

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

CONTENTS

Contents

List of Figures						
Li	st of	Tables	2			
1	Mo	ivation	3			
2	Sim	ilarity Transformations	5			
3	Alg	orithms	6			
	3.1	Jacobi Method	6			
	3.2	QR-Method	8			
		3.2.1 Hessenberg Variant	10			
		3.2.2 Accelerated Variant	11			
4	Ana	lysis	13			
	4.1	Accuracy	14			
	4.2	Efficiency	15			
5	Cor	clusion	17			
6	App	endix	18			
	6.1	Householder-Reflections	18			
	6.2	Givens-Rotations	20			
	6.3	Code	22			
		6.3.1 Eigenvalue Routines	22			
		6.3.2 Analysis: Figures	31			
		6.3.3 Analysis: Unit tests	37			
7	Ref	erences	42			

List of Algorithms

1	jacobi 🝳	8
2	QRM1 •	10
3	QRM2 •	11
4	QRM3 •	13
List	of Figures	
1	Progress Jacobi-Method 🚨	8
2	Progress basic QR-Method •	9
3	Progress Hessenberg-QR-Method \mathbf{Q}	11
4	Progress Accelerated QR-Method Q	13
5	Unit-tests: Iterations 🚨	15
6	Decision process of the final eigenvalue routine \mathbf{Q}	16
7	Rotation of $x \ \square$	20
List	of Tables	
1	Failed unit tests accross matrix-sizes Q	14

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 algorithm's 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,

$$det (A - \lambda I) = 0. (2)$$

Each eigenvalue is associated with an eigenvector. Geometrically speaking, we require a vector which, when multiplied by matrix A, will not get rotated but only elongated by the eigenvalue λ (Börm and Mehl, 2012, p. 13).

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 data's 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

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

The solution to this optimization problem is given by

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

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, p. 234).

A second use for Eigenvalues lies in Fisher's Linear Discriminant Analysis for classification problems. In this setting we would like to find the rotation that is best suited to separate the classes. To this effect we try to maximize the between-class-variance while simoultaneously trying to minimize the within-class variance. This is equivalent to maximizing the so called Rayleigh-coefficient

$$maxJ = \frac{J_b(w)}{J_w(w)} = \frac{w'S_bw}{w'S_ww}.$$
 (5)

The between and within class variance matrices are defined as $S_b = \sum_{c=1}^{C} (\mu_c - \mu)(\mu_c - \mu)'$ and $S_w = \sum_{c=1}^{C} \sum_{i \in c} (x_i - \mu_c)(x_i - \mu_c)'$. The solution to this maximization problem is given by

$$S_B^{\frac{1}{2}} S_W^{-1} S_B^{\frac{1}{2}} v = \lambda v, \tag{6}$$

which again is an eigenvalue problem of a square, symmetric matrix and can be readily computed (Duda et al., 2000, p. 219).

In the following we will try to extract the eigenvalues of matrices by reformulating any complicated eigenvalue problem into a straightforward one by diagonalizing the matrix in question, without altering the eigenvalues we would like to compute. In section two we provide the mathematical tools, which allow us to do so. 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, all plots have been produced with the matplotlib-module and

the custom algorithms rely solely on modules from Python's scipy environment for scientific computing (van Rossum, 1995; Hunter, 2007; McKinney, 2010). All scripts are available online. 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 as entries on the main diagonal. 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

$$A = P^{-1}BP. (7)$$

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

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

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. For our custom implementations we will make use of Givens-Rotations and Householder Reflectors (for a detailed discussion consider the appendix).

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

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

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 a matrix that performs a Givens-Rotation. Such a rotation is performed by an identity matrix with four of its elements replaced, such that

$$V_{pq}(\theta) = \begin{bmatrix} 1 & & & & & \\ & \ddots & & & \\ & & \cos \theta & & \sin \theta & & \\ & & & \ddots & & \\ & & -\sin \theta & & \cos \theta & & \\ & & & \ddots & & \\ & & & & 1 \end{bmatrix}$$
 (8)

3.1 Jacobi Method 3 ALGORITHMS

The Jacobi iteration is defined as

$$A^{(k)} = V'_{p_k,q_k}(\theta_k)A^{(k-1)}V_{p_k,q_k}(\theta_k), \tag{9}$$

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. Also, note that $A^{(k)}$ and $A^{(k-1)}$ are similar, since $V'_{p_k,q_k}(\theta_k)$ is an orthogonal matrix for every step k. The benefit of matrices as in (8) is, that it is easy to choose an angle θ_k in order to introduce a zero in a single multiplication step. 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

$$a_{pq}^{(k-1)}(\cos^2\theta - \sin^2\theta) + \left(a_{pp}^{(k-1)} - a_{qq}^{(k-1)}\right)\cos\theta\sin\theta = 0.$$
 (10)

We can use the trigonometric identities

$$\cos(2\theta) = \cos^2\theta \sin^2\theta$$

$$\sin(2\theta) = 2\cos\theta\sin\theta,$$

in (10) we have

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

From this we can retrieve the angle and obtain the rotation matrix in each iteration (Gentle, 1998, p. 128).

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, p. 128).

Figure 1: Progress Jacobi-Method Q Demonstrate jacobi on a 5x5 matrix

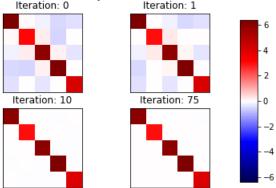


Figure 1 visualizes the progress of the Jacobi-method on a symmetric 5×5 matrix. As we can see, in the first iteration the element a_{43} is eliminated. In the subsequent operations the Jacobi-method continues to eliminate any non-zero entries on the off-diagonal until the algorithm convergences after 10 iterations.

```
Algorithm 1 jacobi
```

```
Require: symmetric matrix A
```

Ensure: 0 < precision < 1

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

- 1: while $L_{max} > precision$ do
- Find indices i, j of largest value in lower triangle of abs(L)2:
- $L_{max} \leftarrow L_{i,j}$ 3:
- $\theta \leftarrow \frac{1}{2} \cdot \arctan(\frac{2A_{i,j}}{A_{i,i} A_{j,j}})$ 4:
- $V_{i,i}, V_{j,j} \leftarrow \cos \theta; V_{i,j}, V_{j,i} \leftarrow -\sin \theta, \sin \theta$ $A \leftarrow V'AV; U \leftarrow UV$
- 8: end while
- 9: **return** diag(A), U

3.2QR-Method

The most widely used algorithm to extract eigenvalues is the so called QR-method. The most important advantage of the QR-method over the Jacobi-method is that it can be applied to non-symmetric matrices. Note however, that it is simpler for symmetric matrices, since the eigenvalues are real-valued.

Figure 2: Progress basic QR-Method Demonstrate qrm1 on a 5x5 matrix Iteration: 0 Iteration: 1

Iteration: 10 Iteration: 75

Iteration: 75

Iteration: 75

The QR-method to extract the eigenvalues of a square matrix $A \in \mathbb{R}^{n \times n}$ is performed by first computing the titular QR decomposition of A.

$$A = QR, (11)$$

where Q is an orthogonal and R is an upper triangular matrix. Then define the QR iteration as

$$A^{k} = Q'_{k-1}A_{k-1}Q_{k-1} = R_{k-1}Q_{k-1}$$
(12)

Note hereby that all matrices in the sequence $\{A_k\}$ share the same eigenvalues, since this procedure is a similarity transformation due to Q's orthogonality (Börm and Mehl, 2012, p. 121). Additionally, for practical applications it is preferable to compute the QR-iteration in the way shown at the rightmost part of equation (12). Although, mathematically, each statement is exactly identical there is a non-negligible difference due to computational imperfections and limited machine precision. The reason is that the computation of $Q'_{k-1}A_{k-1}Q_{k-1}$, obviously, requires two matrix multiplications whereas the result of $R_{k-1}Q_{k-1}$ can be readily obtained by one. When combining multiple steps over a long sequence of QR iterations the additional computations lead to additional rounding errors, which can have an influence on the accuracy of the obtained results. Besides, less computations lead of course to a faster procedure in general.

Figure 3.2 visualizes the progress of the basic QR-method on the same 5×5 matrix as in Figure 1. Compared to the Jacobi-method it does not explicitly pick a single element that will be eliminated per iteration. Instead, the QR-method extracts the eigenvalues by a process that is called chasing. By that we mean that alternating steps are being performed, which create non-zero eintries in positions (i+2,i), (i+3,i) and (i+3,i+1) and restore them to zero, as the nonzero entries are moved farther down the matrix (Gentle, 1998, p. 130). We can also see that compared to the Jacobi-Method, so far, the QR-algorithm lacks in speed. Where the Jacobi-method was almost done diagonalizing the matrix in iteration 10, the basic QR-algorithm still had multiple non-zero entries left. Thus we would like to make minor improvements on the algorithm's efficiency.

```
Algorithm 2 QRM1 \bigcirc

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
```

3.2.1 Hessenberg Variant

In order to speed up the QR-method it is advisable to transform the matrix to its upper Hessenberg form. A matrix A is of upper Hessenberg form if it is upper triangular except for the first subdiagonal, which may be non-zero. In particular $a_{ij} = 0 \ \forall i > j+1$:

$$\begin{bmatrix} X & X & X & \dots & X & X \\ X & X & X & \dots & X & X \\ 0 & X & X & \dots & X & X \\ 0 & 0 & X & \dots & X & X \\ \vdots & \vdots & & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & X & X \end{bmatrix}$$

Figure 3: Progress Hessenberg-QR-Method

Demonstrate qrm2 on a 5x5 matrix
Iteration: 0

Iteration: 1

Iteration: 10

Iteration: 75

Iteration: 75

A matrix can be reduced to *Hessenberg* form in a finite number of similarity transformations using Householder transformations or Givens rotations. For symmetric matrices the transformation into a *Hessenberg*-form results in a tridiagonal matrix. But even for non-symmetric matrices, the *Hessenberg*-form allows a large saving in subsequent computations. After the transformation we can deploy the previously defined *QR*-method (Gentle, 1998, p. 130).

Figure 3 visualizes the progress of the Hessenberg variant of the QR-method. In order to make it comparable to the previous algorithms the same 5×5 -dimensional matrix is being evaluated. We can readily see, that the transformation to Hessenberg-form results in a tridiagonal matrix. This facilitates computations and explains the vastly improved resulting matrix after 10 iterations compared to the basic QR-method. However, it still does not match the progress of the Jacobi-method after the same number of iterations.

Algorithm 3 QRM2 Q

Require: square matrix A

- 1: $A \leftarrow \mathtt{hessenberg}(A)$
- 2: continue with: QRM1(A)

3.2.2 Accelerated Variant

We could already improve the QR-method and cut down on computational cost. However, we still cannot match the results of the Jacobi method. To this effect we present one additional adjustment on the QR-method that aims to improve convergence speed. The

general idea is, that we deliberately create an additional zero entry on the main diagonal by subtracting a scalar on each element, perform the QR-iteration and finally undo the subtraction. In particular, we define

$$T^{m} = \begin{bmatrix} \alpha_{1}^{m} & \beta_{1}^{m} & 0 & 0 & \dots & 0 \\ \beta_{1}^{m} & \alpha_{2}^{m} & \beta_{2}^{m} & & & & \\ 0 & \beta_{2}^{m} & \alpha_{3}^{m} & \beta_{3}^{m} & & \vdots & & \\ & \ddots & \ddots & \ddots & & & \\ & & \beta_{n-2}^{m} & \alpha_{n-1}^{m} & \beta_{n-1}^{m} \\ 0 & & & \beta_{n-1}^{m} & \alpha_{n}^{m} \end{bmatrix}$$

$$(13)$$

$$T^{m} = T - t_{n,n}I$$

$$T^{m} = QR$$

$$T^{m+1} = T^{m} + t_{n,n}I$$

After this we can define the accelerated iteration step as

$$R_m = Q_m' \left(T^m - \alpha_n^m I \right) \tag{14}$$

$$T^{m+1} = Q_m' \left(T^m - \alpha_n^m I \right) Q_m + \alpha_n^m I \tag{15}$$

$$=Q_m'T^mQ_m\tag{16}$$

Again T^{m+1} is similar to T^m .

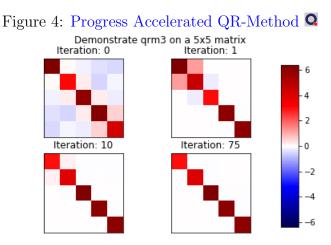


Figure 4 visualizes the progress of the accelerated QR-method. For comparability, the matrix used, is the same 5×5 matrix as before. Most notably is, that the accelerated method still performs worse than the Jacobi method. So far results seem similar compared to the Hessenberg variant of the QR-method. The case can be made that the accelerated method performs considerably better than the basic QR-method and slightly better than the Hessenberg variant after 10 iterations. Howbeit, this claim warrants further analysis.

```
Algorithm 4 QRM3 Q
```

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

1: T \leftarrow \text{hessenberg}(A), conv \leftarrow False

2: while not 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: conv \leftarrow True

7: end if

8: end while

9: return diag(T), Q
```

4 Analysis

In order to test the quality of our implemented algorithms we will follow a two pronged approach. First, we will scrutinize the quality of the obtained results. Once we can be sure that the implemented routines are reliable or, to be more precise, under which conditions

4.1 Accuracy 4 ANALYSIS

our routines are reliable we can analyze their convergence behavior and compare the efficiency across algorithms in terms of computational cost.

4.1 Accuracy

If we wish to investigate the accuracy of our algorithms, we first need to define what we mean by that. We set up an environment of unit tests that compare the absolute difference between the vector of true eigenvalues of a matrix and the vector of computed eigenvalues by our custom algorithms. If any element of this L_1 -norm is larger than a threshold of 10^{-5} we consider the test as failed. In order to obtain matrices with known eigenvalues we will reverse the spectral decomposition, which states that every square, symmetric matrix $A \in \mathbb{R}^{m \times m}$ can be written as

$$A = \Gamma \Lambda \Gamma' = \sum_{j=1}^{m} \lambda_j \gamma_j \gamma_j', \tag{17}$$

where $\Lambda = diag(\lambda_1, \ldots, \lambda_m)$ is a diagonal matrix with the eigenvalues of A on its main diagonal and where $\Gamma = (\gamma_1, \ldots, \gamma_m)$ is the matrix containing the associated eigenvectors (Härdle and Simar, 2015). This means we can draw a random vector of eigenvalues λ_{true} and a random orthogonal matrix to construct a test matrix A_{test} (Anderson et al., 1999). Once the matrix A_{test} is initialized, we can plug it into the eigenvalue-routine we wish to test.

Table 1: Failed unit tests accross matrix-sizes

Algorithm	Maximum number	Dimension				
	of iterations	3	4	5	6	7
Jacobi	-	0	0	0	0	0
QRM1	10	785	952	994	999	1000
	100	110	224	349	474	565
	1000	7	16	33	38	45
	10000	1	1	1	1	3
	100000	0	0	0	0	0
QRM2	10	826	975	997	1000	1000
	100	111	191	360	467	602
	1000	9	16	22	35	58
	10000	0	0	3	2	0
	100000	0	0	0	0	0
QRM3	10	251	616	896	983	997
	100	29	52	116	222	311
	1000	6	2	8	19	21
	10000	0	0	1	1	3
	100000	0	0	0	0	0

4.2 Efficiency 4 ANALYSIS

The number of failures out of 1000 unit tests on matrices of different dimensions are catalogued in Table 1. Most notably, we see that the Jacobi-method outperforms any of the QR-methods in terms of accuracy. This is mostly due to its convergence criterion being defined differently than the convergence criterion of the QR-methods. Simply put, the Jacobi-method will always converge in this setup. Also keep in mind, that since the Jacobi-method is only allowed for symmetric matrices, it essentially eliminates two nonzero off-diagonal elements at once. Comparing only the QR-methods amongst each other one can clearly see, that the quality of the obtained results increases, the more willing an analyst is to wait. Convergence is slower for the QR-methods and a premature stop leads to poor eigenvalues. One the other hand, if the QR-methods converge, they unanimously pass all tests. It would therefore be possible to always achieve an arbitrary grade of precision by allowing for a higher number of iterations. For the highest possible accuracy one can set the maximum number of iterations to infinity.

4.2 Efficiency

Now that we can be confident that each of the routines is able to offer accurate results we are interested in which method is most appropriate for which kind of matrix. Especially, we are interested in convergence speed. Figure 5 shows the distribution of steps performed before the algorithms reached convergence or their maximum allowed number

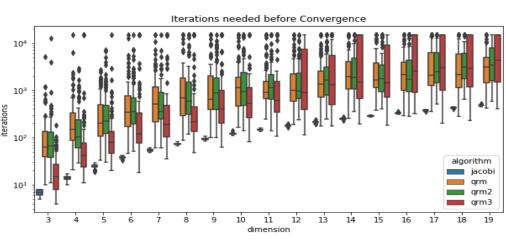


Figure 6: Decision process of the final eigenvalue routine \square Input: matrix A

Is A symmetric?

yes

no

rows of A \leq 11?

yes

QRM3

QRM2

of iterations. Again, the Jacobi-method outperforms every QR-method. For matrices of small dimensions, i.e. eleven or less, the accelerated QR-method outperforms the other QR-methods. However, for larger matrices the Hessenberg variant seems to be slightly more appropriate, since the accelerated method tends to have a wider range of needed steps. Additionally, the accelerated method's distribution has a wider tail whenever the dimension of the test matrices is larger than eleven. This means that high waiting times, operating on the same matrix, are more likely to occur using the accelerated variant compared to the *Hessenberg* variant or even the basic variant. This can only mean that the supplementary additions and subtractions of the accelerated method provide a questionable aid. In some cases they do seem to help for small matrices, however when confronted with larger matrices the cost of the additions and subtractions outweighs the benefit of the facilitated QR-step. Surprisingly, there seems to be no striking difference between the distributions of the basic and the *Hessenberg* variants. Looking closely, the only effect of performing a Hessenberg transformation prior to running a QR-algorithm on a test matrix is to mitigate outliers in terms of computation steps needed before convergence. And even that is true, only for large matrices. This can only mean that the cost of computing the Hessenberg-form of the matrix is roughly as high as the reduction of steps needed before convergence is reached.

5 Conclusion

Finally, we would like to present a compound solution which combines the strengths of all presented algorithms and automatically picks the procedure that fits best to the problem. For obvious reasons, it is best to chose the Jacobi-method, whenever applicable. That is, when the input-matrix is symmetric. Whenever the matrix is not symmetric, a second decision based on the matrix' size will determine which QR-method will be used. If the matrix' dimension is smaller than eleven it will chose the accelerated QR method. Otherwise, the Hessenberg variant of the QR-method will be deployed.

We have thus demonstrated the practical importance of eigenvalues in the context of Principal Component Analysis and Fisher's Linear Discriminant Analysis. For computing the eigenvalues we have shown various numerical procedures. We examinated the Jacobi-and the QR-method, as well as a selection of simple improvements towards the basic QR-method. We have shown that the implemented routines can achieve an arbitrary grade of precision and provided a guided procedure for when to use which of the provided algorithms.

6 Appendix

6.1 Householder-Reflections

Our goal is to diagonalize a matrix in order to programmatically extract its eigenvalues. In section 2 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 a particular 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

$$P = I - 2uu'. (18)$$

Note that

$$Px = c_1 u + c_2 v - 2c_1 u u u' - 2c_2 v u u'$$

$$= c_1 u + c_2 v - 2c_1 u' u u - 2c_2 u' v u$$

$$= -c_1 u + c_2 v$$

$$= \hat{x}.$$

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

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

into a vector

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

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

$$v = (x_1 + sign(x_1)||x||_2, x_2, \dots, x_n)$$
(19)

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 $A \in \mathbb{R}^{n \times n}$.

$$A = QR \tag{20}$$

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 factorization in this fashion is $2n^3/3$ multiplications and $2n^3/3$ additions (Gentle, 1998, p. 110).

6.2 Givens-Rotations 6 APPENDIX

6.2 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 7: Rotation of $x \supseteq x_2$ (4,3) (5,0)

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 $matrixV_{pq}$, that will transform the vector

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

to

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

. The matrix that does this is

$$V_{pq}(\theta) = \begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & \\ & & \cos \theta & & \sin \theta & & \\ & & & \ddots & & \\ & & -\sin \theta & & \cos \theta & & \\ & & & \ddots & & \\ & & & & 1 \end{bmatrix}$$
 (21)

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

A rotation matrix is therefore the same as an identity matrix, in which we change four elements (Börm and Mehl, 2012, p. 105). We will use Givens rotations primarily in the Jacobi-Method.

6.3 Code

6.3.1 Eigenvalue Routines

```
import numpy as np
2 import copy
3
  def hreflect1D(x):
5
       n n n
6
       Calculate Householder reflection: Q = I - 2*uu'.
7
8
       Parameters:
           X: numpy array.
10
11
       Returns:
12
           Qx: reflected vector.
13
           Q: Reflector (matrix).
14
       n n n
15
       # Construct v:
16
      v = copy.deepcopy(x)
17
      v[0] += np.linalg.norm(x)
18
19
       # Construct u: normalize v.
20
      vnorm = np.linalg.norm(v)
21
       if vnorm:
22
           u = v / np.linalg.norm(v)
23
       else:
24
           u = v
25
26
       # Construct Q:
27
       Q = np.eye(len(x)) - 2 * np.outer(u, u)
       Qx = np.dot(Q, x)
29
       return Qx, Q
31
32
```

```
33
  def qr_factorize(X, offset=0):
34
       11 11 11
35
       Compute QR factorization of X s.t. QR = X.
36
37
      Parameters:
38
           - X: square numpy ndarray.
39
           - offset: (int) either 0 or 1. If offset is unity: compute
40
             Hessenberg -
                      matrix.
41
42
      Returns:
43
           Q: square numpy ndarray, same shape as X. Rotation matrix.
44
           R: square numpy ndarray, same shape as X. Upper triangular
45
             matrix if
              offset is 0, Hessenberg-matrix if offset is 1.
46
       n n n
47
      assert offset in [0, 1]
48
      assert type(X) == np.ndarray
49
      assert X.shape[0] == X.shape[1]
50
51
      R = copy.deepcopy(X)
52
      Q = np.eye(X.shape[0])
53
54
      for i in range(X.shape[0]-offset):
55
           Pi = np.eye(R.shape[0])
           _, Qi = hreflect1D(R[i+offset:, i])
57
           Pi[i+offset:, i+offset:] = Qi
58
59
           Q = Pi.dot(Q)
60
           R = Pi.dot(R)
61
62
      return Q.T, R
63
```

```
1 """
2 Algorithms for solving eigenvalue problems.
```

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

```
Compute Eigenvalues and Eigenvectors for symmetric matrices.
38
39
      Parameters:
40
          X - 2D numpy ndarray which represents a symmetric matrix
41
          precision - float in (0, 1). Convergence criterion.
42
43
      Returns:
          A - 1D numpy array with eigenvalues sorted by absolute value
45
          U - 2D numpy array with associated eigenvectors (column).
      n n n
      assert 0 < precision < 1.
48
      assert type(X) == np.ndarray
49
      n, m = X.shape
50
      assert n == m
51
      assert all(np.isclose(X - X.T, np.zeros(n)).flatten())
52
      A = copy.deepcopy(X)
53
      U = np.eye(A.shape[0])
54
      L = np.array([1])
55
      iterations = 0
56
57
      while L.max() > precision:
58
          L = np.abs(np.tril(A, k=0) - np.diag(A.diagonal()))
59
          i, j = np.unravel_index(L.argmax(), L.shape)
60
          alpha = 0.5 * np.arctan(2*A[i, j] / (A[i, i]-A[j, j]))
61
62
          V = np.eye(A.shape[0])
          V[i, i], V[j, j] = np.cos(alpha), np.cos(alpha)
64
          V[i, j], V[j, i] = -np.sin(alpha), np.sin(alpha)
65
66
          A = np.dot(V.T, A.dot(V))
67
          U = U.dot(V)
68
          iterations += 1
70
      # Sort by eigenvalue (descending order) and flatten A
71
      A = np.diag(A)
72
```

```
order = np.abs(A).argsort()[::-1]
73
       if debug:
74
           return iterations
75
76
       return A[order], U[:, order]
77
78
79
  def qrm(X, maxiter=15000, debug=False):
80
81
       Compute Eigenvalues and Eigenvectors using the QR-Method.
82
83
       Parameters:
84
           - X: square numpy ndarray.
85
       Returns:
86
           - Eigenvalues of A.
87
           - Eigenvectors of A.
88
       n n n
89
       n, m = X.shape
90
       assert n == m
91
92
       # First stage: transform to upper Hessenberg-matrix.
93
       A = copy.deepcopy(X)
94
       conv = False
95
       k = 0
96
97
       \# Second stage: perform QR-transformations.
       while (not conv) and (k < maxiter):
99
           k += 1
100
           Q, R = helpers.qr_factorize(A)
101
           A = R.dot(Q)
102
103
           conv = np.alltrue(np.isclose(np.tril(A, k=-1), np.zeros((n, n)
             )))
105
       if not conv:
106
```

```
warnings.warn("Convergence was not reached. Consider raising
107
             maxiter.")
       if debug:
108
           return k
109
       Evals = A.diagonal()
110
       order = np.abs(Evals).argsort()[::-1]
111
       return Evals[order], Q[order, :]
112
113
114
  def qrm2(X, maxiter=15000, debug=False):
116
       First compute similar matrix in Hessenberg form, then compute the
117
       Eigenvalues and Eigenvectors using the QR-Method.
118
119
       Parameters:
120
            - X: square numpy ndarray.
121
       Returns:
122
            - Eigenvalues of A.
123
            - Eigenvectors of A.
124
       11 11 11
125
       n, m = X.shape
126
       assert n == m
127
128
       # First stage: transform to upper Hessenberg-matrix.
129
       A = lin.hessenberg(X)
130
       conv = False
131
       k = 0
132
133
       \# Second stage: perform QR-transformations.
       while (not conv) and (k < maxiter):
135
           k += 1
136
            Q, R = helpers.qr_factorize(A)
137
            A = R.dot(Q)
138
139
            conv = np.alltrue(np.isclose(np.tril(A, k=-1), np.zeros((n, n)
140
```

```
)))
141
       if not conv:
142
            warnings.warn("Convergence was not reached. Consider raising
143
             maxiter.")
       if debug:
144
           return k
145
       Evals = A.diagonal()
146
       order = np.abs(Evals).argsort()[::-1]
147
       return Evals[order], Q[order, :]
148
149
150
  def qrm3(X, maxiter=15000, debug=False):
151
152
       First compute similar matrix in Hessenberg form, then compute the
153
       Eigenvalues and Eigenvectors using the QR-Method.
154
155
       Parameters:
156
            - X: square numpy ndarray.
157
       Returns:
158
           - Eigenvalues of A.
159
           - Eigenvectors of A.
160
       n n n
161
       n, m = X.shape
162
       assert n == m
163
164
       # First stage: transform to upper Hessenberg-matrix.
165
       T = lin.hessenberg(X)
166
167
       conv = False
168
       k = 0
169
       # Second stage: perform QR-transformations.
171
       while (not conv) and (k < maxiter):
172
           k += 1
173
```

```
Q, R = helpers.qr_factorize(T - T[n-1, n-1] * np.eye(n))
174
            T = R.dot(Q) + T[n-1, n-1] * np.eye(n)
175
176
            conv = np.alltrue(np.isclose(np.tril(T, k=-1), np.zeros((n, n)
177
             )))
178
       if not conv:
179
            warnings.warn("Convergence was not reached. Consider raising
180
             maxiter.")
       if debug:
181
            return k
       Evals = T.diagonal()
183
       order = np.abs(Evals).argsort()[::-1]
184
       return Evals[order], Q[order, :]
185
186
187
   def eigen(X):
188
       n n n
189
       Compute eigenvalues and eigenvectors of X.
190
191
       Parameters:
192
            - X: square numpy ndarray.
193
       Returns:
194
            - Eigenvalues of A.
195
            - Eigenvectors of A.
196
197
        11 11 11
198
199
       symmetric = np.alltrue(np.isclose(X - X.T, np.zeros(n)))
200
       small = max(X.shape) <= 11
201
202
       if symmetric:
203
            return jacobi(X)
204
       elif small:
205
            maxiter = 10 ** max(*X.shape, 4)
206
```

```
return qrm3(X, maxiter=maxiter)

else:
maxiter = 10 ** max(*X.shape, 4)

return qrm2(X, maxiter=maxiter)
```

6.3.2 Analysis: Figures

```
import os
  import copy
  import pandas as pd
  import numpy as np
 import seaborn as sns
 from scipy import linalg as lin
 from scipy.stats import ortho_group
  from matplotlib import pyplot as plt
  datadir = os.path.join("analysis", "benchmarks.csv")
  outpath = os.path.join("media", "plots")
  trials = pd.read_csv(datadir, index_col=0)
  trials.groupby(["algorithm", "dimension"]).iterations.describe()
15
  # Boxplot iteration:
  fig = plt.figure(figsize=(10, 5))
  sns.boxplot(x="dimension", y="iterations", hue="algorithm", data=
   trials)
  plt.yscale("log")
  plt.title("Iterations needed before Convergence")
 plt.savefig(os.path.join(outpath, "iterations_boxplot.png"))
  plt.show()
  plt.close()
23
24
  # 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)")
29
 plt.yscale('log')
  plt.savefig(os.path.join(outpath, "time_boxplot.png"))
  plt.show()
  plt.close()
```

```
34
  # Visualize Algorithm-Progress:
35
  np.random.seed(42)
36
  size = 5
37
  Lambda = np.diag(np.random.randint(low=0, high=10, size=size))
  G = ortho_group.rvs(dim=size)
39
  X = np.dot(G, Lambda.dot(G.T))
42
  def plot_factory(func):
      def plotter(savepath, **fig_kw):
           def algorithm_generator(*args, **kwargs):
45
               return func(*args, **kwargs)
46
47
           fig, ax = plt.subplots(nrows=2, ncols=2, **fig_kw)
48
           algorithm_iterator = algorithm_generator()
49
           i = -1
50
51
           for i, A in enumerate(algorithm_iterator):
52
               if i in (0, 1, 10, 75):
53
                   j += 1
54
55
                   hm = ax[j // 2, j \% 2].imshow(A,
56
                                                    cmap=plt.get_cmap('
57
                                                      seismic'),
                                                    vmin=-X.max(),
58
                                                    vmax=X.max())
59
                   ax[j // 2, j % 2].set_yticks([])
                   ax[j // 2, j % 2].set_xticks([])
61
                   ax[j // 2, j % 2].set_title("Iteration: " + str(i))
62
63
                   if i > 75:
                        break
65
66
           fig.subplots_adjust(right=0.8)
67
```

```
cbar_ax = fig.add_axes([0.85, 0.15, 0.05, 0.7])
68
           fig.colorbar(hm, cax=cbar_ax)
69
70
           sup_title = "Demonstrate {} on a {}x{} matrix".format(
71
                func.__name__,
72
                *X.shape)
73
74
           fig.suptitle(sup_title)
75
           fig.savefig(savepath)
           return fig, ax
78
79
       return plotter
80
81
82
  @plot_factory
83
  def jacobi():
84
       n n n
85
       Compute Eigenvalues and Eigenvectors for symmetric matrices using
86
        the
       jacobi method.
87
88
       Yields:
89
           * A - 2D numpy array of current iteration step.
90
       H H H
91
       A = copy.deepcopy(X)
92
       U = np.eye(A.shape[0])
93
       L = np.array([1])
       iterations = 0
96
       while iterations < 5000:
97
           L = np.abs(np.tril(A, k=0) - np.diag(A.diagonal()))
           i, j = np.unravel_index(L.argmax(), L.shape)
99
           alpha = 0.5 * np.arctan(2*A[i, j] / (A[i, i]-A[j, j]))
100
101
```

```
V = np.eye(A.shape[0])
102
            V[i, i], V[j, j] = np.cos(alpha), np.cos(alpha)
103
            V[i, j], V[j, i] = -np.sin(alpha), np.sin(alpha)
104
105
            A = np.dot(V.T, A.dot(V))
106
            U = U.dot(V)
107
            iterations += 1
108
            yield A
109
110
111
   @plot_factory
   def qrm1():
113
        n n n
114
       Create generator for transformed matrices after applying the QR-
115
         Method.
116
        Yields:
117
            - T: 2D-numpy array. Similar matrix to X.
118
119
       \hbox{\it\# First stage: transform to upper Hessenberg-matrix.}
120
       T = copy.deepcopy(X)
121
122
       k = 0
123
       \# Second stage: perform QR-transformations.
124
       while k < 5000:
125
            k += 1
126
            Q, R = np.linalg.qr(T)
127
            T = R.dot(Q)
128
            yield T
129
130
131
   @plot_factory
   def qrm2():
133
        n n n
134
```

```
Create generator for transformed matrices after applying the QR-
135
         Method.
136
       Yields:
137
           - T: 2D-numpy array. Similar matrix to X.
138
       H H H
139
       # First stage: transform to upper Hessenberg-matrix.
140
       T = lin.hessenberg(X)
141
142
       k = 0
143
       # Second stage: perform QR-transformations.
144
       while k < 5000:
145
           if k == 0:
146
                yield X
147
           k += 1
148
           Q, R = np.linalg.qr(T)
149
           T = R.dot(Q)
150
           yield T
151
152
153
  @plot_factory
154
  def qrm3():
155
       n n n
156
       First compute similar matrix in Hessenberg form, then compute the
157
       Eigenvalues and Eigenvectors using the accelerated QR-Method.
158
159
       Yields:
160
            * T - 2D numpy array of current iteration step.
161
       H/H/H
162
       # First stage: transform to upper Hessenberg-matrix.
163
       T = lin.hessenberg(X)
164
       k = 0
165
       n, _= X.shape
166
167
       # Second stage: perform QR-transformations.
168
```

```
while k < 5000:
169
           if k == 0:
170
               yield X
171
           k += 1
172
           Q, R = np.linalg.qr(T - T[n-1, n-1] * np.eye(n))
173
           T = R.dot(Q) + T[n-1, n-1] * np.eye(n)
174
175
           yield T
176
177
  jacobi(os.path.join(outpath, "jacobi.png"))
  qrm1(os.path.join(outpath, "qrm1.png"))
180
  qrm2(os.path.join(outpath, "qrm2.png"))
181
  qrm3(os.path.join(outpath, "qrm3.png"))
183
plt.show()
plt.close()
```

6.3.3 Analysis: Unit tests

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

```
self.maxiter = kwargs.get("maxiter", None)
34
35
           if not os.path.exists(self.path):
36
               self.save(header=["algorithm", "dimension", "maxiter", "
37
                failed"])
38
      def __get_test_algorithm(self, algorithm, *args, **kwargs):
39
           @wraps(algorithm)
40
           def algo(X):
41
               return algorithm(X, *args, **kwargs)
           return algo
44
      def __get_test_matrix(self):
45
           """Return matrix with assosiated Eigenvalues."""
46
           eigenvalues = np.random.uniform(size=self.dim)
47
           eigenvectors = ortho_group.rvs(dim=self.dim)
48
           Lambda = np.diag(eigenvalues)
49
50
           matrix = np.dot(eigenvectors, Lambda).dot(eigenvectors.T)
51
52
           order = np.abs(eigenvalues).argsort()[::-1]
53
           return matrix, eigenvalues[order]
54
55
      def __run_test(self):
56
           """Run singular test."""
57
           mat, true_eig = self.__get_test_matrix()
           test_eig, _ = self.algorithm(mat)
59
           test_res = np.alltrue(np.isclose(true_eig, test_eig))
60
           self.failed.append(not test_res)
61
62
      def __run_tests(self):
63
           """Run multiple tests."""
           for _ in range(self.n):
65
               try:
66
                   self.__run_test()
67
```

```
except (KeyboardInterrupt, SystemExit):
68
                    self.__save()
69
                    sys.exit(0)
70
71
       def run(self):
72
           """Distribute tests accross threads."""
73
           threadlist = [None] * self.jobs
75
           for i in range(self.jobs):
               threadlist[i] = Thread(target=self.__run_tests, daemon=
                 True)
               threadlist[i].start()
79
80
           for thread in threadlist:
81
               thread.join()
82
83
           self.result = sum(self.failed)
84
           self.tests["algorithm"].append(self.algorithm.__name__)
85
           self.tests["dimension"].append(self.dim)
86
           self.tests["failed"].append(self.result)
87
           if self.algorithm.__name__ == "jacobi":
88
               self.tests["maxiter"].append(None)
89
           else:
90
               self.tests["maxiter"].append(self.maxiter)
91
      def save(self, header=False):
93
           if os.path.exists(self.path):
                   print("Saving results.")
           df = pd.DataFrame(self.tests)
           with open(self.path, 'a') as f:
97
               df.to_csv(f, header=header, index=False,
                          columns=["algorithm", "dimension", "maxiter", "
99
                            failed"])
100
```

```
101
  # Unit tests
102
   if __name__ == "__main__":
103
       # Define Flags.
104
       JOBS = 20
105
       MAXITER = (10, 100, 1000, 10000, 100000)
106
       DIMS = range(3, 8)
107
       ALGOS = {'jacobi': eigen.jacobi,
108
                 'qrm': eigen.qrm,
                 'qrm2': eigen.qrm2,
110
                 'qrm3': eigen.qrm3}
112
       # Define parameters of all runs.
113
       parameters = []
114
       for algo in ALGOS.values():
115
            for maxiter in MAXITER:
116
                for dim in range(3, 8):
117
                     param = (algo,
118
                               maxiter if algo.__name__ != "jacobi" else
119
                                None,
                               dim)
120
                    parameters.append(param)
121
122
       # Check progress of previous runs.
123
       if os.path.exists(data_out):
124
            print("Check progress:")
125
            required = [(algo.__name__, m, dim) for (algo, m, dim) in
126
             parameters]
            required = set(required)
127
128
           progress = pd.read_csv(data_out)
129
            done = []
131
            for (a, d, m, _) in progress.values:
132
                if np.isnan(m):
133
```

```
param = (a, None, d)
134
                else:
135
                     param = (a, int(m), d)
136
                done.append(param)
137
            done = set(done)
138
139
            to_do = required.difference(done)
140
            parameters = [(ALGOS[a], m, d) for a, m, d in to_do]
141
142
       for algo, maxiter, dim in tqdm(parameters):
143
            if algo is eigen.jacobi:
                algo_test = AlgoTest(filepath=data_out,
145
                                        algo=algo,
146
                                        dim=dim,
147
                                        jobs=JOBS)
148
            else:
149
                algo_test = AlgoTest(filepath=data_out,
150
                                        algo=algo,
151
                                        dim=dim,
152
                                        maxiter=maxiter,
153
                                        jobs=JOBS)
154
            algo_test.run()
155
156
       algo_test.save()
157
158
       results = pd.read_csv(data_out)
159
       results.head(10)
160
       results.loc[results.algorithm == 'jacobi', 'maxiter'] = '-'
161
162
       crosstab = pd.crosstab(index=[results.algorithm, results.maxiter],
163
                                  columns=results.dimension,
164
                                  values=results.failed,
165
                                  aggfunc=np.mean,
166
                                  dropna=True)
167
       print(crosstab.to_latex())
168
```

7 References

- Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. (1999). LAPACK Users' Guide. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition.
- Börm, S. and Mehl, C. (2012). Numerical Methods for Eigenvalue Problems. Walter de Gruyter GmbH & Co.KG, Berlin/Boston.
- Duda, R. O., Hart, P. E., and Stork, D. G. (2000). *Pattern Classification (2Nd Edition)*. Wiley-Interscience, New York, NY, USA.
- Gentle, J. E. (1998). Numerical Linear Algebra for Applications in Statistics. Springer Science and Business Media, New York.
- Härdle, W. K. and Simar, L. (2015). Applied Multivariate Statistical Analysis. Springer-Verlag Gmbh, Berlin, Heidelberg.
- Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. Computing In Science & Engineering, 9(3):90–95.
- 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, G. (1995). Python tutorial. Technical Report CS-R9526, Centrum voor Wiskunde en Informatica (CWI), Amsterdam.