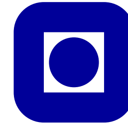
Norwegian University of Science and Technology
Department of Mathematical Sciences



MA2501 Numerical Methods
Eirik Hoel Høiseth

Semester Project

- This project counts for 30% of the final grade.
- You may work alone or in groups of two people.
- You have to produce a report written on a computer with your solutions (preferably in LaTeX, but there is nothing wrong with using something else). The report should **not exceed 8 pages**.

  You do not need to include code in your report. It is enough to refer to the code files produced. It is fine to just go through the tasks point by point and answer them. Just try to keep the document organized and readable.
- Try to write somewhat efficient, organized, and well documented MATLAB code.
- The report, and the code produced, should be sent electronically to eirik.hoiseth@math.ntnu.no. The deadline for everything is **20 March at 18:00**. Mark all the material with your candidate number(s), not your name(s).
- Problems are roughly worth the same when grading.

Matlab advice

All Matlab code created in this project should satisfy the following criteria:

- Function files should contain a help text. A user should be able to use the routine from the information given by the command:
  `help [function name]`.
- The code should be self-documented, with a reasonable amount of comments in the code.
- For the iterative methods, a warning message should be printed if the iterations do not converge.

In addition try to make use of Matlab's proficiency at working with vectors and matrices, i.e. avoid unnecessary for-loops. Also try to avoid stuff like repeating the same computation several times, rather than computing the result once and storing it. Feel free to make use of built-in functions like `norm`, `max`, `sort`, etc. Specifically feel free to use `eig` to get accurate values of the eigenvalues and eigenvectors to compare with.

Estimating eigenvalues and eigenvectors

a) Implement the power method (with normalization), from Cheney & Kincaid, for computing eigenvalues and eigenvectors of a matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ in Matlab. The first line of the function should be:

```
function [r,x,nIt,iFlag] = stdPowEig(A,x0,tol,nMax)
```

Specifications:

- **Arguments in:**
  - `A`: An $n \times n$ real matrix.
  - `x0`: An initial guess for the eigenvector of $\mathbf{A}$ associated with the dominant eigenvalue.
  - $\epsilon \equiv$ `tol`: An error tolerance.
  - `nMax`: The maximum number of allowed iterations.
- **Arguments out:**
  - r: The approximate value of the dominant eigenvalue of $\mathbf{A}$.
  - `x`: The approximation of the eigenvector of $\mathbf{A}$ associated with the dominant eigenvalue.
  - `nIt`: The number of iterations used.
  - `iFlag`: A flag, telling whether the iterations were successful or not.
- **Additional details:**

– Let $r^{(k)}$ and $\mathbf{x}^{(k)}$ denote the estimates after k iterations of the dominant eigenvalue λ_1 and corresponding eigenvector $\mathbf{v}_1$. Stop the iterations when $\|\mathbf{x}^{(k)} - \text{sgn}(r^{(k)})\mathbf{x}^{(k-1)}\| \leq \epsilon$ (success) or $\texttt{nIt} = \texttt{nMax}$ (failure)

b) Test your power method on the following matrices

$$\mathbf{A}_1 = \begin{bmatrix} 4 & 4 & 4 & -2 \\ 3 & 1 & 1 & -1 \\ 2 & -1 & 2 & 0 \\ -1 & 5 & 1 & -2 \end{bmatrix}, \quad \mathbf{A}_2 = \begin{bmatrix} 5 & -1 & 3 & -1 \\ -1 & 5 & -3 & 1 \\ -1 & 1 & 1 & 1 \\ 1 & -1 & 3 & 3 \end{bmatrix},$$

$$\mathbf{A}_3 = \begin{bmatrix} -14 & -20 & 6 & 4 \\ -20 & -8 & 6 & 16 \\ 6 & 6 & 9 & 24 \\ 4 & 16 & 24 & 4 \end{bmatrix}.$$

Use several different random initial vectors and limit the calculations to at most 20 iterations. Comment on the behaviour of the eigenvalue and eigenvector sequences. Try to explain the observed behaviour.

c) For the remaining tasks assume the matrices are symmetric. For the purposes of eigenvalue and eigenvector computations, symmetric matrices have the following useful properties.

- The eigenvalues are real.
- There exists a full set of n eigenvectors $\mathbf{v}_1, \mathbf{v}_1, \ldots, \mathbf{v}_n$ that are orthonormal with respect to the standard inner product on $\mathbb{R}^n$. That is for $i, j = 1, 2, \ldots, n$

$$\mathbf{v}_i^T \mathbf{v}_j = \delta_{ij} = \begin{cases} 1 & \text{if } \ i = j, \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

- The matrix is orthogonally similar to a diagonal matrix (more on this in the appendix).

Implement the following modification to the power method for symmetric matrices. Let the eigenvalues be ordered by decreasing order of magnitude $|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq .... \geq |\lambda_n|$. Again, let $r^{(k)}$ and $\mathbf{x}^{(k)}$ denote the estimates after k iterations of the dominant eigenvalue λ_1 and corresponding eigenvector $\mathbf{v}_1$. Normalize the initial vector $\mathbf{x}^{(0)}$ to unit length, i.e. ${\mathbf{x}^{(0)}}^T \mathbf{x}^{(0)} = \|\mathbf{x}^{(0)}\|_2^2 = 1$, and compute new estimates as follows: For $k = 1, 2, 3, \ldots$, calculate

$$\begin{aligned} \mathbf{y}^{(k)} &= \mathbf{A}\mathbf{x}^{(k-1)}, \\ r^{(k)} &= {\mathbf{x}^{(k-1)}}^T \mathbf{y}^{(k)}, \\ \mathbf{x}^{(k)} &= \mathbf{y}^{(\mathbf{k})} / \|\mathbf{y}^{(k)}\|_2. \end{aligned}$$

Call the modified procedure `symPowEig`, and keep the rest of the procedure identical to `stdPowEig`. Let $e^{(k)}$ denote the eigenvalue error in computing λ_1 after k steps, i.e. $e^{(k)} = |r^{(k)} - \lambda_1|$. Verify experimentally that for both methods, the dominant eigenvalue converges linearly like

$$e_{k+1} \approx e_k \left(\frac{|\lambda_2|}{|\lambda_1|}\right)^p$$

for some positive integer p. Determine the value of p for both methods.

d) Aitken acceleration can be used in `symPowEig` to speed up convergence. Modify `symPowEig` to include Aitken acceleration. Call the modified procedure `symPowEigAit`. Investigate experimentally how this modification impacts convergence of the eigenvalue and eigenvector.
Hint: It is possible to also use Aitken acceleration elementwise on the eigenvector iterates $\mathbf{x}^{(k)}$, since they also converge linearly.

e) Modify `symPowEigAit` to include the option to choose between the regular and inverse power method, including shifts. To implement the inverse power method, you may use MATLAB to solve the linear system $\mathbf{Ax} = \mathbf{b}$ efficiently using the LU-factorization with row permutations

1) `[L,U,p] = lu(A,'vector');`
2) `x = U\(L\(b(p,:)));`

Call the method `symEig`.

f) Consider the following matrix

$$\mathbf{B} = \mathbf{A} - \lambda_i \mathbf{v}_i \mathbf{v}_i^T,$$

where (1) is assumed to hold. How does the eigenvectors and eigenvalues of $\mathbf{B}$ compare to those of $\mathbf{A}$?

g) Suggest and implement an algorithm to compute the m largest (in absolute value) eigenvalues of $\mathbf{A}$ by modifying `symPowEigAit` based on the previous result. Call the modified procedure `symEigComp`. The first line should be
`function [r,X,iFlag] = symEigComp(A,x0,m,tol,nMax)`
where the modified arguments in and out are.

- **Arguments in:**
  - `A`: An $n \times n$ real matrix.
  - `x0`: An initial guess for an eigenvector of $\mathbf{A}$.
  - `m` the number of required eigenvalues.
  - $\epsilon \equiv$ `tol`: The error tolerance for an eigenvalue-eigenvector pair.

  - `nMax`: The maximum number of allowed iterations for an eigenvalue-eigenvector pair.
- **Arguments out:**
  - **r**: An $m \times 1$ vector with the eigenvalue approximations to the m largest eigenvalues of $\mathbf{A}$.
  - **X** An $n \times m$ matrix whose column vectors are the eigenvector approximations.
  - `iFlag`: An $m \times 1$ vector flag, telling which of the m estimations that were successful.

h) Read the appendix part about Householder's method. Show that $\hat{\mathbf{w}}$ given by (8) solves (7).

i) Implement Householder's method as described in the appendix. The algorithm should take in a symmetric matrix $\mathbf{A}$, and reduce this to a similar symmetric tridiagonal matrix by calling Householder's method. Call the function `householders`. It should return just the nonzero elements of this matrix as column vectors **a** and **b** corresponding to (10).

j) Implement the full QR-algorithm as described in the appendix. The algorithm should take in a symmetric matrix $\mathbf{A}$, reduce this to a similar symmetric tridiagonal matrix by calling `householders`, and then determine and return the full set of eigenvalues of this matrix, using the described algorithm. Call this function `qrAlg`. Test the algorithm on the symmetric $n \times n$ matrix $\mathbf{A} = (a_{ij})$ whose upper triangular part is given by $a_{ij} = n + 1 - j$ for $1 \leq i \leq j \leq n$. This matrix has known eigenvalues

$$\lambda_i = \frac{1}{2 - 2\cos\left[\pi\frac{2i-1}{2n+1}\right]} \quad \text{for} \quad i = 1, 2, 3 \ldots, n.$$

Verify the eigenvalues experimentally for some reasonably large $n > 300$. Plot the errors in the computed eigenvalues vs. i using logarithmic y-axis.

k) Briefly describe a way we could modify the QR-algorithm to also compute the eigenvectors.

Hint: Recall that for similar matrices $\mathbf{A}$ and $\mathbf{B}$ related by (2), eigenvectors $\mathbf{v_A}$ and $\mathbf{v_B}$ corresponding to the same eigenvalue are related by $\mathbf{v_A} = \mathbf{M}\mathbf{v_B}$

l) **Application**: On the course web page you will find a MATLAB data file `faces.mat`. This contains 10 grey scale face images of 40 different subjects[1]. Each image is represented as a 92×112 matrix of intensity

[1] The images are credited to the Database of Faces from AT&T Laboratories Cambridge

values between 0 and 1, where 0 equals black and 1 white. After loading `faces.mat` you will have a variable faces which contains the images stored as a 4 dimensional array (4 indexes are needed to give the position of an element). The pixel in position (i, j) of the image matrix of face image number k of person l is stored in `faces(l,k,i,j)`. The range of these indexes are thus $1 \leq l \leq 40$, $1 \leq k \leq 10$, $1 \leq i \leq 92$ and $1 \leq j \leq 112$. To display image k of person l use the command `imshow(squeeze(faces(l,k,:,:))`.

Read the first part of the appendix on computing eigenfaces. Divide the integers from 1 to 10 into 2 random sets, one S_{tr} with 7 and one S_{ts} with the remaining 3. Different groups are expected choose different sets. Report your choice. The training set contains the 7 images for each person which correspond to image numbers $k \in S_{tr}$. The remaining images, i.e. the ones for which the images number $k \in S_{ts}$, make up the test set.

Compute the eigenfaces for your training set. Use the QR algorithm to compute the eigenvalues. Use the inverse power method, shifted with the computed eigenvalues, to compute the eigenvectors and possibly refine the returned eigenvalues. If you have not succeded in implementing the QR-algorithm, you may use the built in MATLAB function `eig`. Display the first 8 eigenfaces $\mathbf{u}_k$ $k = 1, 2, \ldots, 8$ as greyscale faces[2], along with the average face $\mathbf{\Psi}$.

Note: The MATLAB function `reshape` is useful for transforming image matrices to column vectors and vice versa.

m) Attempt to classify the images in your test set based on the approach in the appendix. Choose the number of eigenfaces $\tilde{m}$ to use for the face space such that 90% of the variation in the training set is accounted for, i.e. choose the smallest $\tilde{m}$ such that

$$\frac{\sum_{i=1}^{\tilde{m}} \lambda_i}{\sum_{i=1}^{m} \lambda_i} \geq 0.9$$

where the eigenvalues λ_i are sorted by decreasing absolute value. Then try to identify all the images in the test set. Report some relevant results from the experiment, and comment on the success of the method. Also choose a correctly identified image $\mathbf{\Gamma}$ from the test set and include in the report:

- A figure where you display (as grey scale images) $\mathbf{\Gamma}$, the projection of this image in face space Ω, and the face class $\Omega^{(k)}$ it was identified to.
- A plot of the Euclidean distance in face space ϵ_k against $k = 1, 2, \ldots 40$.

[2] Here it is a good idea to plot a scaled version of the eigenfaces $\mathbf{u}_k / \|\mathbf{u}_k\|_\infty$

Briefly discuss the following: How small can you choose $\tilde{m}$ before performance deteriorates? What are some restrictions on the face images you suspect are necessary for the method to work? Is the minimum face class distance (15) significantly higher for the faces which are wrongly identified? Would it make sense to implement an upper limit for this metric, such that the face is considered unknown if the value is higher than this.

Appendices

A Finding all eigenvalues of a symmetric matrix

From the perspective of exact arithmetic, `symEigComp` from task g) could be used to determine all eigenvalues and eigenvectors of a matrix, provided the eigenvalues all had distinct size. However, accumulation of roundoff error makes this method impractical for computing more than the first few largest eigenvalues and eigenvectors. We therefore look for an algorithm that can compute all the eigenvalues of a symmetric matrix [3]. Our chosen algorithm consists of two main steps:

1. Reduce the symmetric matrix $\mathbf{A}$ to a similar symmetric tridiagonal matrix using Householder's method.

2. Apply the QR-algorithm to further reduce this symmetric tridiagonal matrix to a similar matrix that is nearly diagonal. As the matrix becomes nearly diagonal, the diagonal elements become nearly the eigenvalues of this matrix, which are the same as the eigenvalues of $\mathbf{A}$.

In both these steps the matrix is reduced using orthogonal similarity transforms. Recall that an $n \times n$ matrix $\mathbf{B}$ is said to be similar to an $n \times n$ matrix $\mathbf{A}$ if there exists an invertible $n \times n$ matrix $\mathbf{M}$ such that

$$\mathbf{B} = \mathbf{M}^{-1}\mathbf{A}\mathbf{M}. \tag{2}$$

Furthermore $\mathbf{A}$ and $\mathbf{B}$ have the same eigenvalues. To do similarity transforms, i.e. convert $\mathbf{A}$ to $\mathbf{B}$, it is desirable to use matrices $\mathbf{M}$ that are simple to invert. Orthogonal matrices, i.e invertible $n \times n$ matrices $\mathbf{Q}$ such that $\mathbf{Q}^{-1} = \mathbf{Q}^T$ are clearly well suited for this purpose. This choice is motivated by the fact that a symmetric matrix $\mathbf{A}$ is orthogonally similar to a diagonal matrix $\mathbf{D}$, i.e. there exists an orthogonal matrix $\mathbf{Q}$ such that

$$\mathbf{D} = \mathbf{Q}^T\mathbf{A}\mathbf{Q}.$$

[3] The algorithm can also be modified to compute the eigenvectors, see task k)

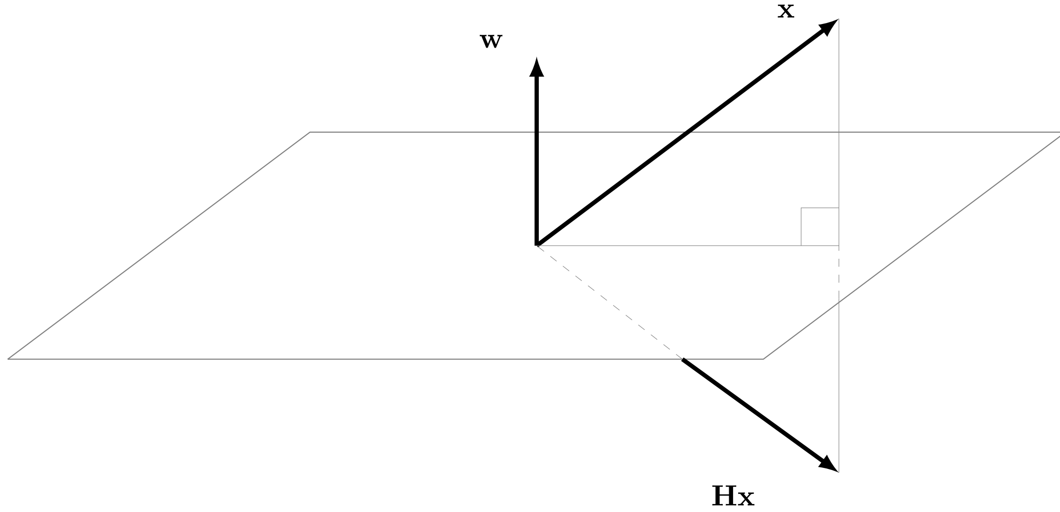


Figure 1: **Hx** corresponds geometrically to a reflection of a vector **x** across the hyperplane whose normal vector is **w**

A.1 Householder's method

As mentioned, Householder's method reduces the symmetric matrix **A** to a similar symmetric tridiagonal matrix.

This method uses similarity transforms based on $n \times n$ Householder matrices, **H**, defined as

$$\mathbf{H} = I - 2\frac{\mathbf{w}\mathbf{w}^T}{\mathbf{w}^T\mathbf{w}}$$

where **I** is the $n \times n$ identity matrix and $\mathbf{w} \in \mathbb{R}^n$ is a non-zero column vector. It is simple to show that **H** is both symmetric and orthogonal, meaning that $\mathbf{H}^{-1} = \mathbf{H}$. This again implies that the similarity transform **HAH** preserves the symmetry of **A**. A geometric interpretation of how **H** operates on a vector **x** is shown in Figure 1.

Householder's method performs $n-2$ such similarity transforms on **A**. Let $\mathbf{A}^{(1)} = \mathbf{A}$, $\mathbf{A}^{(k)}$ be the matrix after $k-1$ similarity transforms have been applied, and $\mathbf{H}^{(k)}$ be the Householder matrix used in the k-th similarity transform, i.e.

$$\mathbf{A}^{(k+1)} = \mathbf{H}^{(k)}\mathbf{A}^{(k)}\mathbf{H}^{(k)}. \tag{3}$$

The trick is then to choose these transformations such that the final matrix

$$\mathbf{A}^{(n-1)} = \mathbf{H}^{(n-2)}\mathbf{H}^{(n-3)}\dots\mathbf{H}^{(2)}\mathbf{H}^{(1)}\mathbf{A}\mathbf{H}^{(1)}\mathbf{H}^{(2)}\dots\mathbf{H}^{(n-3)}\mathbf{H}^{(n-2)}$$

is tridiagonal. The transformation with $\mathbf{H}^{(k)}$ places zeros in the last $n-k-1$ elements of the k-th column, while preserving all previously created zeros below the diagonal. Due to symmetry the same thing happens above the

diagonal for the k-th row. The process is shown in (4) for a 5×5 matrix

$$\begin{aligned}
\mathbf{A}^{(1)} = \mathbf{A} &= \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \\
\mathbf{A}^{(2)} = \mathbf{H}^{(1)}\mathbf{A}\mathbf{H}^{(1)} &= \begin{bmatrix} \times & \times & 0 & 0 & 0 \\ \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \end{bmatrix} \\
\mathbf{A}^{(3)} = \mathbf{H}^{(2)}\mathbf{H}^{(1)}\mathbf{A}\mathbf{H}^{(1)}\mathbf{H}^{(2)} &= \begin{bmatrix} \times & \times & 0 & 0 & 0 \\ \times & \times & \times & 0 & 0 \\ 0 & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times \\ 0 & 0 & \times & \times & \times \end{bmatrix} \qquad (4) \\
\mathbf{A}^{(4)} = \mathbf{H}^{(3)}\mathbf{H}^{(2)}\mathbf{H}^{(1)}\mathbf{A}\mathbf{H}^{(1)}\mathbf{H}^{(2)}\mathbf{H}^{(3)} &= \begin{bmatrix} \times & \times & 0 & 0 & 0 \\ \times & \times & \times & 0 & 0 \\ 0 & \times & \times & \times & 0 \\ 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & \times & \times \end{bmatrix}
\end{aligned}$$

Each $\times$ here denotes an element which is not necessarily 0.

We now consider the problem of determining the k-th transformation matrix $\mathbf{H}^k$, and leave out the superscript for ease of notation. Thus $\mathbf{H} \equiv \mathbf{H}^{(k)}$, $\mathbf{A} \equiv \mathbf{A}^{(k)}$ and $\mathbf{w} \equiv \mathbf{w}^{(k)}$ It turns out that this problem then amounts to finding a vector $\mathbf{w} \neq \mathbf{0}$, such that $\mathbf{H}$ satisfies

$$\mathbf{H} \begin{bmatrix} a_{1k} \\ a_{2k} \\ \vdots \\ a_{kk} \\ a_{k+1,k} \\ a_{k+2,k} \\ \vdots \\ a_{n-1,k} \\ a_{n,k} \end{bmatrix} = \begin{bmatrix} a_{1k} \\ a_{2k} \\ \vdots \\ a_{kk} \\ \alpha \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}, \qquad (5)$$

where α is some real number. This gives n conditions for the n unknowns w_i $i = 1, 2, \ldots, n$. Leaving $a_{i,k}$ unchanged for $i = 1, 2, \ldots, k$ is achieved by setting

$$w_1 = w_2 = \cdots = w_k = 0. \qquad (6)$$

Now, define

$$\hat{\mathbf{w}} = \begin{bmatrix} w_{k+1} \\ w_{k+2} \\ \vdots \\ w_{n-1} \\ w_n \end{bmatrix} \in \mathbb{R}^{n-k}, \quad \hat{\mathbf{a}} = \begin{bmatrix} a_{k+1,k} \\ a_{k+2,k} \\ \vdots \\ a_{n-1,k} \\ a_{n,k} \end{bmatrix} \in \mathbb{R}^{n-k},$$

$$\hat{\mathbf{H}} = I - 2\frac{\hat{\mathbf{w}}\hat{\mathbf{w}}^T}{\hat{\mathbf{w}}^T\hat{\mathbf{w}}} \in \mathbb{R}^{(n-k)\times(n-k)},$$

and let $\mathbf{I}$ be the $k \times k$ identity matrix and $\mathbf{e_i} \in \mathbb{R}^{(n-k)}$ be the vector where the i-th element is 1 and all others 0. We see that (5) using (6) can be written

$$\begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \hat{\mathbf{H}} \end{bmatrix} \begin{bmatrix} a_{1k} \\ \vdots \\ a_{kk} \\ \hat{\mathbf{a}} \end{bmatrix} == \begin{bmatrix} a_{1k} \\ \vdots \\ a_{kk} \\ \alpha\mathbf{e_1} \end{bmatrix}$$

and the problem reduces to finding $\hat{\mathbf{w}} \neq \mathbf{0}$ such that

$$\hat{\mathbf{H}}\hat{\mathbf{a}} = \alpha\mathbf{e_1} \tag{7}$$

A solution to (7) is

$$\hat{\mathbf{w}} = \hat{\mathbf{a}} \pm \|\hat{\mathbf{a}}\|_2\mathbf{e_1},$$

and to ensure $\hat{\mathbf{w}}$ is non-zero, the sign is chosen to be

$$\hat{\mathbf{w}} = \begin{cases} \hat{\mathbf{a}} + \operatorname{sgn}(a_{k+1,k})\|\hat{\mathbf{a}}\|_2\mathbf{e_1}, & \text{if } \ a_{k+1,k} \neq 0 \\ \hat{\mathbf{a}} + \|\hat{\mathbf{a}}\|_2\mathbf{e_1}, & \text{if } \ a_{k+1,k} = 0 \end{cases} \tag{8}$$

A few points which should be used to make the implementation reasonably efficient are:

- In practice there is no need to form $\mathbf{H}$ since $\mathbf{H}$ is given by the choice of $\mathbf{w}$.
- If we first normalize $\mathbf{w}$, so that $\|\mathbf{w}\|_2^2 = 1$, we can write $\mathbf{H}$ as $\mathbf{H} = I - 2\mathbf{w}\mathbf{w}^T$. Computation of the similarity transform $\mathbf{HAH}$ is then tremendously simplified if we compute this as

$$\begin{aligned} \mathbf{u} &= \mathbf{A}\mathbf{w} \\ k &= \mathbf{w}^T\mathbf{A}\mathbf{w} = \mathbf{w}^T\mathbf{u} \\ \mathbf{q} &= \mathbf{u} - k\mathbf{w} \\ \mathbf{HAH} &= \mathbf{A} - 2(\mathbf{w}\mathbf{q}^T + \mathbf{q}\mathbf{w}^T) \end{aligned} \tag{9}$$

Further simplification is possible by exploiting that in step $k \geq 1$ the first k elements of $\mathbf{w}$ are 0. From the format of $\mathbf{A}$ the first $k-1$ elements of $\mathbf{u}$ and consequently $\mathbf{q}$ are also 0. This implies that the we only have to compute the last $n-k+1$ elements of $\mathbf{u}$ and $\mathbf{q}$, and only need to change the last $n-k+1$ rows and columns of $\mathbf{A}$.

A.2 QR-factorization

In general the QR-factorization for square matrices factors the $n \times n$ matrix $\mathbf{A}$ as $\mathbf{A} = \mathbf{QR}$ where $\mathbf{Q}$ is an $n \times n$ orthogonal matrix and $\mathbf{R}$ is an upper triangular matrix. This is a similar problem to the LU factorization, but with the lower triangular $\mathbf{L}$ replaced by the orthogonal $\mathbf{Q}$.

From the application of Householders method, we may assume that the initial matrix $\mathbf{A} \equiv \mathbf{A}^{(1)}$ is symmetric tridiagonal

$$\mathbf{A} = \begin{bmatrix} a_1 & b_1 & 0 & \dots & 0 \\ b_1 & a_2 & b_2 & \ddots & \vdots \\ 0 & b_2 & a_3 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & b_{n-1} \\ 0 & \dots & 0 & b_{n-1} & a_n \end{bmatrix}. \tag{10}$$

This simple structure allows for efficient computation of $\mathbf{Q}$ and $\mathbf{R}$ and the product $\mathbf{RQ}$ using rotation matrices. Though the theory behind this is not very complicated, a function `qrStep` which does this, has been provided in order to limit the scope of the project.

A.3 QR-algorithm

The initial matrix is called $\mathbf{A}^{(1)}$, again assumed on the form (10), i.e. symmetric and tridiagonal. Having computed the matrix $\mathbf{A}^{(k)}$ $k \geq 1$ the QR-algorithm computes $\mathbf{A}^{(k+1)}$ by first choosing a shift $\sigma_k \in \mathbb{R}$ and then

1. Factoring $\mathbf{A}^{(k)} - \sigma_k \mathbf{I}$ as $\mathbf{Q}^{(k)}\mathbf{R}^{(k)}$.
2. Compute $\mathbf{A}^{(k+1)} = \mathbf{R}^{(k)}\mathbf{Q}^{(k)} + \sigma_k \mathbf{I}$.

Since $\mathbf{Q}^{(k)}$ is orthogonal it is clear that

$$\mathbf{A}^{(k+1)} = \mathbf{Q}^{(k)^T}\mathbf{A}^{(k)}\mathbf{Q}^{(k)}.$$

This means each step performs an orthogonal similarity transform, which as mentioned preserves the eigenvalues of the matrix. The sequence of matrices $\{\mathbf{A}^{(k)}\}_{k=1}^{\infty}$ generated by this algorithm converges, with carefully chosen shifts, rapidly towards a diagonal matrix. Because the eigenvalues of a diagonal matrix are simply the diagonal elements, the diagonal elements of the matrices $\mathbf{A}^{(k)}$ converge towards the eigenvalues of the original matrix.

It is not hard to show that $\mathbf{A}^{(k+1)}$ will be symmetric tridiagonal if $\mathbf{A}^{(k)}$ is. Thus $\mathbf{A}^{(k)}$ for all $k \geq 1$ has this beneficial structure, making all the iterations efficient. This was the motivation for applying Householders method to the original symmetric matrix.

In accordance with (10), let now $a_i^{(k)}$ and $b_i^{(k)}$ denote the diagonal and off-diagonal elements of $\mathbf{A}^{(k)}$, and λ_i denote the eigenvalue $a_i^{(k)}$ converges towards. We could use the QR-algorithm without shifts, i.e. $\sigma_k = 0$. However then convergence will be slow whenever the eigenvalues of $\mathbf{A}$ are closely spaced in magnitude. To speed up convergence we shift the eigenvalues of $\mathbf{A}^{(k)}$ by subtracting a multiple of the identity matrix, $\sigma_k \mathbf{I}$. This shifts all eigenvalues by a constant σ_k, and is similar to the technique used with the shifted inverse power method. The rate at which $a_i^{(k)}$ converges towards λ_i can become significantly faster if σ_k is sufficiently close to λ_i.

Initially the point of these shifts is to try to force the $a_n^{(k)}$ to converge fast towards λ_n. This is done by trying to force the off diagonal element $b_{n-1}^{(k)}$ to tend towards 0 faster than $b_j^{(k)}$ for any $j < n-1$. A simple obvious choice is then $\sigma_k = a_n^{(k)}$. However a generally better choice turns out to be to set σ_k equal to the eigenvalue of the matrix

$$\begin{bmatrix} a_{n-1}^{(k)} & b_{n-1}^{(k)} \\ b_{n-1}^{(k)} & a_n^{(k)} \end{bmatrix}$$

closest to $a_n^{(k)}$. These eigenvalues are easily computed directly using the well known formula for the roots of a quadratic polynomial. We then iterate until $|b_{n-1}^{(k)}| < \epsilon$ where ϵ is some error tolerance. Then we return $a_n^{(k)}$ as an approximation of λ_n.

At this point we no longer have to include the last row and column of $A^{(k)}$ in the computations, and we therefore work with the resulting $(n-1)\times(n-1)$ matrix, which is again symmetric and tridiagonal, in subsequent iterations. We now use shifts equal to the eigenvalue of the matrix

$$\begin{bmatrix} a_{n-2}^{(k)} & b_{n-2}^{(k)} \\ b_{n-2}^{(k)} & a_{n-1}^{(k)} \end{bmatrix}$$

and continue to iterate until $|b_{n-2}^{(k)}| < \epsilon$. Then we return $a_{n-1}^{(k)}$ as an approximation of λ_{n-1}.

Again the last row and column is no longer necessary, and we proceed with the calculations on the $(n-2) \times (n-2)$ matrix consisting of the first $n-2$ rows and columns of $\mathbf{A}^{(k)}$.

In general when looking for λ_i, $i = n, n-1, \ldots, 2$ we work with the $i \times i$ matrix consisting of the first i rows and columns of $\mathbf{A}^{(k)}$. We use the shift

σ_k equal to the eigenvalue of

$$\begin{bmatrix} a_{i-1}^{(k)} & b_{i-1}^{(k)} \\ b_{i-1}^{(k)} & a_i^{(k)} \end{bmatrix}$$

and return $a_i^{(k)}$ as the approximation to λ_i when $|b_{i-1}^{(k)}| < \epsilon$. Once $|b_1^{(k)}| < \epsilon$ we also return $a_1^{(k)}$ in addition to $a_2^{(k)}$, since the eigenvalue of a 1×1 matrix is trivial.

We note in passing that once λ_{i+1} has been computed, doing orthogonal similarity transforms with the $i \times i$ matrix $\tilde{\mathbf{Q}}$ on the first i rows and columns is equivalent to doing orthogonal similarity transforms on the full $n \times n$ matrix with the orthogonal matrix

$$\mathbf{Q} = \begin{bmatrix} \tilde{\mathbf{Q}} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \in \mathbb{R}^{n \times n}.$$

This justifies only working with part of the original matrix in the QR-algorithm.

A final note regarding implementation. As mentioned we should technically compute $\mathbf{A}^{(k+1)} = \mathbf{R}^{(k)}\mathbf{Q}^{(k)} + \sigma_k \mathbf{I}$. However, in practice we don't usually add $\sigma_k \mathbf{I}$. Instead we keep track of the accumulated shifts σ_k in a variable $S_k = \sum_{j=1}^{k} \sigma_j$. We then return $a_i^{(k)} + S_k$ rather than $a_i^{(k)}$ as the approximation to λ_i once convergence has been achieved, i.e. $|b_{i-1}^{(k)}| < \epsilon$.

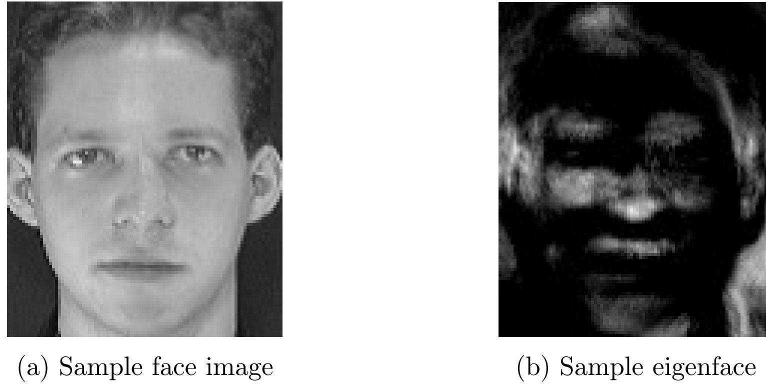
B Eigenfaces and principal component analysis

Automated face recognition have many interesting applications, e.g. criminal identification, security systems and general human-computer interaction. We here present a simple method for identifying a set of face images based on *eigenfaces* and *principal component analysis.* [4]

We assume we are given a number of grey scale face images, of the same size, of a group of p individuals. Call this the *training set*, and denote it's size by m. Our problem can be described as follows: Given another set of face images, termed the *test set*, of the same group of individuals, classify these images according to the person they are depicting. For simplicity we don't allow for images of something other than a face or an unknown face, i.e. a face which does not belong to a person in our group of individuals.

A normalized grey scale image can be considered as an $n_1 \times n_2$ matrix of intensity values between 0 and 1, where 0 equals black and 1 white. We shall however in the following also treat the face images equivalently as size

[4] The material here is based on the article *Eigenfaces for Recognition* by Matthew Turk and Alex Pentland



(a) Sample face image (b) Sample eigenface

Figure 2

$n = n_1 n_2$ column vectors, formed by placing all the columns of the matrix in a vector.

Images of faces, then amounts to points in $\mathbb{R}^n$. However since images of faces are similar, these points will not be randomly distributed in space. The main idea of principal component analysis is to attempt to find the vectors that best accounts for the distribution of face images within the entire space of images. These vectors define the subspace of face images, which we call *face space*. These vectors $\mathbf{u}_k$ turn out to be the eigenvectors of the covariance matrix $\mathbf{C}$ for the original face images. Since they themselves resemble ghostly faces, when considered as grey scale images, we call them eigenfaces. Below we detail how to find the eigenfaces for the training set. A sample face image and eigenface is shown in Figure 2

B.1 Computing the eigenfaces

Let the training set of images be $\mathbf{\Gamma}_1, \mathbf{\Gamma}_2, \ldots, \mathbf{\Gamma}_m \in \mathbb{R}^n$. The average face of the set is defined as

$$\mathbf{\Psi} = \frac{1}{m} \sum_{l=1}^{m} \mathbf{\Gamma}_l. \tag{11}$$

Each face then differs from the average by

$$\mathbf{\Phi}_l = \mathbf{\Gamma}_l - \mathbf{\Psi}, \quad l = 1, 2, \ldots, m. \tag{12}$$

From principal component analysis, we now seek the set of m orthonormal vectors $\mathbf{u}_k \in \mathbb{R}^n$ for $1 \leq k \leq m$, which best describes the distribution of the data. This is done by choosing $\mathbf{u}_k$ such that the quantity

$$\lambda_k = \frac{1}{m} \sum_{l=1}^{m} (\mathbf{u}_k^T \mathbf{\Phi}_l)^2.$$

is maximized subject to the orthonormality constraint

$$\mathbf{u}_k^T \mathbf{u}_l = \delta_{lk} = \begin{cases} 1, & \text{if } \ l = k \\ 0 & \text{otherwise} \end{cases} \tag{13}$$

$\mathbf{u}_k$ and λ_k are the eigenvectors and eigenvalues respectively, of the covariance matrix

$$\mathbf{C} = \frac{1}{m} \sum_{l=1}^{m} \mathbf{\Phi}_l \mathbf{\Phi}_l^T = \mathbf{A}\mathbf{A}^T \in \mathbb{R}^{n \times n}$$

where $A = \frac{1}{\sqrt{m}}[\mathbf{\Phi}_1, \mathbf{\Phi}_2, \ldots, \mathbf{\Phi}_m]$. However, since n equals the number of pixels in the images, determining all the n eigenvectors and eigenvalues of $\mathbf{C}$ is not a very attractive proposition, even for modest image sizes. Fortunately, this is not required. Usually the number of images in the training set $m \ll n$, and then there will only be $m-1$ rather than n meaningful, i.e. not associated with a zero eigenvalue, eigenvectors.

Furthermore, instead of solving for the n dimensional eigenvectors, $\mathbf{u}_k$, of $\mathbf{C}$ directly, we can first solve for the m eigenvalues, μ_i and associated eigenvectors, $\mathbf{v}_i$, $i = 1, 2, \ldots, m$, of the $m \times m$ matrix $\mathbf{L} = \mathbf{A}^T\mathbf{A}$. To see why, we start with the eigenvalue equation for $\mathbf{L}$

$$\mathbf{A}^T\mathbf{A}\mathbf{v}_i = \mu_i \mathbf{v}_i.$$

By multiplying on both sides with $\mathbf{A}$ we observe that

$$\mathbf{A}\mathbf{A}^T\mathbf{A}\mathbf{v}_i = \mu_i \mathbf{A}\mathbf{v}_i.$$

From this we see that $\tilde{\mathbf{u}}_i = \mathbf{A}\mathbf{v}_i$ is an eigenvector of $\mathbf{C} = \mathbf{A}\mathbf{A}^T$ with equal eigenvalue. Furthermore, since $\mathbf{L}$ has rank $m - 1$ (under the reasonable assumption that all the images in the training set are unique), the $m - 1$ eigenvectors associated with a non-zero eigenvalue are exactly the $m - 1$ meaningful eigenvectors we are looking for. The computation of $\mathbf{u}_k$ and λ_k can thus be done as follows:

1. Compute the m eigenvalues, μ_i and associated eigenvectors, $\mathbf{v}_i$, $i = 1, 2, \ldots, m$, of the $m \times m$ matrix $\mathbf{L} = \mathbf{A}^T\mathbf{A}$.

2. Discard the 0 eigenvalue and associated eigenvector of $\mathbf{L}$. Transform the remaining $m - 1$ eigenvalues and associated eigenvectors of $\mathbf{L}$ into the $m - 1$ nonzero eigenvalues, $\lambda_i = \mu_i$ and associated eigenvectors, $\tilde{\mathbf{u}}_i = \mathbf{A}\mathbf{v}_i$ of the $n \times n$ matrix $\mathbf{C} = \mathbf{A}\mathbf{A}^T$.

3. Normalize the set of eigenvectors $\tilde{\mathbf{u}}_i$ for $i = 1, 2, \ldots, m-1$, to get the set of eigenvectors $\mathbf{u}_i$ for $i = 1, 2, \ldots, m$ that satisfies the orthonormality constraint (13). These $\mathbf{u}_i$ are the eigenfaces.

B.2 Face recognition using eigenfaces

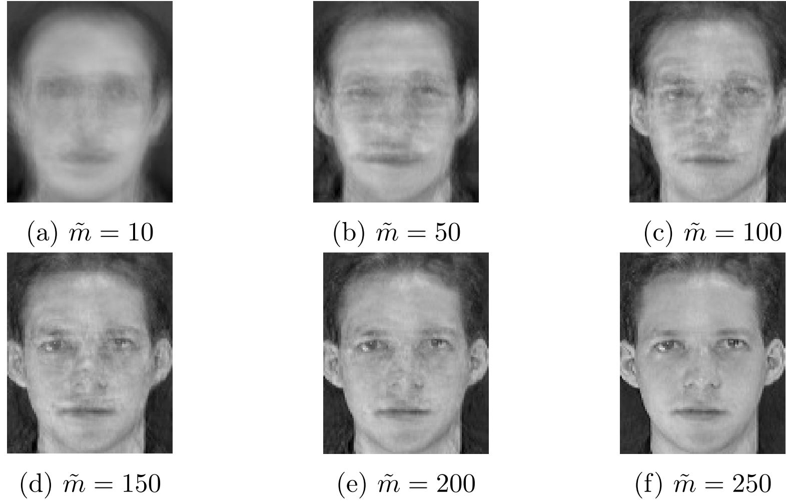
The eigenfaces $\mathbf{u}_i$, $i = 1, 2, \ldots, m-1$, span a basis set which can be used to describe and identify face images. The associated eigenvalues lets us rank the eigenvectors according to their usefulness in accounting for the variation among the images. Using all $m-1$ eigenfaces is not necessary, so we pick the $\tilde{m} < m$ eigenfaces associated with the largest (in absolute value) eigenvalues. The $\tilde{m}$-dimensional vector space with this basis is called the *face space*. The central process now is to transform a face image $\mathbf{\Gamma}$ into its eigenface components, by a simple projection operation into face space

$$\omega_k = \mathbf{u}_k^T(\mathbf{\Gamma} - \mathbf{\Psi}) \quad \text{for} \quad k = 1, 2, \ldots, \tilde{m} \tag{14}$$

These weights $\mathbf{\Omega} = [\omega_1, \omega_2, \ldots, \omega_{\tilde{m}}]^T$ describes the contribution of each eigenface in describing how the input face image differs from the mean face $\mathbf{\Psi}$. Figure 3 shows the approximate representation of a face as the weighted sum

$$\mathbf{\Gamma} \approx \mathbf{\Psi} + \sum_{k=1}^{\tilde{m}} \omega_k \mathbf{u}_k$$

of eigenfaces for various values of $\tilde{m}$.



(a) $\tilde{m} = 10$ (b) $\tilde{m} = 50$ (c) $\tilde{m} = 100$
(d) $\tilde{m} = 150$ (e) $\tilde{m} = 200$ (f) $\tilde{m} = 250$

Figure 3: Representation of a sample image in the training set using the face space representation for various values of $\tilde{m}$. m is here 280

The classification process can now be described as follows

1. For each person j in the group $j = 1, 2, \ldots, p$, compute the vector of weights $\mathbf{\Omega}^{(jl)} = [\omega_1^{(jl)}, \omega_2^{(jl)}, \ldots, \omega_{\tilde{m}}^{(jl)}]^T$ for each face image $\mathbf{\Phi}_{jl}$ $l = 1, 2, \ldots, p_j$ of that individual by the projection into face space, i.e.

$$\omega_k^{(jl)} = \mathbf{u}_k^T[\mathbf{\Phi}_{jl} - \mathbf{\Psi}]$$

for $k = 1, 2, \dots, \tilde{m}$ and $l = 1, 2, \dots, p_j$. Compute from this the class vector, $\mathbf{\Omega}^{(j)}$, by averaging

$$\mathbf{\Omega}^{(j)} = \frac{1}{p_j} \sum_{s=1}^{p_j} \mathbf{\Omega}^{(jl)}.$$

2. For a given unknown face image $\mathbf{\Gamma}$ in the test set, compute the weights vector $\mathbf{\Omega}$ using (14). Then identify $\mathbf{\Gamma}$ with the person j that minimizes the Euclidean distance to the associated class vector

$$\epsilon_j = \|\mathbf{\Omega} - \mathbf{\Omega}^{(j)}\|_2 \tag{15}$$