors by revising two applications from statistics, Principal Component Analysis and Fisher's Linear Discriminant Analysis, which we follow up by investigating four algorithms suited for eigenvalue problems. Finally we provide a compound solution that takes advantage of each algorithms strengths.

For many statistical applications eigenvectors provide a formidable solution. Be it dimensionality reduction in terms of a Principal Component Analysis or classification by Fisher's Linear Discriminant Analysis, both come in the guise of optimization problems. But what are eigenvalues and eigenvectors?

If A is an $n \times n$ matrix, v is a non-zero vector and λ is a scalar, such that

$$Av = \lambda v \tag{1}$$

then v is called an *eigenvector* and λ is called an *eigenvalue* of the matrix A. An eigenvalue of A is a root of the characteristic equation,

$$det\,(A - \lambda I) = 0. \tag{2}$$

Geometrically speaking, we require a vector which, when multiplied by matrix A, will not get rotated but only elongated by a factor λ.

When confronted with a high-dimensional data matrix $X \in \mathbb{R}^{n \times m}$ an analyst often wishes to find a lower-dimensional representation, while conserving as much of

the structure as possible. One way of achieving this goal is to choose a standardized linear combination of features that aim to maximize the variance of the projection $\delta' X$. We can formalize this as

$$max\ \delta' Var\left(X\right)\delta\ s.t.\ \sum \delta_i^2 = 1. \tag{3}$$

where $X \in \mathbb{R}^{n\times m}; m, n \in \mathbb{N}; \delta \in \mathbb{R}^m$. The Lagrangean that corresponds to the constrained maximization problem in 3 is

$$\mathcal{L}(Var\left(X\right), \delta, \lambda) = \delta' Var\left(X\right)\delta - \lambda\left(\delta'\delta - 1\right),$$

where $\lambda \in \mathbb{R}^m$

Taking derivatives we obtain the first order condition:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \delta} &\overset{!}{=} 0 \\ 2Var(X)\delta - 2\lambda_k\delta &\overset{!}{=} 0 \\ Var(X)\delta &= \lambda_k\delta \end{aligned}$$

Which is now reduced to a common Eigenvalue problem as posed in (1).

$$Y = \Gamma'\left(X - \mu\right) \tag{4}$$

where $Y \in \mathbb{R}^{n\times m}$ is the matrix of rotations, $\Gamma \in \mathbb{R}^{m\times m}$ is the matrix of eigenvectors, $\mu \in \mathbb{R}^m$ is the vector of sample means. [Härdle and Simar, 2015]

In section two we lay out the mathematical foundations for the operations we are about to perform. In particular, we will try to reformulate any complicated

eigenvalue problem into a straightforward one by diagonalizing the matrix in question, without altering the eigenvalues we would like to compute. We follow these justifications by proposing two main algorithms for computing eigenvalues, first the Jacobi-Method for symmetric matrices, then the QR-Method for arbitrary square matrices in section 3. Additionally, for the QR-Method we define two extensions which try to increase the initial QR-algorithm's speed. For all algorithms we provide implementations in the `Python`-programming-language [van Rossum, 1995, Hunter, 2007, McKinney, 2010]. In section 4 we will analyse the implemented routines by critically reflecting upon the accuracy of the obtained results as well as their efficiency. In the final section we provide a final algorithm which combines the strengths of the defined procedures by chosing the algorithm that is most fit for the underlying problem.

2 Similarity Transformations

In general we want to reformulate the eigenvalue problem of a complicated matrix into an eigenvalue problem of a simple matrix, which yields the same eigenvalues. Simple matrices in our case will be diagonal matrices, since with them it is possible to identify their eigenvalues simply as entries on the main diagonal. Such a transformation that conserves the eigenvalues of a matrix is called a *similarity transformation*.

Two $n \times n$ matrices A and B are called *similar* if there exists an invertible matrix P such that

$$A = P^{-1}BP. \tag{5}$$

It is obvious that the similarity relationship is commutative as well as transitive. If A and B are similar, it holds that

$$\begin{aligned} B - \lambda I &= P^{-1}BP - \lambda P^{-1}IP \\ &= A - \lambda I. \end{aligned}$$

Hence A and B have the same eigenvalues. This fact also follows immediately from the transitivity of the similarity relationship and the fact that a matrix is similar to the diagonal matrix formed from its eigenvalues, as stated in the spectral-decomposition. Important types of similarity transformations are based around orthogonal matrices. If Q is orthogonal and

$$A = Q'BQ,$$

A and B are called *orthogonally similar* [Gentle, 1998]. We will use *orthogonal similarity transformations* to diagonalize matrices we wish to know the eigenvalues of.

2.1 Householder-Reflections

Our goal is still to diagonalize a matrix in order to programmatically extract it's eigenvalues. So far we have seen that there exist such transformations that conserve the eigenvalues of a given matrix. However, we require transformations that, additionally, eliminate non-zero entries on the off-diagonal elements of said matrix. A greedy technique, that eliminates all but the first elements of a vector is proposed

in the form of Householder-Reflections.

Let u and v be orthonormal vectors and let x be a vector in the space spanned by u and v, such that

$$x = c_1 u + c_2 + v$$

for some scalars c_1 and c_2. The vector

$$\tilde{x} = -c_1 u + c_2 v$$

is a *reflection* of x through the line difined by the vector u. Now consider the matrix

$$P = I - 2uu'. \tag{6}$$

Note that

$$\begin{aligned} Px &= c_1 u + c_2 v - 2c_1 uuu' - 2c_2 vuu' \\ &= c_1 u + c_2 v - 2c_1 u'uu - 2c_2 u'vu \\ &= -c_1 u + c_2 v \\ &= \hat{x}. \end{aligned}$$

The matrix P is called a reflector. The usefulness of Householder-Reflections stems from the fact that it is easy to transform a vector of the form

$$x = (x_1, x_2, \ldots, x_n)$$

into a vector

$$\hat{x} = (\hat{x}_1, 0, \ldots, 0).$$

If $Qx = \hat{x}$, then $||x||_2 = ||\hat{x}||_2$ and thus $\hat{x_1} = \pm||x||_2$, since it is the only non-zero entry. To construct the reflector let

$$v = (x_1 + sign(x_1)||x||_2, x2, \ldots, x_n) \tag{7}$$

and $u = \frac{v}{||v||_2}$ [Gentle, 1998]. We use the *sign*-function, which simply returns the sign of its argument in order to avoid the numerical problem known as *catastrophic cancellation*. It can occur when adding two very close, but different, floating point numbers of differing signs. In some unfortunate cases both of these numbers get represented by the same computer number and, because of their opposing signs cancel each other out. In our case this would mean, that we reflect the vector onto the origin. Fortunately, by making use of the sign function we can make sure that both summands will share the same sign, thus mitigating any concerns about catastrophic cancellation.

We use reflectors to compute the so called QR factorization of an aribitrary square matrix $A \in \mathbb{R}^{n \times n}$.

$$A = QR \tag{8}$$

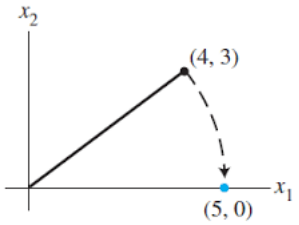
where Q is orthogonal and R is upper triangular. We use Householder transformations to reflect the i^{th} column and produce zeros below the (i, i) element. The QR-factorization of a matrix $A \in \mathbb{R}^5$ would therefore consist of five Householder-reflections with $Q = P_5 P_4 P_3 P_2 P_1$. The number of computations for the QR factor-

ization in this fashion is $2n^3/3$ multiplications and $2n^3/3$ additions [Gentle, 1998].

2.2 Givens-Rotations

Another way of forming the QR-factorization is by using orthogonal transformations which rotate a vector in a way such that a specific element becomes 0 and only one other element in the vector being changed. These transformations are called *Givens transformations, Givens Rotations* or *Jacobi transformations*

Figure 1: Rotation of x



A Givens rotation in $\mathbb{R}^2$.

Using orthogonal transformations we can also rotate a vector in such a way that a specified element becomes 0 and only one other element in the vector is changed. The basic idea can be seen in a two-dimensional space. We wish to rotate the vector $x = (x_1, x_2)$ to $\tilde{x} = (\tilde{x_1}, 0)$ as with a reflector.

It is easy to see that the orthogonal matrix

$$Q = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$

performs the desired rotation, if $\cos\theta = \frac{x_1}{||x||_2}$ and $\sin\theta = \frac{x_2}{||x||_2}$

In general, we can construct an orthogonal $matrix V_{pq}$, that will transform the vector

$$x = (x_1, \ldots, x_p, \ldots x_q, \ldots, x_n)$$

to

$$\tilde{x} = (x_1, \ldots, \tilde{x}_p, \ldots 0, \ldots, x_n)$$

. The matrix that does this is

$$V_{pq}(\theta) = \begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & \cos\theta & & \sin\theta & & \\ & & & \ddots & & & \\ & & -\sin\theta & & \cos\theta & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{bmatrix} \tag{9}$$

where $\cos\theta = \frac{x_p}{||x||}$ and $\sin\theta = \frac{x_q}{||x||}$.

A rotation matrix is therefore the same as an identitiy matrix, in which we change four elements [Börm and Mehl, 2012]. We will use Givens rotations primarily in the Jacobi-Method.

3 Algorithms

3.1 Jacobi Method

The *Jacobi-Method* for computing the eigenvalues of a symmetric matrix $A \in \mathbb{R}^{n\times n}$ deploys a sequence of orthogonal similarity transformations that eventually results in

$$A = P\Lambda P^{-1}$$

or

$$\Lambda = P^{-1}AP$$

,

where Λ is diagonal and P consists of a sequence of matrix multiplications $P = \prod_{k=1}^{K} V_{p_k,q_k}(\theta_k)$ and $V_{p_k,q_k}(\theta_k)$ is of the form proposed in (9). More specific the *Jacobi iteration* is

$$A^{(k)} = V'_{p_k,q_k}(\theta_k) A^{(k-1)} V_{p_k,q_k}(\theta_k), \tag{10}$$

where p_k, q_k and θ_k are chosen such that A^k resembles more a diagonal matrix than A^{k-1}. Specifically they will be chosen as to reduce the sum of squares of the off-diagonal elements. As we saw in (9) it is easy to chose an angle θ_k in order to introduce a zero in a single Givens rotation. Here we use the rotations in the context of a similarity transformation, so it is a little more complicated.

We require that $a_{pq}^{(k)} = 0$, this implies

$$a_{pq}^{(k-1)}(\cos^2\theta - \sin^2\theta) + \left(a_{pp}^{(k-1)} - a_{qq}^{(k-1)}\right)\cos\theta\sin\theta = 0. \tag{11}$$

We can use the trigonometric identities

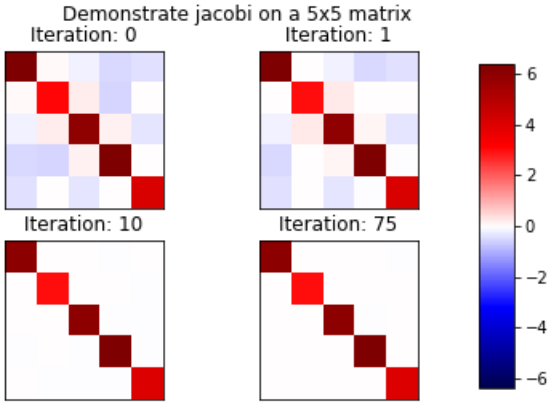
$$\begin{aligned}\cos(2\theta) &= \cos^2\theta \sin^2\theta \\ \sin(2\theta) &= 2\cos\theta\sin\theta \\ \tan(2\theta) &= \frac{\sin(2\theta)}{\cos(2\theta)},\end{aligned}$$

in (11) we have

$$\tan(2\theta) = \frac{2a_{pq}^{(k-1)}}{a_{pp}^{(k-1)} - a_{qq}^{(k-1)}}.$$

[Gentle, 1998]

Figure 2: Progress Jacobi-Method



From this we can retrieve the angle and obtain the rotation matrix in each iteration.

The algorithm converges if the off-diagonal elements are sufficiently small. The best index pair at a given iteration is the pair (p, q) that satisfies

$$|a_{pq}^{(k-1)}| = \max_{i<j} |a_{ij}^{(k-1)}|.$$

If this choice is made, the Jacobi Method can be shown to converge [Gentle, 1998].

Figure 1 visualizes the progress of the *Jacobi*-method on a symmetric 5×5 matrix. As we can see the matrix converges to a diagonal matrix already after 10 iterations.

Algorithm 1 `jacobi`

Require: symmetric matrix A
Ensure: $0 < precision < 1$
initialize: $L \leftarrow A$; $U \leftarrow I$; $L_{max} \leftarrow 1$
while $L_{max} > precision$ **do**
  Find indices i, j of largest value in lower triangle of $abs(L)$
  $L_{max} \leftarrow L_{i,j}$
  $\alpha \leftarrow \frac{1}{2} \cdot \arctan(\frac{2A_{i,j}}{A_{i,i}-A_{j,j}})$
  $V \leftarrow I$
  $V_{i,i}, V_{j,j} \leftarrow \cos\alpha$; $V_{i,j}, V_{j,i} \leftarrow -\sin\alpha, \sin\alpha$
  $A \leftarrow V'AV$; $U \leftarrow UV$
end while
return $diag(A)$, U

Algorithm 2 `QRM1`

Require: square matrix A
initialize: $conv \leftarrow False$
while not $conv$ **do**
  $Q, R \leftarrow$ QR-Factorization of A
  $A \leftarrow RQ$
  if A is diagonal **then**
    $conv \leftarrow$ `True`
  end if
end while
return $diag\,(A)\,,\ Q$

Algorithm 3 `QRM2`

Require: square matrix A
$A \leftarrow$ `hessenberg(`A`)`
continue with: QRM1(A)

Figure 3: Progress basic QR-Method

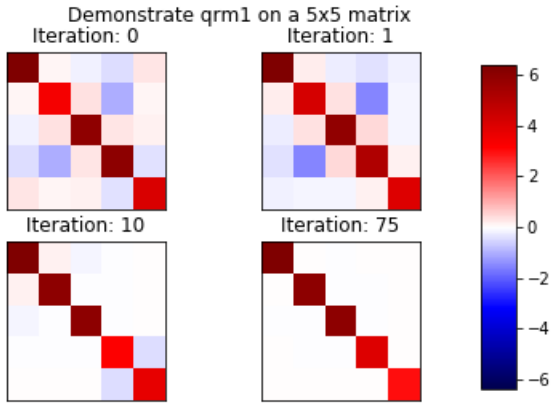


Figure 4: Progress Hessenberg-QR-Method

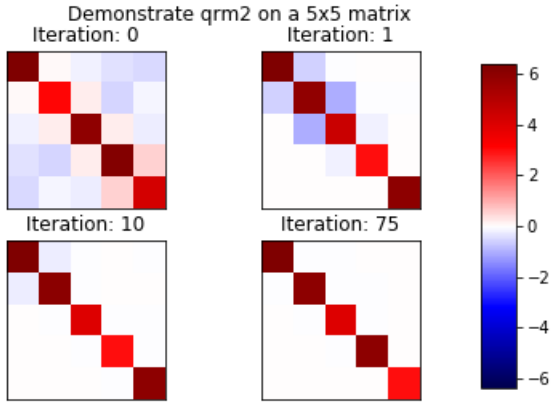
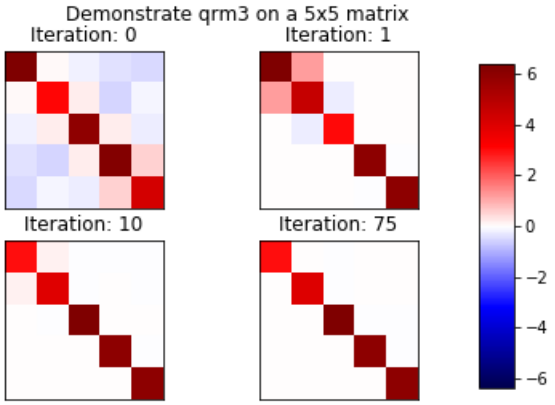


Figure 5: Progress Accelerated QR-Method



3.2 QR-Method

3.2.1 Hessenberg Variant

3.2.2 Accelerated Variant

Algorithm 4 `QRM3`

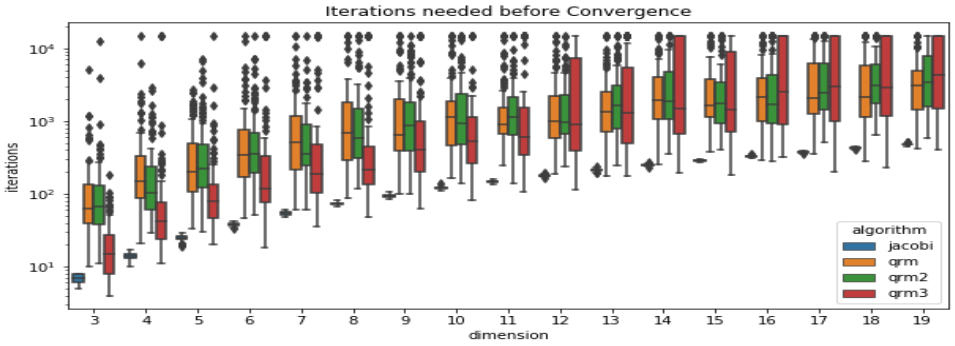
Require: square matrix $A \in \mathbb{R}^{p \times p}$

$T \leftarrow \texttt{hessenberg}(A),\ conv \leftarrow False$
while not $conv$ **do**
  $Q, R \leftarrow$ QR-Factorization of $T - t_{p-1,p-1}I$
  $T \leftarrow RQ + t_{p-1,p-1}I$
  if T is diagonal **then**
    $conv \leftarrow True$
  end if
end while
return $diag\,(T)\,,\ Q$

Table 1: Unit tests accross matrix-sizes

awesome	sauce
nothing	to
see	here

Figure 6: Unit-tests: Iterations



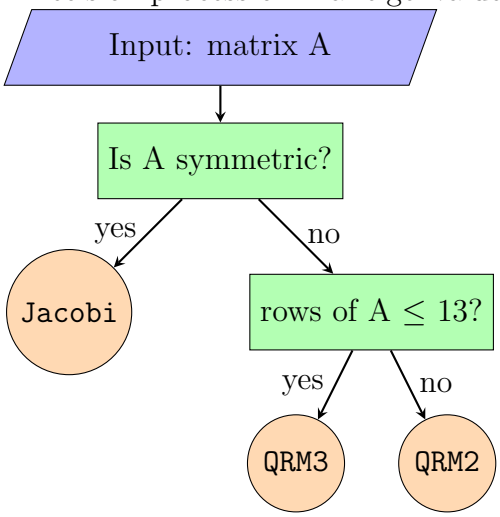
4 Analysis

4.1 Accuracy

4.2 Efficiency

5 Conclusion

Figure 7: Decision process of final eigenvalue routine



6 Appendix

6.1 Eigenvalue Routines

```
import numpy as np
import copy



def hreflect1D(x):
    """
    Calculate Householder reflection: Q = I - 2*uu'.

    Parameters:
        X: numpy array.

    Returns:
        Qx: reflected vector.
        Q: Reflector (matrix).
```

```
15      """
16      # Construct v:
17      v = copy.deepcopy(x)
18      v[0] += np.linalg.norm(x)
19
20      # Construct u: normalize v.
21      vnorm = np.linalg.norm(v)
22      if vnorm:
23          u = v / np.linalg.norm(v)
24      else:
25          u = v
26
27      # Construct Q:
28      Q = np.eye(len(x)) - 2 * np.outer(u, u)
29      Qx = np.dot(Q, x)
30
31      return Qx, Q
32
33
34  def qr_factorize(X, offset=0):
35      """
36      Compute QR factorization of X s.t. QR = X.
37
38      Parameters:
39          - X: square numpy ndarray.
40          - offset: (int) either 0 or 1. If offset is unity: compute
               Hessenberg-
41                      matrix.
42
43      Returns:
```

```
        Q: square numpy ndarray, same shape as X. Rotation matrix.
        R: square numpy ndarray, same shape as X. Upper triangular
           matrix if
            offset is 0, Hessenberg-matrix if offset is 1.
    """
    assert offset in [0, 1]
    assert type(X) == np.ndarray
    assert X.shape[0] == X.shape[1]

    R = copy.deepcopy(X)
    Q = np.eye(X.shape[0])

    for i in range(X.shape[0]-offset):
        Pi = np.eye(R.shape[0])
        _, Qi = hreflect1D(R[i+offset:, i])
        Pi[i+offset:, i+offset:] = Qi

        Q = Pi.dot(Q)
        R = Pi.dot(R)

    return Q.T, R
```

```
"""
Algorithms for solving eigenvalue problems.

1. Compute diagonalization of 2x2 matrices via jacobi iteration.
2. Generalize Jacobi iteration for symmetric matrices.
"""
import numpy as np
import copy
```

```
9  import warnings
10 from scipy import linalg as lin
11 from algorithms import helpers
12
13
14 def jacobi2x2(A):
15     """
16     Diagonalize a 2x2 matrix through jacobi step.
17
18     Solve: U' A U = E s.t. E is a diagonal matrix.
19
20     Parameters:
21         A - 2x2 numpy array.
22     Returns:
23         A - 2x2 diagonal numpy array
24     """
25     assert type(A) == np.ndarray
26     assert A.shape == (2, 2)
27     assert A[1, 0] == A[0, 1]
28
29     alpha = 0.5 * np.arctan(2*A[0, 1]/(A[1, 1] - A[0, 0]))
30     U = np.array([[np.cos(alpha), np.sin(alpha)],
31                   [-np.sin(alpha), np.cos(alpha)]])
32     E = np.matmul(U.T, np.matmul(A, U))
33     return E
34
35
36 def jacobi(X, precision=1e-6, debug=False):
37     """
38     Compute Eigenvalues and Eigenvectors for symmetric matrices.
```

```
39
40      Parameters:
41          X - 2D numpy ndarray which represents a symmetric matrix
42          precision - float in (0, 1). Convergence criterion.
43
44      Returns:
45          A - 1D numpy array with eigenvalues sorted by absolute
              value
46          U - 2D numpy array with associated eigenvectors (column).
47      """
48      assert 0 < precision < 1.
49      assert type(X) == np.ndarray
50      n, m = X.shape
51      assert n == m
52      assert all(np.isclose(X - X.T, np.zeros(n)).flatten())
53      A = copy.deepcopy(X)
54      U = np.eye(A.shape[0])
55      L = np.array([1])
56      iterations = 0
57
58      while L.max() > precision:
59          L = np.abs(np.tril(A, k=0) - np.diag(A.diagonal()))
60          i, j = np.unravel_index(L.argmax(), L.shape)
61          alpha = 0.5 * np.arctan(2*A[i, j] / (A[i, i]-A[j, j]))
62
63          V = np.eye(A.shape[0])
64          V[i, i], V[j, j] = np.cos(alpha), np.cos(alpha)
65          V[i, j], V[j, i] = -np.sin(alpha), np.sin(alpha)
66
67          A = np.dot(V.T, A.dot(V))
```

```
68          U = U.dot(V)
69          iterations += 1
70
71      # Sort by eigenvalue (descending order) and flatten A
72      A = np.diag(A)
73      order = np.abs(A).argsort()[::-1]
74      if debug:
75          return iterations
76
77      return A[order], U[:, order]
78
79
80  def qrm(X, maxiter=15000, debug=False):
81      """
82      Compute Eigenvalues and Eigenvectors using the QR-Method.
83
84      Parameters:
85          - X: square numpy ndarray.
86      Returns:
87          - Eigenvalues of A.
88          - Eigenvectors of A.
89      """
90      n, m = X.shape
91      assert n == m
92
93      # First stage: transform to upper Hessenberg-matrix.
94      A = copy.deepcopy(X)
95      conv = False
96      k = 0
97
```

```
98      # Second stage: perform QR-transformations.
99      while (not conv) and (k < maxiter):
100         k += 1
101         Q, R = helpers.qr_factorize(A)
102         A = R.dot(Q)
103
104         conv = np.alltrue(np.isclose(np.tril(A, k=-1), np.zeros((n
              , n))))
105
106     if not conv:
107         warnings.warn("Convergence was not reached. Consider
              raising maxiter.")
108     if debug:
109         return k
110     Evals = A.diagonal()
111     order = np.abs(Evals).argsort()[::-1]
112     return Evals[order], Q[order, :]
113
114
115 def qrm2(X, maxiter=15000, debug=False):
116     """
117     First compute similar matrix in Hessenberg form, then compute
          the
118     Eigenvalues and Eigenvectors using the QR-Method.
119
120     Parameters:
121         - X: square numpy ndarray.
122     Returns:
123         - Eigenvalues of A.
124         - Eigenvectors of A.
```

```
125     """
126     n, m = X.shape
127     assert n == m
128 
129     # First stage: transform to upper Hessenberg-matrix.
130     A = lin.hessenberg(X)
131     conv = False
132     k = 0
133 
134     # Second stage: perform QR-transformations.
135     while (not conv) and (k < maxiter):
136         k += 1
137         Q, R = helpers.qr_factorize(A)
138         A = R.dot(Q)
139 
140         conv = np.alltrue(np.isclose(np.tril(A, k=-1), np.zeros((n
              , n))))
141 
142     if not conv:
143         warnings.warn("Convergence was not reached. Consider
              raising maxiter.")
144     if debug:
145         return k
146     Evals = A.diagonal()
147     order = np.abs(Evals).argsort()[::-1]
148     return Evals[order], Q[order, :]
149 
150 
151 def qrm3(X, maxiter=15000, debug=False):
152     """
```

```
153     First compute similar matrix in Hessenberg form, then compute
          the
154     Eigenvalues and Eigenvectors using the QR-Method.
155
156     Parameters:
157         - X: square numpy ndarray.
158     Returns:
159         - Eigenvalues of A.
160         - Eigenvectors of A.
161     """
162     n, m = X.shape
163     assert n == m
164
165     # First stage: transform to upper Hessenberg-matrix.
166     T = lin.hessenberg(X)
167
168     conv = False
169     k = 0
170
171     # Second stage: perform QR-transformations.
172     while (not conv) and (k < maxiter):
173         k += 1
174         Q, R = helpers.qr_factorize(T - T[n-1, n-1] * np.eye(n))
175         T = R.dot(Q) + T[n-1, n-1] * np.eye(n)
176
177         conv = np.alltrue(np.isclose(np.tril(T, k=-1), np.zeros((n
              , n))))
178
179     if not conv:
180         warnings.warn("Convergence was not reached. Consider
```

```
            raising maxiter.")
181     if debug:
182         return k
183     Evals = T.diagonal()
184     order = np.abs(Evals).argsort()[::-1]
185     return Evals[order], Q[order, :]
```

6.2 Analysis: Figures

```
1  import os
2  import copy
3  import pandas as pd
4  import numpy as np
5  import seaborn as sns
6  from scipy import linalg as lin
7  from scipy.stats import ortho_group
8  from matplotlib import pyplot as plt
9
10 datadir = os.path.join("analysis", "benchmarks.csv")
11 outpath = os.path.join("media", "plots")
12 trials = pd.read_csv(datadir, index_col=0)
13
14 trials.groupby(["algorithm", "dimension"]).iterations.describe()
15
16 # Boxplot iteration:
17 fig = plt.figure(figsize=(10, 5))
18 sns.boxplot(x="dimension", y="iterations", hue="algorithm", data=
     trials)
19 plt.yscale("log")
```

```
plt.title("Iterations needed before Convergence")
plt.savefig(os.path.join(outpath, "iterations_boxplot.png"))
plt.show()
plt.close()

# Boxplot elapsed time:
fig = plt.figure(figsize=(10, 5))
sns.boxplot(x="dimension", y="time", hue="algorithm", data=trials)
plt.title("Time needed before Convergence")
plt.ylabel("time (sec)")
plt.yscale('log')
plt.savefig(os.path.join(outpath, "time_boxplot.png"))
plt.show()
plt.close()

# Visualize Algorithm-Progress:
np.random.seed(42)
size = 5
Lambda = np.diag(np.random.randint(low=0, high=10, size=size))
G = ortho_group.rvs(dim=size)
X = np.dot(G, Lambda.dot(G.T))


def plot_factory(func):
    def plotter(savepath, **fig_kw):
        def algorithm_generator(*args, **kwargs):
            return func(*args, **kwargs)

        fig, ax = plt.subplots(nrows=2, ncols=2, **fig_kw)
        algorithm_iterator = algorithm_generator()
```

```
50          j = -1
51
52          for i, A in enumerate(algorithm_iterator):
53              if i in (0, 1, 10, 75):
54                  j += 1
55
56                  hm = ax[j // 2, j % 2].imshow(A,
57                                                cmap=plt.get_cmap('
                                                    seismic'),
58                                                vmin=-X.max(),
59                                                vmax=X.max())
60                  ax[j // 2, j % 2].set_yticks([])
61                  ax[j // 2, j % 2].set_xticks([])
62                  ax[j // 2, j % 2].set_title("Iteration: " + str(i)
                      )
63
64                  if i > 75:
65                      break
66
67          fig.subplots_adjust(right=0.8)
68          cbar_ax = fig.add_axes([0.85, 0.15, 0.05, 0.7])
69          fig.colorbar(hm, cax=cbar_ax)
70
71          sup_title = "Demonstrate {} on a {}x{} matrix".format(
72              func.__name__,
73              *X.shape)
74
75          fig.suptitle(sup_title)
76          fig.savefig(savepath)
77
```

```
78          return fig, ax
79
80      return plotter
81
82
83  @plot_factory
84  def jacobi():
85      """
86      Compute Eigenvalues and Eigenvectors for symmetric matrices
          using the
87      jacobi method.
88
89      Yields:
90          * A - 2D numpy array of current iteration step.
91      """
92      A = copy.deepcopy(X)
93      U = np.eye(A.shape[0])
94      L = np.array([1])
95      iterations = 0
96
97      while iterations < 5000:
98          L = np.abs(np.tril(A, k=0) - np.diag(A.diagonal()))
99          i, j = np.unravel_index(L.argmax(), L.shape)
100         alpha = 0.5 * np.arctan(2*A[i, j] / (A[i, i]-A[j, j]))
101
102         V = np.eye(A.shape[0])
103         V[i, i], V[j, j] = np.cos(alpha), np.cos(alpha)
104         V[i, j], V[j, i] = -np.sin(alpha), np.sin(alpha)
105
106         A = np.dot(V.T, A.dot(V))
```

```
107         U = U.dot(V)
108         iterations += 1
109         yield A
110
111
112 @plot_factory
113 def qrm1():
114     """
115     Create generator for transformed matrices after applying the
          QR-Method.
116
117     Yields:
118         - T: 2D-numpy array. Similar matrix to X.
119     """
120     # First stage: transform to upper Hessenberg-matrix.
121     T = copy.deepcopy(X)
122
123     k = 0
124     # Second stage: perform QR-transformations.
125     while k < 5000:
126         k += 1
127         Q, R = np.linalg.qr(T)
128         T = R.dot(Q)
129         yield T
130
131
132 @plot_factory
133 def qrm2():
134     """
```

```
135     Create generator for transformed matrices after applying the
          QR-Method.
136
137     Yields:
138         - T: 2D-numpy array. Similar matrix to X.
139     """
140     # First stage: transform to upper Hessenberg-matrix.
141     T = lin.hessenberg(X)
142
143     k = 0
144     # Second stage: perform QR-transformations.
145     while k < 5000:
146         if k == 0:
147             yield X
148         k += 1
149         Q, R = np.linalg.qr(T)
150         T = R.dot(Q)
151         yield T
152
153
154 @plot_factory
155 def qrm3():
156     """
157     First compute similar matrix in Hessenberg form, then compute
          the
158     Eigenvalues and Eigenvectors using the accelerated QR-Method.
159
160     Yields:
161         * T - 2D numpy array of current iteration step.
162     """
```

```
163     # First stage: transform to upper Hessenberg-matrix.
164     T = lin.hessenberg(X)
165     k = 0
166     n, _ = X.shape
167 
168     # Second stage: perform QR-transformations.
169     while k < 5000:
170         if k == 0:
171             yield X
172         k += 1
173         Q, R = np.linalg.qr(T - T[n-1, n-1] * np.eye(n))
174         T = R.dot(Q) + T[n-1, n-1] * np.eye(n)
175 
176         yield T
177 
178 
179 jacobi(os.path.join(outpath, "jacobi.png"))
180 qrm1(os.path.join(outpath, "qrm1.png"))
181 qrm2(os.path.join(outpath, "qrm2.png"))
182 qrm3(os.path.join(outpath, "qrm3.png"))
183 
184 plt.show()
185 plt.close()
```

6.3 Analysis: Unit tests

```
1 """
2 Automated tests for different algorithms.
3 """
```

```
4  import os
5  import sys
6  import numpy as np
7  from threading import Thread
8  import pandas as pd
9  from algorithms import eigen
10 from scipy.stats import ortho_group
11 from tqdm import tqdm
12 from functools import wraps
13
14
15 data_out = os.path.join("data", "accuracy_tests.csv")
16
17
18 class AlgoTest(object):
19     tests = {"algorithm": [],
20              "dimension": [],
21              "maxiter": [],
22              "failed": []}
23
24     def __init__(self, algo, dim, filepath, n_tests=1000, jobs=1,
25                  *args, **kwargs):
26         assert n_tests % jobs == 0
27         self.algorithm = self.__get_test_algorithm(algo, *args, **
           kwargs)
28         self.dim = dim
29         self.n = n_tests // jobs
30         self.failed = []
31         self.jobs = jobs
32         self.result = None
```

```
33          self.path = filepath
34          self.maxiter = kwargs.get("maxiter", None)
35
36          if not os.path.exists(self.path):
37              self.save(header=["algorithm", "dimension", "maxiter",
                    "failed"])
38
39      def __get_test_algorithm(self, algorithm, *args, **kwargs):
40          @wraps(algorithm)
41          def algo(X):
42              return algorithm(X, *args, **kwargs)
43          return algo
44
45      def __get_test_matrix(self):
46          """Return matrix with assosiated Eigenvalues."""
47          eigenvalues = np.random.uniform(size=self.dim)
48          eigenvectors = ortho_group.rvs(dim=self.dim)
49          Lambda = np.diag(eigenvalues)
50
51          matrix = np.dot(eigenvectors, Lambda).dot(eigenvectors.T)
52
53          order = np.abs(eigenvalues).argsort()[::-1]
54          return matrix, eigenvalues[order]
55
56      def __run_test(self):
57          """Run singular test."""
58          mat, true_eig = self.__get_test_matrix()
59          test_eig, _ = self.algorithm(mat)
60          test_res = np.alltrue(np.isclose(true_eig, test_eig))
61          self.failed.append(not test_res)
```

```
62
63      def __run_tests(self):
64          """Run multiple tests."""
65          for _ in range(self.n):
66              try:
67                  self.__run_test()
68              except (KeyboardInterrupt, SystemExit):
69                  self.__save()
70                  sys.exit(0)
71
72      def run(self):
73          """Distribute tests accross threads."""
74
75          threadlist = [None] * self.jobs
76
77          for i in range(self.jobs):
78              threadlist[i] = Thread(target=self.__run_tests, daemon
                =True)
79              threadlist[i].start()
80
81          for thread in threadlist:
82              thread.join()
83
84          self.result = sum(self.failed)
85          self.tests["algorithm"].append(self.algorithm.__name__)
86          self.tests["dimension"].append(self.dim)
87          self.tests["failed"].append(self.result)
88          if self.algorithm.__name__ == "jacobi":
89              self.tests["maxiter"].append(None)
90          else:
```

```
91              self.tests["maxiter"].append(self.maxiter)
92
93      def save(self, header=False):
94          if os.path.exists(self.path):
95                  print("Saving results.")
96          df = pd.DataFrame(self.tests)
97          with open(self.path, 'a') as f:
98              df.to_csv(f, header=header, index=False,
99                        columns=["algorithm", "dimension", "maxiter"
                            , "failed"])
100
101
102 # Unit tests
103 if __name__ == "__main__":
104     # Define Flags.
105     JOBS = 20
106     MAXITER = (10, 100, 1000, 10000)
107     DIMS = range(3, 8)
108     ALGOS = {'jacobi': eigen.jacobi,
109              'qrm': eigen.qrm,
110              'qrm2': eigen.qrm2,
111              'qrm3': eigen.qrm3}
112
113     # Define parameters of all runs.
114     parameters = []
115     for algo in ALGOS.values():
116         for maxiter in MAXITER:
117             for dim in range(3, 8):
118                 param = (algo,
```

```
119                         maxiter if algo.__name__ != "jacobi" else
                               None,
120                         dim)
121                 parameters.append(param)
122
123     # Check progress of previous runs.
124     if os.path.exists(data_out):
125         required = [(algo.__name__, m, dim) for (algo, m, dim) in
                      parameters]
126         required = set(required)
127
128         progress = pd.read_csv(data_out)
129         done = []
130         for (a, d, m, _) in progress.values:
131             if np.isnan(m):
132                 param = (a, d, None)
133             else:
134                 param = (a, d, m)
135             done.append(param)
136         done = set(done)
137
138         to_do = required.difference(done)
139         parameters = [(ALGOS[a], m, d) for a, m, d in to_do]
140
141     for algo, maxiter, dim in tqdm(parameters):
142                 algo_test = AlgoTest(filepath=data_out,
143                                      algo=algo,
144                                      dim=dim,
145                                      maxiter=maxiter,
146                                      jobs=JOBS)
```

```
147                     algo_test.run()
148
149         algo_test.save()
```

7 References

[Börm and Mehl, 2012] Börm, S. and Mehl, C. (2012). *Numerical Methods for Eigenvalue Problems*. Walter de Gruyter GmbH & Co.KG, Berlin/Boston.

[Gentle, 1998] Gentle, J. E. (1998). *Numerical Linear Algebra for Applications in Statistics*. Springer Science and Business Media, New York.

[Härdle and Simar, 2015] Härdle, W. K. and Simar, L. (2015). *Applied Multivariate Statistical Analysis*. Springer-Verlag Gmbh, Berlin, Heidelberg.

[Hunter, 2007] Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95.

[McKinney, 2010] McKinney, W. (2010). Data structures for statistical computing in python. In van der Walt, S. and Millman, J., editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56.

[van Rossum, 1995] van Rossum, G. (1995). Python tutorial. Technical Report CS-R9526, Centrum voor Wiskunde en Informatica (CWI), Amsterdam.