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 30, 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,

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

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.

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 $P = \prod\limits_{k=1}^{K} V_{p_k,q_k}(\theta_k)$ and $V_{p_k,q_k}(\theta_k)$ is of the form proposed in (17). More specific the *Jacobi iteration* is

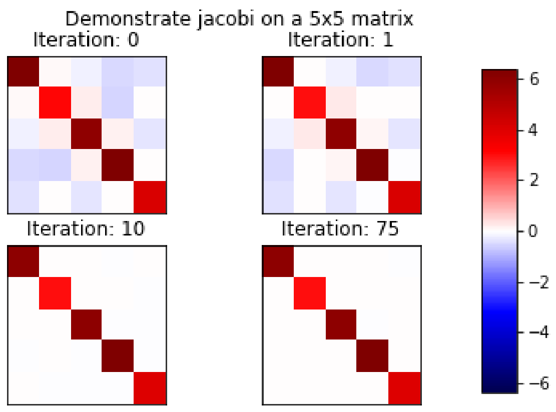
$$A^{(k)} = V'_{p_k,q_k}(\theta_k)A^{(k-1)}V_{p_k,q_k}(\theta_k), \tag{6}$$

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 (17) 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

$$a_{pq}^{(k-1)}(\cos^2\theta - \sin^2\theta) + \left(a_{pp}^{(k-1)} - a_{qq}^{(k-1)}\right)\cos\theta\sin\theta = 0. \tag{7}$$

Figure 1: Progress Jacobi-Method



We can use the trigonometric identities

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

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

in (7) 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].

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, 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
2:     Find indices $i$, $j$ of largest value in lower triangle of $abs(L)$
3:     $L_{max} \leftarrow L_{i,j}$
4:     $\alpha \leftarrow \frac{1}{2} \cdot \arctan(\frac{2A_{i,j}}{A_{i,i}-A_{j,j}})$
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 $diag(A)$, $U$
```

3.2 QR-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.

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*.

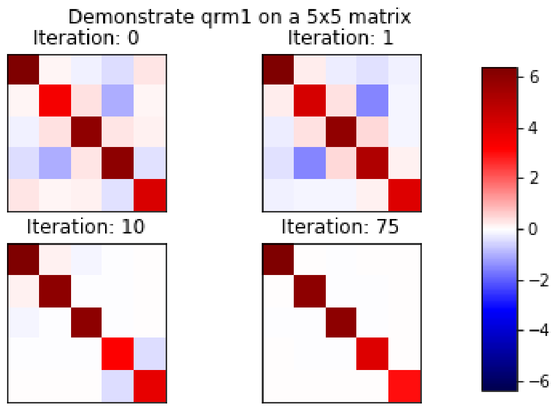
$$A = QR, \tag{8}$$

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

$$A^k = Q'_{k-1}A_{k-1}Q_{k-1} = R_{k-1}Q_{k-1} \tag{9}$$

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. Additionally, in an implementation it is usually preferable to compute the *QR*-iteration in the way shown at the rightmost part of equation (9) [Börm and Mehl, 2012]. Although, mathematically, each statement is exactly identical there is a practical difference in due to computational imperfections and limited machine precision.

Figure 2: Progress basic QR-Method



The reason being 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. Additionally, 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]. 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`

```
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 $a_{ij} = 0\ \forall i > j + 1$:

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

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].

Figure 3: Progress Hessenberg-QR-Method

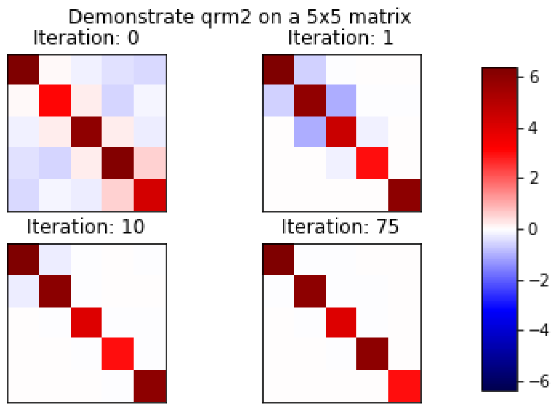


Figure 3 visualizes the progress of the *Hessenberg* variant of the *QR*-method. In order to make it comparable to the previous algorithms the same 5×5-dimensional matrix is being evaluated. We can readily see, that the transformation to *Hessenberg*-form results in a tridiagonal matrix. This facilitates 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`

```
Require: square matrix $A$
1: $A \leftarrow$ 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 the final adjustment on the *QR*-method 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

$$T^m = \begin{bmatrix} \alpha_1^m & \beta_1^m & 0 & 0 & \dots & 0 & \\ \beta_1^m & \alpha_2^m & \beta_2^m & & & & \\ 0 & \beta_2^m & \alpha_3^m & \beta_3^m & & \vdots & \\ & & \ddots & \ddots & \ddots & & \\ & & & & \beta_{n-2}^m & \alpha_{n-1}^m & \beta_{n-1}^m \\ 0 & & & & & \beta_{n-1}^m & \alpha_n^m \end{bmatrix} \tag{10}$$

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

$$T^m = QR$$

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

After this we can define the accelerated iteration step as

$$R_m = Q'_m \left(T_m - \alpha_n^m I\right) \tag{11}$$

$$T_{m+1} = Q'_m \left(T_m - \alpha_n^m I\right) Q_m + \alpha_n^m I \tag{12}$$

$$= Q'_m T_m Q_m \tag{13}$$

Again T_{m+1} is similar to T_m.

Figure 4: Progress Accelerated QR-Method

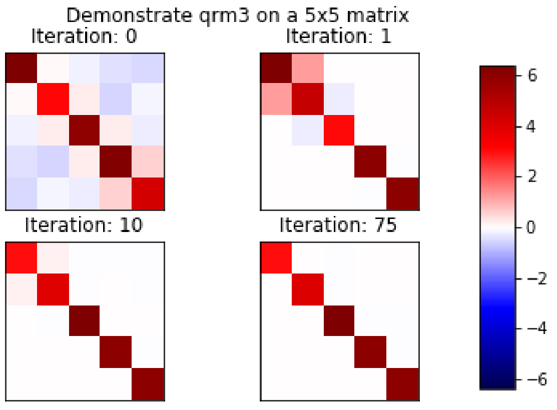


Table 1: Unit tests accross matrix-sizes

awesome	sauce
nothing	to
see	here

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 than the. Howbeit, this claim warrants further analysis.

Algorithm 4 `QRM3`

```
Require: square matrix $A \in \mathbb{R}^{p\times p}$
1: $T \leftarrow \texttt{hessenberg}(A),\ conv \leftarrow False$
2: while not $conv$ do
3:     $Q, R \leftarrow$ 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

4.1 Accuracy

4.2 Efficiency

5 Conclusion

Figure 5: Unit-tests: Iterations

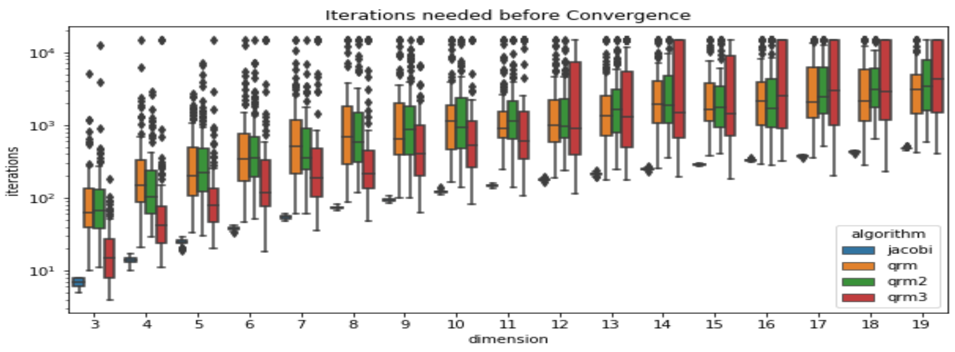
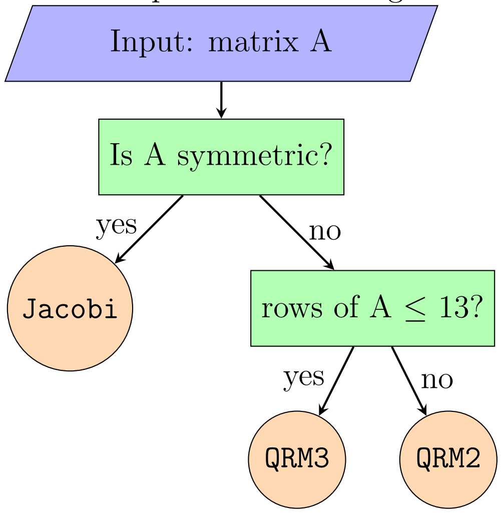


Figure 6: Decision process of final eigenvalue routine



6 Appendix

6.1 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

$$P = I - 2uu'. \tag{14}$$

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

$$v = (x_1 + sign(x_1)||x||_2, x2, \ldots, x_n) \tag{15}$$

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

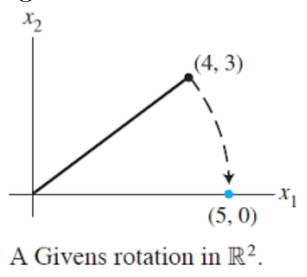
$$A = QR \tag{16}$$

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 $Q = P_5P_4P_3P_2P_1$. The number of computations for the QR factorization in this fashion is $2n^3/3$ multiplications and $2n^3/3$ additions [Gentle, 1998].

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



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

$$V_{pq}(\theta) = \begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & \cos\theta & & \sin\theta & & \\ & & & \ddots & & & \\ & & -\sin\theta & & \cos\theta & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{bmatrix} \tag{17}$$

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.

6.3 Eigenvalue Routines

```python
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).
    """
    # Construct v:
    v = copy.deepcopy(x)
    v[0] += np.linalg.norm(x)

    # Construct u: normalize v.
    vnorm = np.linalg.norm(v)
    if vnorm:
        u = v / np.linalg.norm(v)
    else:
        u = v

    # Construct Q:
    Q = np.eye(len(x)) - 2 * np.outer(u, u)
    Qx = np.dot(Q, x)

    return Qx, Q


def qr_factorize(X, offset=0):
    """
    Compute QR factorization of X s.t. QR = X.

    Parameters:
        - X: square numpy ndarray.
        - offset: (int) either 0 or 1. If offset is unity: compute Hessenberg-
                  matrix.

    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
```

```python
"""
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
import warnings
from scipy import linalg as lin
from algorithms import helpers


def jacobi2x2(A):
    """
    Diagonalize a 2x2 matrix through jacobi step.

    Solve: U' A U = E s.t. E is a diagonal matrix.

    Parameters:
        A - 2x2 numpy array.
    Returns:
        A - 2x2 diagonal numpy array
    """
    assert type(A) == np.ndarray
    assert A.shape == (2, 2)
    assert A[1, 0] == A[0, 1]

    alpha = 0.5 * np.arctan(2*A[0, 1]/(A[1, 1] - A[0, 0]))
    U = np.array([[np.cos(alpha), np.sin(alpha)],
                  [-np.sin(alpha), np.cos(alpha)]])
    E = np.matmul(U.T, np.matmul(A, U))
    return E


def jacobi(X, precision=1e-6, debug=False):
    """
    Compute Eigenvalues and Eigenvectors for symmetric matrices.

    Parameters:
        X - 2D numpy ndarray which represents a symmetric matrix
        precision - float in (0, 1). Convergence criterion.

    Returns:
        A - 1D numpy array with eigenvalues sorted by absolute
          value
        U - 2D numpy array with associated eigenvectors (column).
    """
    assert 0 < precision < 1.
    assert type(X) == np.ndarray
    n, m = X.shape
    assert n == m
    assert all(np.isclose(X - X.T, np.zeros(n)).flatten())
    A = copy.deepcopy(X)
    U = np.eye(A.shape[0])
    L = np.array([1])
    iterations = 0
```

```python

    while L.max() > precision:
        L = np.abs(np.tril(A, k=0) - np.diag(A.diagonal()))
        i, j = np.unravel_index(L.argmax(), L.shape)
        alpha = 0.5 * np.arctan(2*A[i, j] / (A[i, i]-A[j, j]))

        V = np.eye(A.shape[0])
        V[i, i], V[j, j] = np.cos(alpha), np.cos(alpha)
        V[i, j], V[j, i] = -np.sin(alpha), np.sin(alpha)

        A = np.dot(V.T, A.dot(V))
        U = U.dot(V)
        iterations += 1

    # Sort by eigenvalue (descending order) and flatten A
    A = np.diag(A)
    order = np.abs(A).argsort()[::-1]
    if debug:
        return iterations

    return A[order], U[:, order]


def qrm(X, maxiter=15000, debug=False):
    """
    Compute Eigenvalues and Eigenvectors using the QR-Method.

    Parameters:
        - X: square numpy ndarray.
    Returns:
        - Eigenvalues of A.
        - Eigenvectors of A.
    """
    n, m = X.shape
    assert n == m

    # First stage: transform to upper Hessenberg-matrix.
    A = copy.deepcopy(X)
    conv = False
    k = 0

    # Second stage: perform QR-transformations.
    while (not conv) and (k < maxiter):
        k += 1
        Q, R = helpers.qr_factorize(A)
        A = R.dot(Q)

        conv = np.alltrue(np.isclose(np.tril(A, k=-1), np.zeros((n
          , n))))

    if not conv:
        warnings.warn("Convergence was not reached. Consider
          raising maxiter.")
    if debug:
        return k
    Evals = A.diagonal()
    order = np.abs(Evals).argsort()[::-1]
    return Evals[order], Q[order, :]


def qrm2(X, maxiter=15000, debug=False):
    """
    First compute similar matrix in Hessenberg form, then compute
      the
    Eigenvalues and Eigenvectors using the QR-Method.

    Parameters:
        - X: square numpy ndarray.
    Returns:
        - Eigenvalues of A.
```

```
          - Eigenvectors of A.
    """
    n, m = X.shape
    assert n == m

    # First stage: transform to upper Hessenberg-matrix.
    A = lin.hessenberg(X)
    conv = False
    k = 0

    # Second stage: perform QR-transformations.
    while (not conv) and (k < maxiter):
        k += 1
        Q, R = helpers.qr_factorize(A)
        A = R.dot(Q)

        conv = np.alltrue(np.isclose(np.tril(A, k=-1), np.zeros((n
          , n))))

    if not conv:
        warnings.warn("Convergence was not reached. Consider
          raising maxiter.")
    if debug:
        return k
    Evals = A.diagonal()
    order = np.abs(Evals).argsort()[::-1]
    return Evals[order], Q[order, :]


def qrm3(X, maxiter=15000, debug=False):
    """
    First compute similar matrix in Hessenberg form, then compute
      the
    Eigenvalues and Eigenvectors using the QR-Method.

    Parameters:
        - X: square numpy ndarray.
    Returns:
        - Eigenvalues of A.
        - Eigenvectors of A.
    """
    n, m = X.shape
    assert n == m

    # First stage: transform to upper Hessenberg-matrix.
    T = lin.hessenberg(X)

    conv = False
    k = 0

    # Second stage: perform QR-transformations.
    while (not conv) and (k < maxiter):
        k += 1
        Q, R = helpers.qr_factorize(T - T[n-1, n-1] * np.eye(n))
        T = R.dot(Q) + T[n-1, n-1] * np.eye(n)

        conv = np.alltrue(np.isclose(np.tril(T, k=-1), np.zeros((n
          , n))))

    if not conv:
        warnings.warn("Convergence was not reached. Consider
          raising maxiter.")
    if debug:
        return k
    Evals = T.diagonal()
    order = np.abs(Evals).argsort()[::-1]
    return Evals[order], Q[order, :]
```

6.4 Analysis: Figures

```python
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()

# 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()

# 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()
        j = -1

        for i, A in enumerate(algorithm_iterator):
            if i in (0, 1, 10, 75):
                j += 1

                hm = ax[j // 2, j % 2].imshow(A,
                                              cmap=plt.get_cmap('
                                                seismic'),
                                              vmin=-X.max(),
                                              vmax=X.max())
                ax[j // 2, j % 2].set_yticks([])
                ax[j // 2, j % 2].set_xticks([])
                ax[j // 2, j % 2].set_title("Iteration: " + str(i)
                  )

                if i > 75:
                    break

        fig.subplots_adjust(right=0.8)
        cbar_ax = fig.add_axes([0.85, 0.15, 0.05, 0.7])
        fig.colorbar(hm, cax=cbar_ax)

        sup_title = "Demonstrate {} on a {}x{} matrix".format(
            func.__name__,
            *X.shape)

        fig.suptitle(sup_title)
        fig.savefig(savepath)

        return fig, ax

    return plotter


@plot_factory
def jacobi():
    """
    Compute Eigenvalues and Eigenvectors for symmetric matrices
      using the
    jacobi method.

    Yields:
        * A - 2D numpy array of current iteration step.
    """
    A = copy.deepcopy(X)
    U = np.eye(A.shape[0])
    L = np.array([1])
    iterations = 0

    while iterations < 5000:
        L = np.abs(np.tril(A, k=0) - np.diag(A.diagonal()))
        i, j = np.unravel_index(L.argmax(), L.shape)
        alpha = 0.5 * np.arctan(2*A[i, j] / (A[i, i]-A[j, j]))
```

```python

        V = np.eye(A.shape[0])
        V[i, i], V[j, j] = np.cos(alpha), np.cos(alpha)
        V[i, j], V[j, i] = -np.sin(alpha), np.sin(alpha)

        A = np.dot(V.T, A.dot(V))
        U = U.dot(V)
        iterations += 1
        yield A


@plot_factory
def qrm1():
    """
    Create generator for transformed matrices after applying the QR-Method.

    Yields:
        - T: 2D-numpy array. Similar matrix to X.
    """
    # First stage: transform to upper Hessenberg-matrix.
    T = copy.deepcopy(X)

    k = 0
    # Second stage: perform QR-transformations.
    while k < 5000:
        k += 1
        Q, R = np.linalg.qr(T)
        T = R.dot(Q)
        yield T


@plot_factory
def qrm2():
    """
    Create generator for transformed matrices after applying the
      QR-Method.

    Yields:
        - T: 2D-numpy array. Similar matrix to X.
    """
    # First stage: transform to upper Hessenberg-matrix.
    T = lin.hessenberg(X)

    k = 0
    # Second stage: perform QR-transformations.
    while k < 5000:
        if k == 0:
            yield X
        k += 1
        Q, R = np.linalg.qr(T)
        T = R.dot(Q)
        yield T


@plot_factory
def qrm3():
    """
    First compute similar matrix in Hessenberg form, then compute
      the
    Eigenvalues and Eigenvectors using the accelerated QR-Method.

    Yields:
        * T - 2D numpy array of current iteration step.
    """
    # First stage: transform to upper Hessenberg-matrix.
    T = lin.hessenberg(X)
    k = 0
    n, _ = X.shape

    # Second stage: perform QR-transformations.
    while k < 5000:
        if k == 0:
            yield X
        k += 1
        Q, R = np.linalg.qr(T - T[n-1, n-1] * np.eye(n))
        T = R.dot(Q) + T[n-1, n-1] * np.eye(n)

        yield T


jacobi(os.path.join(outpath, "jacobi.png"))
qrm1(os.path.join(outpath, "qrm1.png"))
qrm2(os.path.join(outpath, "qrm2.png"))
qrm3(os.path.join(outpath, "qrm3.png"))

plt.show()
plt.close()
```

6.5 Analysis: Unit tests

```python
"""
Automated tests for different algorithms.
"""
import os
import sys
import numpy as np
from threading import Thread
import pandas as pd
from algorithms import eigen
from scipy.stats import ortho_group
from tqdm import tqdm
from functools import wraps


data_out = os.path.join("data", "accuracy_tests.csv")


class AlgoTest(object):
    tests = {"algorithm": [],
             "dimension": [],
             "maxiter": [],
             "failed": []}

    def __init__(self, algo, dim, filepath, n_tests=1000, jobs=1,
                 *args, **kwargs):
        assert n_tests % jobs == 0
        self.algorithm = self.__get_test_algorithm(algo, *args, **
          kwargs)
        self.dim = dim
        self.n = n_tests // jobs
        self.failed = []
        self.jobs = jobs
        self.result = None
        self.path = filepath
        self.maxiter = kwargs.get("maxiter", None)

        if not os.path.exists(self.path):
            self.save(header=["algorithm", "dimension", "maxiter", "failed"])

    def __get_test_algorithm(self, algorithm, *args, **kwargs):
        @wraps(algorithm)
        def algo(X):
            return algorithm(X, *args, **kwargs)
        return algo

    def __get_test_matrix(self):
        """Return matrix with assosiated Eigenvalues."""
        eigenvalues = np.random.uniform(size=self.dim)
        eigenvectors = ortho_group.rvs(dim=self.dim)
        Lambda = np.diag(eigenvalues)

        matrix = np.dot(eigenvectors, Lambda).dot(eigenvectors.T)

        order = np.abs(eigenvalues).argsort()[::-1]
        return matrix, eigenvalues[order]

    def __run_test(self):
        """Run singular test."""
        mat, true_eig = self.__get_test_matrix()
        test_eig, _ = self.algorithm(mat)
        test_res = np.alltrue(np.isclose(true_eig, test_eig))
        self.failed.append(not test_res)

    def __run_tests(self):
        """Run multiple tests."""
        for _ in range(self.n):
            try:
                self.__run_test()
            except (KeyboardInterrupt, SystemExit):
                self.__save()
                sys.exit(0)

    def run(self):
        """Distribute tests accross threads."""

        threadlist = [None] * self.jobs

        for i in range(self.jobs):
            threadlist[i] = Thread(target=self.__run_tests, daemon
              =True)
            threadlist[i].start()

        for thread in threadlist:
            thread.join()

        self.result = sum(self.failed)
        self.tests["algorithm"].append(self.algorithm.__name__)
        self.tests["dimension"].append(self.dim)
        self.tests["failed"].append(self.result)
        if self.algorithm.__name__ == "jacobi":
            self.tests["maxiter"].append(None)
        else:
            self.tests["maxiter"].append(self.maxiter)

    def save(self, header=False):
        if os.path.exists(self.path):
                print("Saving results.")
        df = pd.DataFrame(self.tests)
        with open(self.path, 'a') as f:
            df.to_csv(f, header=header, index=False,
                      columns=["algorithm", "dimension", "maxiter"
                        , "failed"])

```

```python

# Unit tests
if __name__ == "__main__":
    # Define Flags.
    JOBS = 20
    MAXITER = (10, 100, 1000, 10000)
    DIMS = range(3, 8)
    ALGOS = {'jacobi': eigen.jacobi,
             'qrm': eigen.qrm,
             'qrm2': eigen.qrm2,
             'qrm3': eigen.qrm3}

    # Define parameters of all runs.
    parameters = []
    for algo in ALGOS.values():
        for maxiter in MAXITER:
            for dim in range(3, 8):
                param = (algo,
                         maxiter if algo.__name__ != "jacobi" else
                            None,
                         dim)
                parameters.append(param)

    # Check progress of previous runs.
    if os.path.exists(data_out):
        required = [(algo.__name__, m, dim) for (algo, m, dim) in
          parameters]
        required = set(required)

        progress = pd.read_csv(data_out)
        done = []
        for (a, d, m, _) in progress.values:
            if np.isnan(m):
                param = (a, d, None)
            else:
                param = (a, d, m)
            done.append(param)
        done = set(done)

        to_do = required.difference(done)
        parameters = [(ALGOS[a], m, d) for a, m, d in to_do]

    for algo, maxiter, dim in tqdm(parameters):
                algo_test = AlgoTest(filepath=data_out,
                                     algo=algo,
                                     dim=dim,
                                     maxiter=maxiter,
                                     jobs=JOBS)
                algo_test.run()

    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.