



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 27, 2018






## Contents

|          |                                   |           |
|----------|-----------------------------------|-----------|
| <b>1</b> | <b>Motivation</b>                 | <b>3</b>  |
| <b>2</b> | <b>Similarity Transformations</b> | <b>4</b>  |
| 2.1      | Householder-Reflections . . . . . | 5         |
| 2.2      | Givens-Rotations . . . . .        | 6         |
| <b>3</b> | <b>Algorithms</b>                 | <b>7</b>  |
| 3.1      | Jacobi Method . . . . .           | 7         |
| 3.2      | QR-Method . . . . .               | 9         |
| 3.2.1    | Hessenberg Variant . . . . .      | 9         |
| 3.2.2    | Accelerated Variant . . . . .     | 11        |
| <b>4</b> | <b>Analysis</b>                   | <b>11</b> |
| 4.1      | Accuracy . . . . .                | 11        |
| 4.2      | Efficiency . . . . .              | 11        |
| <b>5</b> | <b>Conclusion</b>                 | <b>11</b> |
| <b>6</b> | <b>Appendix</b>                   | <b>12</b> |
| 6.1      | Eigenvalue Routines . . . . .     | 12        |
| 6.2      | Analysis: Figures . . . . .       | 21        |
| 6.3      | Analysis: Unit tests . . . . .    | 27        |

## List of Tables

|   |   |   |
|---|---|---|
| 1 | Unit tests accross matrix-sizes . . . . . | 9 |
|---|---|---|

## List of Figures

|   |  |    |
|---|--|----|
| 1 | Progress Jacobi-Method          | 7  |
| 2 | Progress basic QR-Method        | 8  |
| 3 | Progress Hessenberg-QR-Method   | 9  |
| 4 | Progress Accelerated QR-Method  | 10 |
| 5 | Unit-tests: Iterations          | 11 |
| 6 | Decision process of final eigenvalue routine   | 12 |

## List of Algorithms

|   |        |    |
|---|--------|----|
| 1 | jacobi | 7  |
| 2 | QRM1   | 8  |
| 3 | QRM2   | 9  |
| 4 | QRM3   | 10 |

# 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 impractical. 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  $\lambda$  is a scalar, such that

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

then  $v$  is called an *eigenvector* and  $\lambda$  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  $\lambda$ .

When confronted with high-dimensional data an analyst often wishes to find a lower-dimensional representation, while conserving as much of the structure as

possible.

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. 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 choosing the algorithm that is most fit for the underlying problem.

## 2 Similarity Transformations

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

$$A = P^{-1}BP. \tag{3}$$

This transformation defined in 3 is also called a *similarity transformation*. 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*.

## 2.1 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

$$x = c_1u + c_2v$$

for some scalars  $c_1$  and  $c_2$ . The vector

$$\tilde{x} = -c_1u + c_2v$$

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

$$Q = I - 2uu'. \quad (4)$$

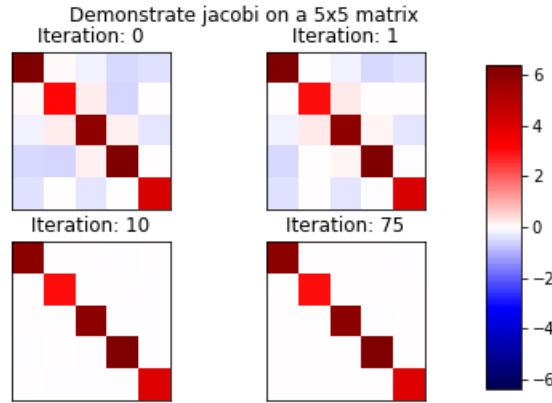
$$\begin{aligned} Qx &= c_1u + c_2v - 2c_1uuu' - 2c_2vuu' \\ &= c_1u + c_2v - 2c_1u'uu - 2c_2u'vu \\ &= -c_1u + c_2v \\ &= \tilde{x} \end{aligned}$$

## 2.2 Givens-Rotations

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.

$$V_{pq}(\theta) = \begin{bmatrix} 1 & & & & \\ & \ddots & & & \\ & & \cos \theta & \sin \theta & \\ & & & \ddots & \\ & & -\sin \theta & \cos \theta & \\ & & & & \ddots & \\ & & & & & 1 \end{bmatrix} \quad (5)$$

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

Figure 1: Progress Jacobi-Method 

### 3 Algorithms

#### 3.1 Jacobi Method

---

##### Algorithm 1 jacobi

---

**Require:** symmetric matrix  $A$

**Ensure:**  $0 < precision < 1$

**initialize:**  $L \leftarrow A$ ;  $U \leftarrow I$ ;  $L_{max} \leftarrow 1$

1: **while**  $L_{max} > precision$  **do**

2:    Find indices  $i, j$  of largest value in lower triangle of  $abs(L)$

3:     $L_{max} \leftarrow L_{i,j}$

4:     $\alpha \leftarrow \frac{1}{2} \cdot \arctan\left(\frac{2A_{i,j}}{A_{i,i} - A_{j,j}}\right)$

5:     $V \leftarrow I$

6:     $V_{i,i}, V_{j,j} \leftarrow \cos \alpha$ ;  $V_{i,j}, V_{j,i} \leftarrow -\sin \alpha, \sin \alpha$

7:     $A \leftarrow V'AV$ ;  $U \leftarrow UV$

8: **end while**

9: **return**  $diag(A)$ ,  $U$

---



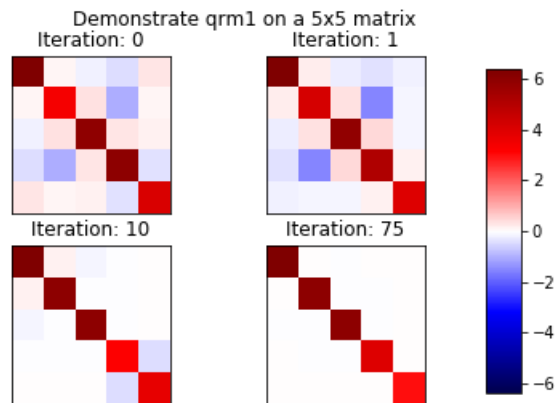
**Algorithm 2** QRM1**Require:** square matrix  $A$ **initialize:**  $conv \leftarrow False$ 1: **while** not  $conv$  **do**2:    $Q, R \leftarrow$  QR-Factorization of  $A$ 3:    $A \leftarrow RQ$ 4:   **if**  $A$  is diagonal **then**5:      $conv \leftarrow True$ 6:   **end if**7: **end while**8: **return**  $diag(A), Q$ Figure 2: Progress basic QR-Method 

Figure 3: Progress Hessenberg-QR-Method

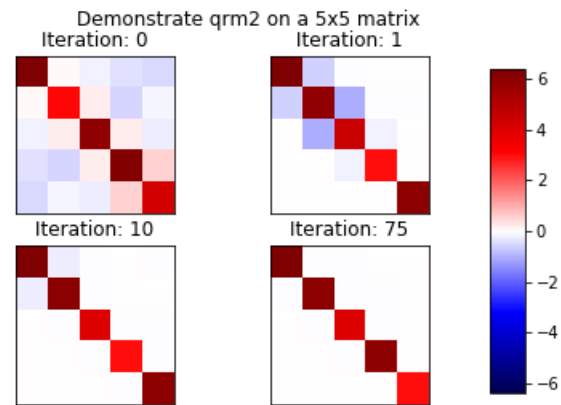


Table 1: Unit tests accross matrix-sizes

|         |       |
|---------|-------|
| awesome | sauce |
| nothing | to    |
| see     | here  |

### 3.2 QR-Method

#### 3.2.1 Hessenberg Variant

|  |
|--|
| <b>Algorithm 3</b> QRM2                |
| <b>Require:</b> square matrix $A$      |
| 1: $A \leftarrow \text{hessenberg}(A)$ |
| 2: continue with: QRM1(A)              |

**Algorithm 4** QRM3**Require:** square matrix  $A \in \mathbb{R}^{p \times p}$ 

```

1:  $T \leftarrow \text{hessenberg}(A)$ ,  $\text{conv} \leftarrow \text{False}$ 
2: while not  $\text{conv}$  do
3:    $Q, R \leftarrow \text{QR-Factorization of } T - t_{p-1,p-1}I$ 
4:    $T \leftarrow RQ + t_{p-1,p-1}I$ 
5:   if  $T$  is diagonal then
6:      $\text{conv} \leftarrow \text{True}$ 
7:   end if
8: end while
9: return  $\text{diag}(T)$ ,  $Q$ 

```

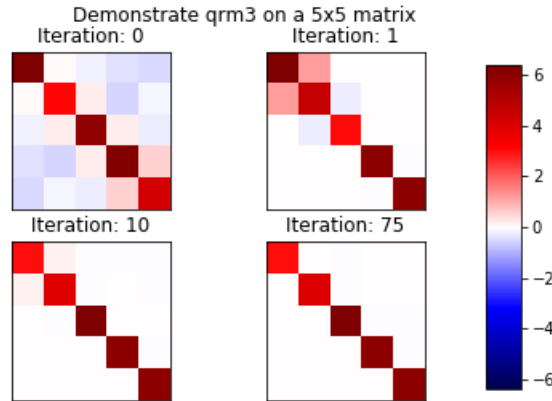
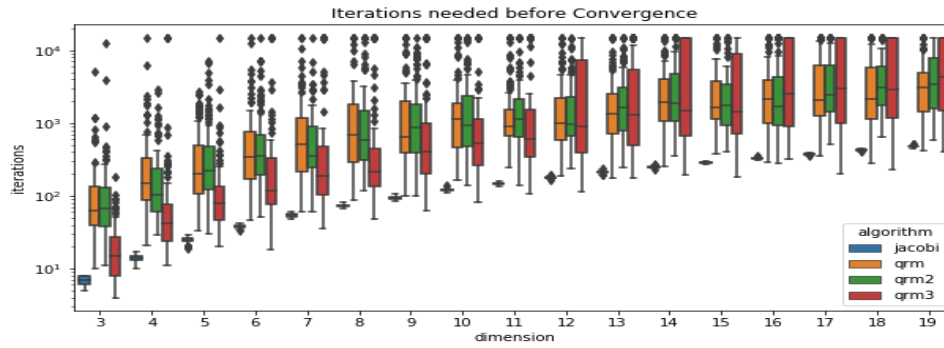
Figure 4: Progress Accelerated QR-Method 

Figure 5: Unit-tests: Iterations 

### 3.2.2 Accelerated Variant

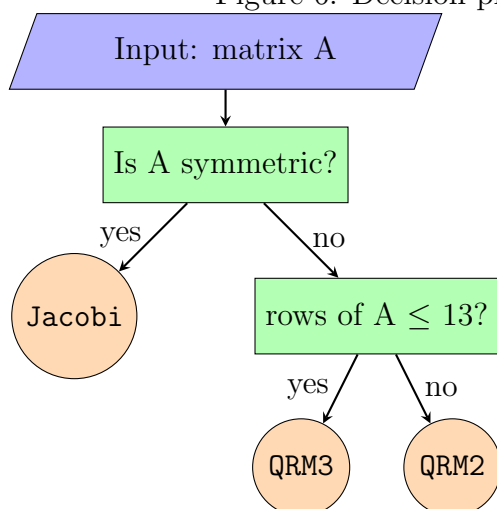
## 4 Analysis

### 4.1 Accuracy

### 4.2 Efficiency

## 5 Conclusion

Figure 6: Decision process of final eigenvalue routine



## 6 Appendix

### 6.1 Eigenvalue Routines

```

1 import numpy as np
2 import copy
3
4
5 def hreflect1D(x):
6     """
7     Calculate Householder reflection:  $Q = I - 2*uu'$ .
8
9     Parameters:
10         X: numpy array.
11
12     Returns:
13         Qx: reflected vector.
14         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
41             Hessenberg-
42             matrix.
43
44     Returns:

```

```

44     Q: square numpy ndarray, same shape as X. Rotation matrix.
45     R: square numpy ndarray, same shape as X. Upper triangular
      matrix if
46     offset is 0, Hessenberg-matrix if offset is 1.
47     """
48     assert offset in [0, 1]
49     assert type(X) == np.ndarray
50     assert X.shape[0] == X.shape[1]
51
52     R = copy.deepcopy(X)
53     Q = np.eye(X.shape[0])
54
55     for i in range(X.shape[0]-offset):
56         Pi = np.eye(R.shape[0])
57         _, Qi = hreflect1D(R[i+offset:, i])
58         Pi[i+offset:, i+offset:] = Qi
59
60         Q = Pi.dot(Q)
61         R = Pi.dot(R)
62
63     return Q.T, R

```

```

1     """
2     Algorithms for solving eigenvalue problems.
3
4     1. Compute diagonalization of 2x2 matrices via jacobi iteration.
5     2. Generalize Jacobi iteration for symmetric matrices.
6     """
7     import numpy as np
8     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
46     value
47     U - 2D numpy array with associated eigenvectors (column).
48     """
49
50     assert 0 < precision < 1.
51     assert type(X) == np.ndarray
52     n, m = X.shape
53     assert n == m
54     assert all(np.isclose(X - X.T, np.zeros(n)).flatten())
55
56     A = copy.deepcopy(X)
57     U = np.eye(A.shape[0])
58     L = np.array([1])
59     iterations = 0
60
61     while L.max() > precision:
62
63         L = np.abs(np.tril(A, k=0) - np.diag(A.diagonal()))
64         i, j = np.unravel_index(L.argmax(), L.shape)
65         alpha = 0.5 * np.arctan(2*A[i, j] / (A[i, i]-A[j, j]))
66
67         V = np.eye(A.shape[0])
68         V[i, i], V[j, j] = np.cos(alpha), np.cos(alpha)
69         V[i, j], V[j, i] = -np.sin(alpha), np.sin(alpha)
70
71         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
87     Returns:
88         - Eigenvalues of A.
89         - Eigenvectors of A.
90     """
91     n, m = X.shape
92     assert n == m
93
94     # First stage: transform to upper Hessenberg-matrix.
95     A = copy.deepcopy(X)
96     conv = False
97     k = 0
```

```

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
105             , n))))
106
107     if not conv:
108         warnings.warn("Convergence was not reached. Consider
109             raising maxiter.")
110
111     if debug:
112         return k
113
114     Evals = A.diagonal()
115     order = np.abs(Evals).argsort()[::-1]
116     return Evals[order], Q[order, :]
117
118 def qrm2(X, maxiter=15000, debug=False):
119     """
120     First compute similar matrix in Hessenberg form, then compute
121     the
122     Eigenvalues and Eigenvectors using the QR-Method.
123
124     Parameters:
125         - X: square numpy ndarray.
126
127     Returns:
128         - Eigenvalues of A.
129         - 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
141             , n))))
142
143     if not conv:
144         warnings.warn("Convergence was not reached. Consider
145             raising maxiter.")
146
147     if debug:
148         return k
149
150     Evals = A.diagonal()
151     order = np.abs(Evals).argsort()[::-1]
152     return Evals[order], Q[order, :]

```

```

151 def qrm3(X, maxiter=15000, debug=False):
152     """

```

```
153     First compute similar matrix in Hessenberg form, then compute
154     the
155     Eigenvalues and Eigenvectors using the QR-Method.
156
157     Parameters:
158     - X: square numpy ndarray.
159
160     Returns:
161     - Eigenvalues of A.
162     - Eigenvectors of A.
163
164     """
165     n, m = X.shape
166     assert n == m
167
168     # First stage: transform to upper Hessenberg-matrix.
169     T = lin.hessenberg(X)
170
171     conv = False
172     k = 0
173
174     # Second stage: perform QR-transformations.
175     while (not conv) and (k < maxiter):
176         k += 1
177         Q, R = helpers.qr_factorize(T - T[n-1, n-1] * np.eye(n))
178         T = R.dot(Q) + T[n-1, n-1] * np.eye(n)
179
180         conv = np.alltrue(np.isclose(np.tril(T, k=-1), np.zeros((n
181             , n))))
182
183     if not conv:
184         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=
19             trials)
19 plt.yscale("log")
```

```

20 plt.title("Iterations needed before Convergence")
21 plt.savefig(os.path.join(outpath, "iterations_boxplot.png"))
22 plt.show()
23 plt.close()
24
25 # Boxplot elapsed time:
26 fig = plt.figure(figsize=(10, 5))
27 sns.boxplot(x="dimension", y="time", hue="algorithm", data=trials)
28 plt.title("Time needed before Convergence")
29 plt.ylabel("time (sec)")
30 plt.yscale('log')
31 plt.savefig(os.path.join(outpath, "time_boxplot.png"))
32 plt.show()
33 plt.close()
34
35 # Visualize Algorithm-Progress:
36 np.random.seed(42)
37 size = 5
38 Lambda = np.diag(np.random.randint(low=0, high=10, size=size))
39 G = ortho_group.rvs(dim=size)
40 X = np.dot(G, Lambda.dot(G.T))
41
42
43 def plot_factory(func):
44     def plotter(savepath, **fig_kw):
45         def algorithm_generator(*args, **kwargs):
46             return func(*args, **kwargs)
47
48         fig, ax = plt.subplots(nrows=2, ncols=2, **fig_kw)
49         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('
58                                           seismic'),
59                                           vmin=-X.max(),
60                                           vmax=X.max())
61
62             ax[j // 2, j % 2].set_yticks([])
63             ax[j // 2, j % 2].set_xticks([])
64             ax[j // 2, j % 2].set_title("Iteration: " + str(i)
65                                         )
66
67             if i > 75:
68                 break
69
70
71     fig.subplots_adjust(right=0.8)
72     cbar_ax = fig.add_axes([0.85, 0.15, 0.05, 0.7])
73     fig.colorbar(hm, cax=cbar_ax)
74
75     sup_title = "Demonstrate {} on a {}x{} matrix".format(
76         func.__name__,
77         *X.shape)
```



```

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
87     using the
88     jacobi method.
89
90     Yields:
91         * A - 2D numpy array of current iteration step.
92     """
93     A = copy.deepcopy(X)
94     U = np.eye(A.shape[0])
95     L = np.array([1])
96     iterations = 0
97
98     while iterations < 5000:
99         L = np.abs(np.tril(A, k=0) - np.diag(A.diagonal()))
100         i, j = np.unravel_index(L.argmax(), L.shape)
101         alpha = 0.5 * np.arctan(2*A[i, j] / (A[i, i]-A[j, j]))
102
103         V = np.eye(A.shape[0])
104         V[i, i], V[j, j] = np.cos(alpha), np.cos(alpha)
105         V[i, j], V[j, i] = -np.sin(alpha), np.sin(alpha)
106
107         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
116     QR-Method.
117
118     Yields:
119     - T: 2D-numpy array. Similar matrix to X.
120     """
121     # First stage: transform to upper Hessenberg-matrix.
122     T = copy.deepcopy(X)
123
124     k = 0
125     # Second stage: perform QR-transformations.
126     while k < 5000:
127         k += 1
128         Q, R = np.linalg.qr(T)
129         T = R.dot(Q)
130         yield T
131
132 @plot_factory
133 def qrm2():
134     """
```

```

135     Create generator for transformed matrices after applying the
136     QR-Method.
137
138     Yields:
139     - T: 2D-numpy array. Similar matrix to X.
140     """
141     # First stage: transform to upper Hessenberg-matrix.
142     T = lin.hessenberg(X)
143
144     k = 0
145     # Second stage: perform QR-transformations.
146     while k < 5000:
147         if k == 0:
148             yield X
149             k += 1
150             Q, R = np.linalg.qr(T)
151             T = R.dot(Q)
152             yield T
153
154 @plot_factory
155 def qrm3():
156     """
157     First compute similar matrix in Hessenberg form, then compute
158     the
159     Eigenvalues and Eigenvectors using the accelerated QR-Method.
160
161     Yields:
162     * T - 2D numpy array of current iteration step.
163     """

```

```

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 numpy as np
6 import threading
7 import pandas as pd
8 from algorithms import eigen
9 from scipy.stats import ortho_group
10 from tqdm import trange, tqdm
11
12 data_out = os.path.join("data", "accuracy_tests.csv")
13
14
15 def get_test_matrix(dim):
16     """Return matrix with associated Eigenvalues."""
17     eigenvalues = np.random.uniform(size=dim)
18     eigenvectors = ortho_group.rvs(dim=dim)
19     Lambda = np.diag(eigenvalues)
20
21     matrix = np.dot(eigenvectors, Lambda).dot(eigenvectors.T)
22
23     order = np.abs(eigenvalues).argsort()[::-1]
24     return matrix, eigenvalues[order]
25
26
27 def test_algo(algo, Ntests=1000, dim=3, *args, **kwargs):
28     """
29     Test routine that allows for threading. Note that the
30     variables:
31     failed, critical and problematic need to be defined in the
32     enveloping or
33     global scope beforehand.
```

```
32
33 Parameters:
34     - algo: algorithm to be tested
35     - Ntests: number of tests to compute
36     - dim: dimensions of matrix
37     - *args, **kwargs: additional arguments to be passed to
      algo.
38
39 Returns:
40     - None, but will update the variables failed, critical and
      problematic.
41     + failed: number of failed tests
42     + critical: number of ZeroDivisionErrors
43     + problematic: list of numpy arrays which led to wrong
      eigenvalues.
44
45 """
46
47 global failed
48 global critical
49 global problematic
50
51 for _ in range(Ntests):
52     try:
53         A, true_eig = get_test_matrix(dim=dim)
54         my_eig, _ = algo(A, *args, **kwargs)
55         assert np.alltrue(np.isclose(my_eig, true_eig))
56
57     except AssertionError:
58         failed += 1
59         problematic.append(A)
```

```
59     except ZeroDivisionError:
60         critical += 1
61
62
63 def threaded_tests(algo, N, nWorkers=10, verbose=True, *args, **
64     kwargs):
65     global failed
66     global critical
67     global problematic
68
69     assert N % nWorkers == 0
70
71     n = N // nWorkers
72     threadlist = [None] * nWorkers
73
74     for i in range(nWorkers):
75         threadlist[i] = threading.Thread(target=test_algo,
76                                         args=(algo, n, *args))
77
78         threadlist[i].start()
79
80     for i in range(nWorkers):
81         threadlist[i].join()
82
83     logstr = """
84     {} out of {} tests failed.
85     {} tests failed critically.
86     """.format(failed, N, critical)
87
88     if verbose:
89         print(logstr)
```

```
88
89
90 # Tests
91 results = {
92     "algorithm": [],
93     "dimension": [],
94     "maxiter": [],
95     "failed": []}
96
97 for algo in tqdm([eigen.jacobi, eigen.qrm, eigen.qrm2, eigen.qrm3
98 ]):
99     for dim in trange(3, 15):
100         for maxiter in 1000, 10000, 100000:
101             if algo.__name__ == eigen.jacobi:
102                 maxiter = 1e-6
103
104                 failed = 0
105                 critical = 0
106                 problematic = []
107                 threaded_tests(algo, 1000, 20, False, dim, maxiter)
108                 results["algorithm"].append(algo.__name__)
109                 results["dimension"].append(dim)
110                 results["maxiter"].append(maxiter)
111                 results["failed"].append(failed)
112
113 test_data = pd.DataFrame(results)
114 test_data.to_csv(data_out)
```



## References

- [1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LA-PACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.
- [2] Seffen Börm and Christian Mehl. *Numerical Methods for Eigenvalue Problems*. Walter de Gruyter GmbH & Co.KG, Berlin/Boston, 2012.
- [3] Wolfgang K. Härdle and Léopold Simar. *Applied Multivariate Statistical Analysis*. Springer-Verlag GmbH, Berlin, Heidelberg, 2015.