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 28, 2018

Contents

List of Tables

List of Figures

List of Algorithms

1 Motivation

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

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

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

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

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

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

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

When confronted with a high-dimensional data matrix $X \in \mathbb{R}^{n \times m}$ an analyst often wishes to find a lower-dimensional representation, while conserving as much of the structure as possible. One way of achieving this goal is to choose a standardized linear combination of features that aim to maximize the variance of the projection $\delta' X$. We can formalize this as

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

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

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

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

Taking derivatives we obtain the first order condition:

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

Which is now reduced to a common Eigenvalue problem as proposed 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

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

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

This transformation defined in 5 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

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

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*.

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

$$x = c_1u + c_2 + v$$

for some scalars c_1 and c_2. The vector

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

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

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

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

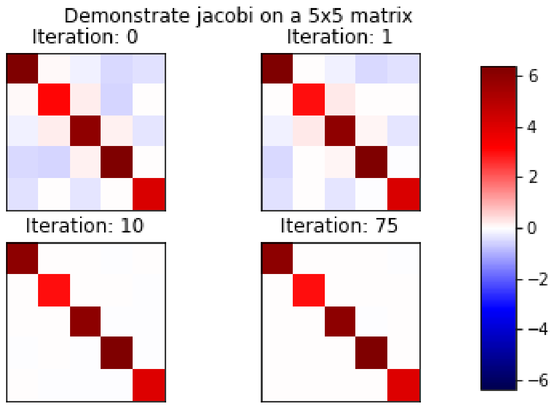
2.2 Givens-Rotations

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.

$$V_{pq}(\theta) = \begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & \cos\theta & & \sin\theta & & \\ & & & \ddots & & & \\ & & -\sin\theta & & \cos\theta & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{bmatrix} \tag{7}$$

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

Algorithm 2 `QRM1`

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

Figure 2: Progress basic QR-Method

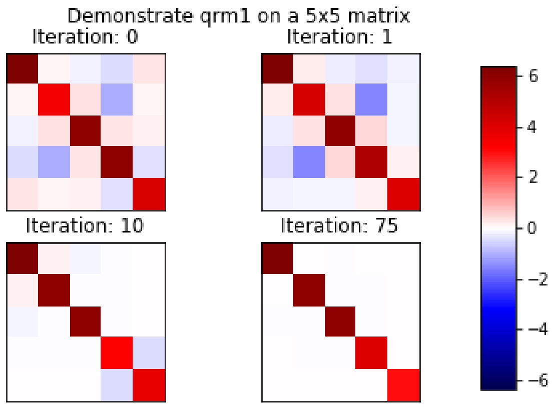


Figure 3: Progress Hessenberg-QR-Method

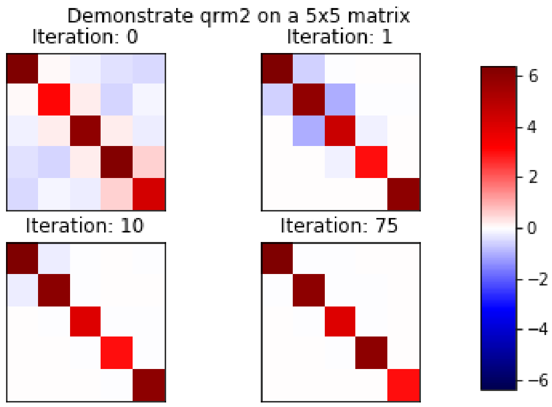


Table 1: Unit tests accross matrix-sizes

awesome	sauce
nothing	to
see	here

3.2 QR-Method

3.2.1 Hessenberg Variant

Algorithm 3 `QRM2`

Require: square matrix A

1: $A \leftarrow$ `hessenberg`(A)

2: continue with: QRM1(A)

Algorithm 4 `QRM3`

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

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

Figure 4: Progress Accelerated QR-Method

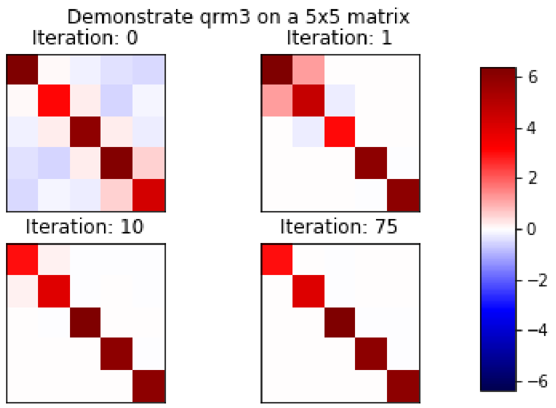
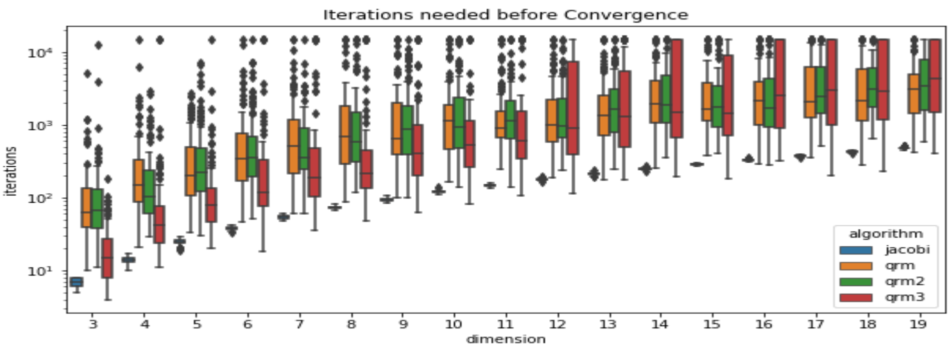


Figure 5: Unit-tests: Iterations



3.2.2 Accelerated Variant

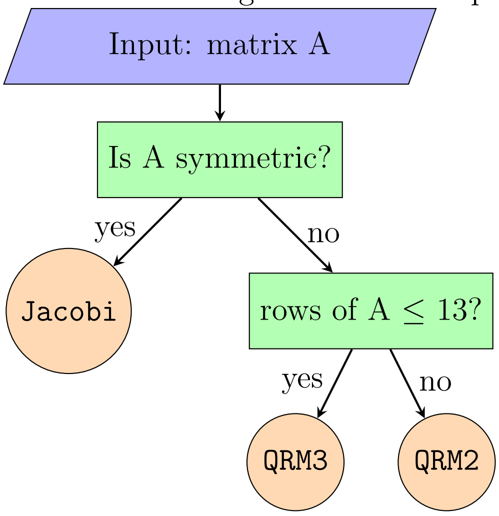
4 Analysis

4.1 Accuracy

4.2 Efficiency

5 Conclusion

Figure 6: Decision process of final eigenvalue routine



6 Appendix

6.1 Eigenvalue Routines

```
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:
```

```python
        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

    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, :]
```

6.2 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()

# 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]))

        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()
```

6.3 Analysis: Unit tests

```
"""
Automated tests for different algorithms.
"""
```

```
import os
import numpy as np
import threading
import pandas as pd
from algorithms import eigen
from scipy.stats import ortho_group
from tqdm import trange, tqdm

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


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

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

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


def test_algo(algo, Ntests=1000, dim=3, *args, **kwargs):
    """
    Test routine that allows for threading. Note that the variables:
    failed, critical and problematic need to be defined in the enveloping or
    global scope beforehand.
```

```

    Parameters:
        - algo: algorithm to be tested
        - Ntests: number of tests to compute
        - dim: dimensions of matrix
        - *args, **kwargs: additional arguments to be passed to algo.

    Returns:
        - None, but will update the variables failed, critical and problematic.
            + failed: number of failed tests
            + critical: number of ZeroDivisionErrors
            + problematic: list of numpy arrays which led to wrong eigenvalues.
    """
    global failed
    global critical
    global problematic

    for _ in range(Ntests):
        try:
            A, true_eig = get_test_matrix(dim=dim)
            my_eig, _ = algo(A, *args, **kwargs)
            assert np.alltrue(np.isclose(my_eig, true_eig))

        except AssertionError:
            failed += 1
            problematic.append(A)

```

```
        except ZeroDivisionError:
            critical += 1


def threaded_tests(algo, N, nWorkers=10, verbose=True, *args, **
  kwargs):
    global failed
    global critical
    global problematic

    assert N % nWorkers == 0

    n = N // nWorkers
    threadlist = [None] * nWorkers

    for i in range(nWorkers):
        threadlist[i] = threading.Thread(target=test_algo,
                                         args=(algo, n, *args))
        threadlist[i].start()

    for i in range(nWorkers):
        threadlist[i].join()

    logstr = """
    {} out of {} tests failed.
    {} tests failed critically.
    """.format(failed, N, critical)

    if verbose:
        print(logstr)
```

```python


# Tests
results = {
    "algorithm": [],
    "dimension": [],
    "maxiter": [],
    "failed": []}

for algo in tqdm([eigen.jacobi, eigen.qrm, eigen.qrm2, eigen.qrm3
 ]):
    for dim in trange(3, 15):
        for maxiter in 1000, 10000, 100000:
            if algo.__name__ == eigen.jacobi:
                maxiter = 1e-6
            failed = 0
            critical = 0
            problematic = []
            threaded_tests(algo, 1000, 20, False, dim, maxiter)
            results["algorithm"].append(algo.__name__)
            results["dimension"].append(dim)
            results["maxiter"].append(maxiter)
            results["failed"].append(failed)

test_data = pd.DataFrame(results)
test_data.to_csv(data_out)
```

7 References

[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.