Principal Component Analysis (PCA)

In many areas and across multiple disciplines, large and especially high-dimensional data sets are used that often need to be analysed or visualised. Since it is hard to visualise data that goes beyond 3D, this often requires what is known as "dimensional reduction" - a process of reducing the dimension of the data while minimising the loss of information. One of the arguably most simple and widely used dimensional reduction algorithms is Principal Component Analysis (PCA for short). Principal component analysis is used in lots of applications including data compression, feature extraction or data visualisation [1].

This project intuitively explains PCA using a simple 2D example: a smiley face. An interactive tool has been created for this purpose, which visualises the example. The central concepts of principal component analysis are explained in this Jupyter notebook, where textual explanations and sections of Python code alternate.
In order to execute the programme code without errors, each section has to be run in sequence. The best way to do this is to click on **"Run all"** (Ctrl + F9) under "Runtime" in the menu in the top bar.
Now we can begin.

At first, we need some imports that are not conceptually relevant.

Imports

[53] Show code

Note: In graphical image processing, the x-axis in two-dimensional coordiante systems points to the right and the y-axis points downwards - instead of upwards (as in maths). This means that the origin of the coordinate system is in the top left-hand corner of a generated image, which usually only has positive values.

A smiley is defined here in simplified form by $p = 9$ points. As each two-dimensional point consists of x and y coordinates, $d = 9 * 2 = 18$ values are required to describe a smiley, which is therefore represented by an 18-dimensional vector.
Two consecutive values denote the x and y coordinates of a point. As we will see later, it is crucial to maintain a consistent order of points across multiple smileys, so in our example the first two points always need to represent the eyes of a smiley, whereas the next four points must always describe a smiley's mouth and the last two its nose. In other words, our smileys need to be in correspondence.
For example, the vector for a happy smiley looks like this:

```
[54] happy = np.array([5.5, 1.75, 2.5, 1.75, 2, 5, 3, 5.75, 4, 6, 5, 5.75, 6, 5, 4, 3, 4, 4])
```

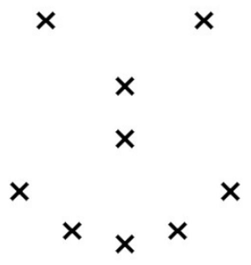
$$\vec{v_1} = \begin{pmatrix} 5.5 \\ 1.75 \\ 2.5 \\ 1.75 \\ 2 \\ 5 \\ 3 \\ 5.75 \\ 4 \\ 6 \\ 5 \\ 5.75 \\ 6 \\ 5 \\ 4 \\ 3 \\ 4 \\ 4 \end{pmatrix}$$

Note: In the programme code, row vectors are used instead of column vectors, which means that these and matrices used later must be transposed in some cases. Therefore, the implementation of the formulas may differ from the textual explanations.

We can now also draw these points from v_1 that define our happy smiley:

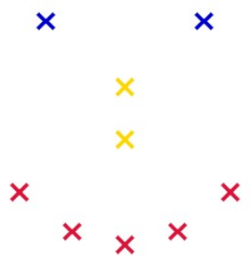
Draw the smiley pointwise

[55] Show code



If we colour the eyes, nose and mouth differently, the smiley can be recognised more easily:

```
[56] draw_smiley_pointwise(happy, "mediumblue", "gold", "crimson")
```



The points of the nose and mouth can now be connected to complete the smiley. The eyes, for example, are also displayed as circles to embellish the result. This turns a vector into a smiley.
The following method **`draw_smiley`** draws a finished smiley from an 18-dimensional vector.

```
[57] # Render Smiley

     def draw_smiley(p: np.array):

         p = copy.copy(p) * 50

         # Draw white background
         draw.rectangle([(0, 0), (400, 400)], "white")

         # Draw eyes
         def draw_ellipse_around_point(draw, center, width=21, fill="black"):
             coords = np.concatenate((center - width, center + width), axis=-1)
             draw.ellipse([(coords[0], coords[1]), (coords[2], coords[3])], fill=fill)

         right_eye_center, left_eye_center = p[0:2], p[2:4]
         draw_ellipse_around_point(draw, left_eye_center, fill="mediumblue")
         draw_ellipse_around_point(draw, right_eye_center, fill="mediumblue")

         # Draw nose
         nose_top, nose_bottom = p[14:16], p[16:18]
         draw.line([tuple(nose_top), tuple(nose_bottom)], fill="gold", width=21)
         draw_ellipse_around_point(draw, nose_top, width=10, fill="gold")
         draw_ellipse_around_point(draw, nose_bottom, width=10, fill="gold")

         # Draw mouth
         mouth = [p[4:6], p[6:8], p[8:10], p[10:12], p[12:14]]
         mouth_tuple = [tuple(x) for x in mouth]
         draw.line(mouth_tuple, fill="crimson", width=21, joint="curve")
         draw_ellipse_around_point(draw, mouth[0], width=10, fill="crimson")
         draw_ellipse_around_point(draw, mouth[4], width=10, fill="crimson")

         display.display(im)


     draw_smiley(happy)
```

A single smiley ($\hat{=}$ d-dimensional vector) forms a data point for our database. The database consists of n data points, with $n = 6$ for us.
Our database therefore consists of six different faces, which (apart from the happy smiley already created above) are defined in the next code section. Each smiley vector is structured in the same order as described above.

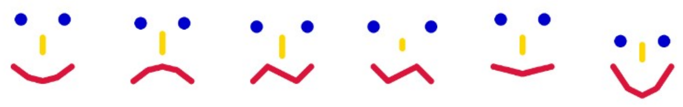
```
[58] # Database

     # happy = np.array([5.5, 1.75, 2.5, 1.75, 2, 5, 3, 5.75, 4, 6, 5, 5.75, 6, 5, 4, 3, 4, 4])
     sad = np.array([5.5, 2, 2.5, 2, 2, 6, 3, 5.25, 4, 5, 5, 5.25, 6, 6, 4, 2.75, 4, 4])
     drunken = np.array([5.75, 2.25, 2.25, 2.25, 2, 6, 3, 5, 4, 5.5, 5, 6, 6, 5, 4, 3, 4, 4.25])
     LSD = np.array([6, 2.25, 2, 2.25, 2, 5, 3, 6, 4, 5.5, 5, 5, 6, 6, 4, 3.25, 4, 3.75])
     neutral = np.array([5.5, 1.75, 2.5, 1.75, 2, 5, 3, 5.25, 4, 5.5, 5, 5.25, 6, 5, 4, 3, 4, 4])
     crazy = np.array([5.5, 3.25, 2.5, 3.25, 2, 5, 3, 6.5, 4, 7, 5, 6.5, 6, 5, 4, 3.5, 4, 4.5])


     # will be needed later:
     horizontalCoord = (happy[5] + happy[9]) / 2
     unenthusiasticIndividual = np.array([5.7, 1.25, 2.3, 1.25, 2, horizontalCoord, 3, horizontalCoord, 4, horizontalCoord, 5, horizontalCoord, 6, horizontalCoord, 4, 2.85, 4, 3.95])
     unenthusiasticPoplulation = np.array([5.5, 1.75, 2.5, 1.75, 2, horizontalCoord, 3, horizontalCoord, 4, horizontalCoord, 5, horizontalCoord, 6, horizontalCoord, 4, 3, 4, 4])
```

```
    smileIndividualVector = happy - unenthusiasticIndividual
    smilePopulationVector = happy - unenthusiasticPoplulation
```

Our database is made up of these six smileys:

Load the database images

[59] Show code



The data matrix X is a $d \times n$ matrix, where the vectors of the n data points form the columns of the data matrix:

$$X = \begin{pmatrix} \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ v_1 & v_2 & v_3 & v_4 & v_5 & v_6 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

```
[60] X_T = np.array([happy, sad, drunken, LSD, neutral, crazy])
```

The principal component analysis is performed (in contrast to the related singular value decomposition (SVD)[1,2]) on mean-subtracted data.
For this purpose, the arithmetic mean μ of the data points v_i is calculated as follows:

$$\mu := \frac{1}{n}\sum_{i=1}^{n} v_i$$

In the programme code, the mean value μ is denoted by **`mean`** or as a line vector by **`mean_T`**.

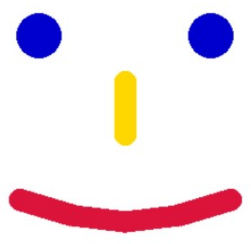
```
[61] mean_T = np.mean(X_T, axis=0)
```

The mean vector μ of our smiley database has the following values:

$$\vec{\mu} = \begin{pmatrix} 5.625 \\ 2.20833333 \\ 2.375 \\ 2.20833333 \\ 2 \\ 5.33333333 \\ 3 \\ 5.625 \\ 4 \\ 5.75 \\ 5 \\ 5.625 \\ 6 \\ 5.33333333 \\ 4 \\ 3.08333333 \\ 4 \\ 4.08333333 \end{pmatrix}$$

The average smiley defined by μ can also be visualised accordingly:

```
[62] draw_smiley(mean_T)
```



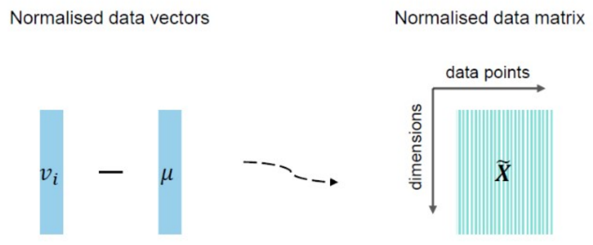
This mean value then has to be subtracted from all data points v_i in order to obtain normalised copies $v_i' := v_i - \mu$. These result in the normalised data matrix $\tilde{X} \in \mathbb{R}^{d \times n}$:

$$\tilde{X} = \begin{pmatrix} \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ v_1' & v_2' & v_3' & v_4' & v_5' & v_6' \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

In pictures

[63] Show code

Show code



The overall idea of principal component analysis is to project the high-dimensional data points v_i, onto a low-dimensional subspace with basis $\{w_j \,|\, j = 1, \ldots, m\}$, where $m < k := min(n, d)$. This new basis should maximise the variance in order to express as much of the span of the original data points as possible. Since the pointwise scaling of w_j also scales the variance, we impose the additional condition that

$$\|w_j\|_2 = 1$$

for all j.

For this purpose, we define the covariance matrix $S \in \mathbb{R}^{d \times d}$:

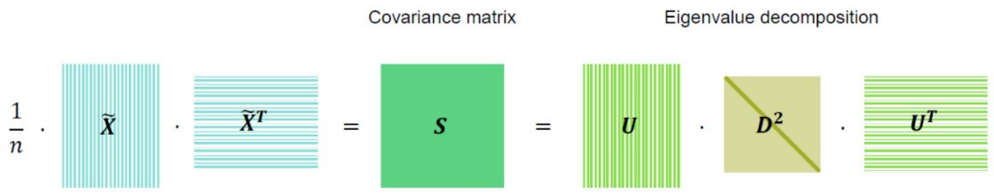
$$S := \frac{1}{n}\tilde{X}\tilde{X}^T$$

Subsequently, the eigenvalues λ_i of S and their corresponding eigenvectors w_i must be calculated. Now only the m eigenvectors $w_1, \ldots, w_m$ of the largest eigenvalues are taken [1]. These m so-called principal components form the new basis we are looking for.

Note: The number m can be chosen arbitrarily. However, since the principal component analysis provides a maximum of $k - 1$ meaningful eigenvectors (i.e. eigenvectors for eigenvalues $\lambda_i \neq 0$), one should choose $m \in \{1, \ldots, k-1\}$ [3]. The smaller the choice of m, the smaller the target dimension, but the greater the loss of information when converting the data from the original data space to the so-called PCA space. The value of an eigenvalue describes the variance that is represented by the associated principal component.

> In pictures

[64] Show code



The principal component analysis is performed in the following programme code, whereby in our example $m = 5$ (in the code **`M_COMPONENTS`**) is selected. The eigenvectors of the basis found are stored in **`components`**.

[65]
```python
# PCA

M_COMPONENTS = 5

PCA = sklearn.decomposition.PCA(n_components=M_COMPONENTS)
PCA.fit(X_T)
components = PCA.components_

stdDeviations = [np.sqrt(PCA.explained_variance_[0]), np.sqrt(PCA.explained_variance_[1]), np.sqrt(PCA.explained_variance_[2]),
                 np.sqrt(PCA.explained_variance_[3]), np.sqrt(PCA.explained_variance_[4])]
```

Now we have the required basis, consisting of the $m = 5$ normed basis vectors w_1, w_2, w_3, w_4 and w_5:

$$\vec{w_1} = \begin{pmatrix} -0.02761514 \\ 0.3934326 \\ 0.02761514 \\ 0.3934326 \\ 0. \\ -0.18074909 \\ 0. \\ 0.36466676 \\ 0. \\ 0.53884804 \\ 0. \\ 0.3728013 \\ 0. \\ 0.18888363 \\ 0. \\ 0.18286673 \\ 0. \\ 0.1574453 \end{pmatrix}, \quad \vec{w_2} = \begin{pmatrix} 0.15837053 \\ 0.05953233 \\ -0.15837053 \\ 0.05953233 \\ 0. \\ -0.42086492 \\ 0. \\ 0.47262055 \\ 0. \\ -0.01420654 \\ 0. \\ -0.4584762 \\ 0. \\ 0.51023183 \\ 0. \\ 0.1353312 \\ 0. \\ -0.23064818 \end{pmatrix}, \quad \vec{w_3} = \begin{pmatrix} -0.11039057 \\ -0.41100849 \\ 0.11039057 \\ -0.41100849 \\ 0. \\ -0.58895954 \\ 0. \\ 0.0654281 \\ 0. \\ 0.24256792 \\ 0. \\ -0.05694117 \\ 0. \\ -0.46659028 \\ 0. \\ 0.01232272 \\ 0. \\ -0.08171833 \end{pmatrix},$$

$$\begin{pmatrix} -0.56129356 \\ -0.1103873 \\ 0.56129356 \\ -0.1103873 \\ 0. \\ 0.12942928 \\ 0. \\ 0.29329039 \\ 0. \end{pmatrix} \quad \begin{pmatrix} -0.30818905 \\ 0.33259592 \\ 0.30818905 \\ 0.33259592 \\ 0. \\ -0.30465606 \\ 0. \\ -0.36574707 \\ 0. \end{pmatrix}$$

$$\vec{w_4} = \begin{pmatrix} 0. \\ 0.11717182 \\ 0. \\ 0.05174094 \\ 0. \\ 0.37097873 \\ 0. \\ -0.28716583 \\ 0. \\ 0.07912895 \end{pmatrix}, \quad \vec{w_5} = \begin{pmatrix} 0. \\ -0.18851046 \\ 0. \\ -0.49185763 \\ 0. \\ -0.17854551 \\ 0. \\ 0.09917843 \\ 0. \\ 0.20745535 \end{pmatrix}$$

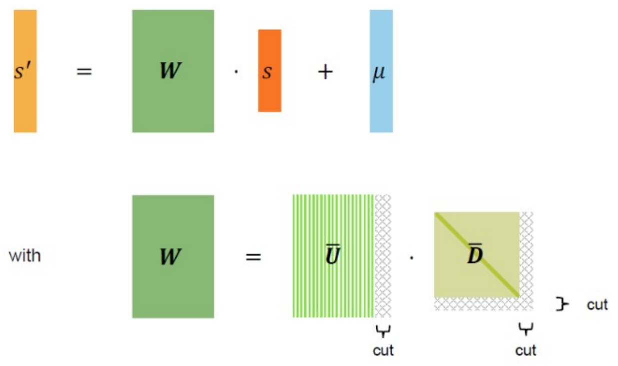
Hence, any smiley can be specified by a 5-dimensional vector s (as a linear combination of the new basis vectors) instead of an 18-dimensional vector.

In order to transform the 5-dimensional vector s from our so-called PCA space back into an 18-dimensional vector s' from our so-called data space, which we can visualise as a smiley using the **`draw_smiley`** method, a transformation matrix W is required. We obtain this matrix by multiplying each basis vector w_j by its standard deviation, i.e. the square root of its eigenvalue $\sqrt{\lambda_j}$. The resulting vectors $\tilde{w_j}$ are then composed column by column to form the matrix W.
For the conversion from PCA space to data space, the new matrix W is multiplied by the 5-dimensional smiley vector s and then the vector of the average face μ is added [4].

$$s' = W \cdot s + \mu \quad , \qquad \text{with } W = \underbrace{\begin{pmatrix} \vdots & \vdots & \vdots & \vdots & \vdots \\ \tilde{w_1} & \tilde{w_2} & \tilde{w_3} & \tilde{w_4} & \tilde{w_5} \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix}}_{\forall j \in \{1, \ldots, 5\}}, \qquad \text{where } \tilde{w_j} := \sqrt{\lambda_j} \cdot \vec{w_j}$$

> In pictures

[66] Show code



This is implemented by the **`render_from_params`** method, which then draws the smiley.

[67]
```
# From 5 to 18 dimensions

W_T = np.array([components[0] * stdDeviations[0], components[1] * stdDeviations[1], components[2] * stdDeviations[2],
                components[3] * stdDeviations[3], components[4] * stdDeviations[4]])

def render_from_params(v_T: np.array, sf1=0.00, sf2=0.00):

    p = copy.copy(mean_T)
    p += np.matmul(v_T, W_T)

    # will be needed later:
    p += sf1 * smileIndividualVector
    p += sf2 * smilePopulationVector

    draw_smiley(p)
```

The vector

$$\vec{s} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

thus represents the average smiley in our new basis.

The linear combination with arbitrary parameters can also be tested in our interactive tool. The parameters can be changed manually using the 5 sliders and the resulting smiley is displayed depending on this using the calculation just explained. It is easy to see that the first principal component (≘ 1st slider) causes the most change to the smiley when it is modified, i.e. it represents the greatest variance, and this decreases from principal component to principal component.
Using the "Random face" button, a random vector s and thus a random smiley can be generated. The random values for the vector s are normally distributed. By clicking on the "Reset face" button, all vector entries of s are reset to 0 and thus the average smiley is displayed.
For example, we start with the smiley for the vector

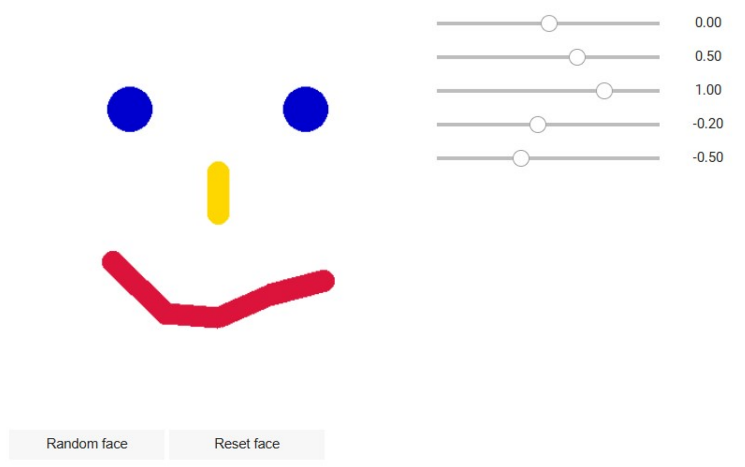
$$\vec{s} = \begin{pmatrix} 0 \\ 0.5 \\ 1 \\ -0.2 \\ -0.5 \end{pmatrix}$$

> Helpers

[68] Show code

Tool part 1

[69] Show code



Conversely, the transformation from data space to PCA space, i.e. from the representation in 18 to that in 5 dimensions, is also possible. If we have a given smiley ($\hat{=}$ 18-dimensional vector s') and want to obtain the linear combination of our new basis vectors that corresponds to the given smiley, the system of linear equations

$$W \cdot s = (s' - \mu)$$

needs to be solved.

This is achieved by the matrix multiplication

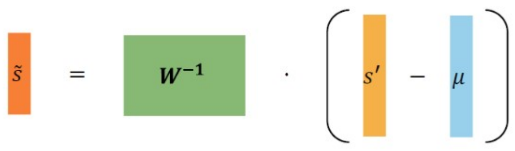
$$\tilde{s} = W^{-1} \cdot (s' - \mu)$$

$\tilde{s}$ denotes the resulting 5-dimensional vector in PCA space.

$Note:$ Since W is not a square matrix, W^{-1} designates the pseudo inverse of W.

In pictures

[70] Show code



The **getLinComb** method is responsible for this calculation:

[71]
```
# From 18 to 5 dimensions

def getLinComb(face: np.array):
  res_T = face - mean_T
  return np.matmul(res_T, np.linalg.pinv(W_T))
```

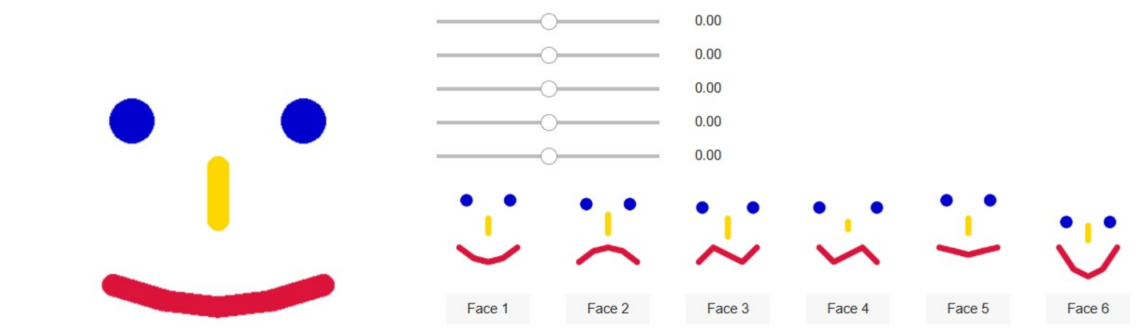
We can therefore describe already known smileys from the data space ($\hat{=}18$-dimensional vectors) in our new basis. For example, we already know the smileys in the database. Clicking on the button under the respective image of one of these smileys performs the calculation just explained. In addition to displaying the correct smiley, the parameters of the 5 base vectors are automatically adjusted to the corresponding values via the sliders (deactivated here).
This function can be tried out directly here:

Helpers

[72] Show code

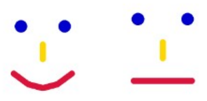
Tool part 2

[73] Show code



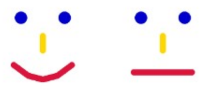
We can not only create faces, but also manipulate existing faces. A smile from our smiley is a good choice for this.
To do so, we first need to calculate a smile vector l_1 from a smiling and a non-smiling smiley by taking the difference between the two.
In our example, we take the difference between the following two smileys (already calculated above in the database as **`smileIndividualVector`**):

```
[74] # smileIndividualVector = happy - unenthusiasticIndividual

     smileIndividualVector_images = widgets.HBox([wiIm1, wiIm7])
     display.display(smileIndividualVector_images)
```



As you can conclude, l_1 changes the position of the eyes as well as the position and length of the nose in addition to the mouth. This is due to the fact that when calculating the smile vector l_1, we compare any two smileys, one of which is smiling and one of which is not. However, this does not correspond to the difference between the smile and non-smile of the "same" smiley.
If we consider the smileys as "real" (stylised) faces, we would not expect a change in the position of the eyes and nose when "the same" smiley smiles, but only a modification of the mouth. This result can also be obtained using the difference of smileys method if all smiling and non-smiling smileys in a representative data set are subtracted from each other and the average is calculated from the results.
Such an "individuality-adjusted" smile vector l_2 is represented in the interactive tool by the "Smile l_2" slider, which only modifies the mouth. To achieve this, we simply took the difference between the two following smileys (calculated as **`smilePopulationVector`** after the database above), which only differ in terms of the mouth.

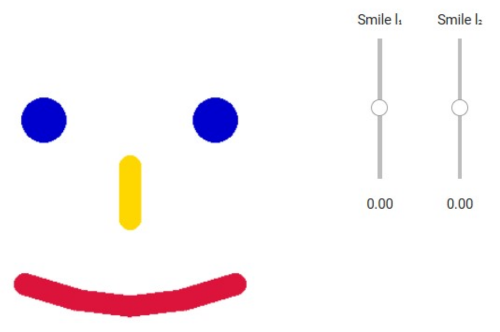
```
[75] # smilePopulationVector = happy - unenthusiasticPoplulation

     smilePopulationVector_images = widgets.HBox([wiIm1, wiIm8])
     display.display(smilePopulationVector_images)
```



This face modification can also be tested here by changing the corresponding "Smile"-sliders:

Tool part 3

[76] Show code



Finally, we come to the interactive tool where you can see the results of the principal component analysis and our calculations using the smiley. By default, the image shows the average smiley, but the sliders and buttons can be used to modify the smiley as required. All the functions presented so far are combined and can be tested here. Have fun with it!

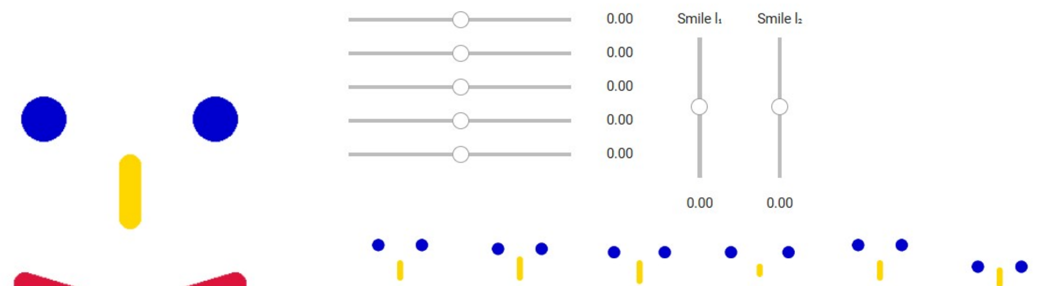
The following programme code is only responsible for the display and is conceptually irrelevant.

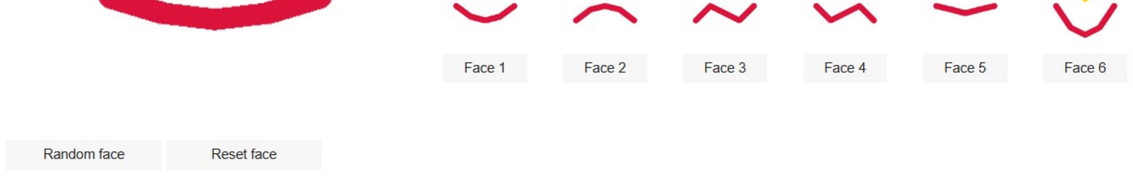
Helpers

[77] Show code

Main tool

[78] Show code





How such an example for principal component analysis can be transferred to more dimensions and more complex applications is explained in the following notebook on PCA on faces in 3D: https://colab.research.google.com/drive/16oMHRt5kj-X-2sBOE5ziw43CgdvJ6mm6?usp=sharing

Sources:

[1] Takio Kurita. “Principal Component Analysis (PCA)”. In: Computer Vision. Springer eBook Collection. Cham: Springer International Publishing and Springer, 2019, pp. 1–4. ISBN: 978-3-030-03243-2. DOI: 10.1007/978-3-030-03243-2_649-1.
[2] Kaare Brandt Petersen and Michael Syskind Pedersen. “The Matrix Cookbook”. In: Technical University of Denmark 7.15 (2012). URL: https://www.math.uwaterloo.ca/~hwolkowi/matrixcookbook.pdf.
[3] M. Turk and A. Pentland. “Eigenfaces for recognition”. In: Journal of cognitive neuroscience 3.1 (1991), pp. 71–86. ISSN: 0898-929X. DOI: 10.1162/jocn.1991.3.1.71.
[4] Thomas Albrecht et al. “Posterior shape models”. In: Medical Image Analysis 17.8 (2013), pp. 959–973. ISSN: 1361-8415. DOI: 10.1016/j.media.2013.05.010. URL: https://www.sciencedirect.com/science/article/pii/S1361841513000844.

Transfer of principal component analysis (PCA) from smileys (2D) to faces (3D)

The principal component analysis can be applied not only to smileys in a two-dimensional coordinate system (see https://colab.research.google.com/drive/1HVipazHa0WX8I1kVSUfW_ouQryTmugOJ?usp=sharing), but also to faces in a three-dimensional coordinate system. Completely different fields of application and examples are also possible. The transfer of the concepts of PCA demonstrated in the smiley example to other (more complex) applications is explained below using 3D face scans.

In order to execute the programme code in this project without errors, each section has to be run in sequence. The best way to do this is to click on **"Run all"** (Ctrl + F9) under "Runtime" in the menu in the top bar.
Now we can begin.

At first, we need some imports that are not conceptually relevant.

Imports

[11] Show code

This Jupyter notebook does not contain any interactive elements for you to try out by yourself, as the example with the faces has already been implemented elsewhere. The link to that website, where you can modify the 3D models of the faces, is as follows: https://maximilian-hahn.github.io/exploreCOSMOS/
Images and excerpts from the tool are provided here for comparison and explanation.

Load the images

[12] Show code

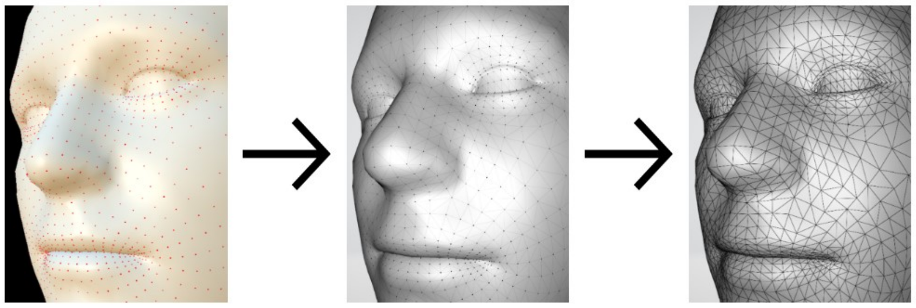
The 3D tool provides a face model by default, which is based on the Basel Face Model from 2019 [1] and also uses its average face. In contrast to the Basel Face Model, it does not use real face scans as a database, but random variations on the average face, which is why the generated faces do not necessarily look realistic. Conceptually, there is no difference between the two models though.
If you would like to work with the realistic faces based on the Basel Face Model, you can download it here and then upload the model to the 3D tool using the "Select file" button in the top left-hand corner, after which the principal component analysis will be performed on the new model data.

In contrast to our smileys, the modelled faces in the tool consist of $p = 1293$ (instead of $p = 9$) points. In addition, these points are now in a three-dimensional coordinate system, i.e. each point has an x, y and z coordinate. Therefore, $d = 1293 * 3 = 3879$ values are required to describe a face. A face can thus be represented by a 3879-dimensional vector. These orders of magnitude now also illustrate the motivation for dimensional reduction.
In the face vectors, each three consecutive values represent a point with x, y and z coordinates. Each of these points also denotes a specific point on a face (e.g. the right corner of the mouth), which is why the sequence of values in the face vector must always follow the same fixed pattern.

As with our smiley, the 1293 points of the face must now be suitably connected to form a face. In the three-dimensional face model, this is not done by connecting several specific points using lines, but most simply using triangles, which then generate surfaces in space. This means that each point and its surrounding points span triangles which, when put together, form the surface of the face. The corners and edges in the model that occur this way are then smoothed in the visualisation. The manner in which points are connected to other points by triangles is defined in advance.
This creation of the facial surface out of the points is illustrated in the following images.

› From points to the face

[13] Show code



The basis for the Basel Face Model is made up of $n = 200$ faces (of 100 men and 100 women), which now also form the data base for the principal component analysis.
The faces of the data base are of the same type as the following ten example faces without their colouring [1].

› Examples for face scans

[14] Show code

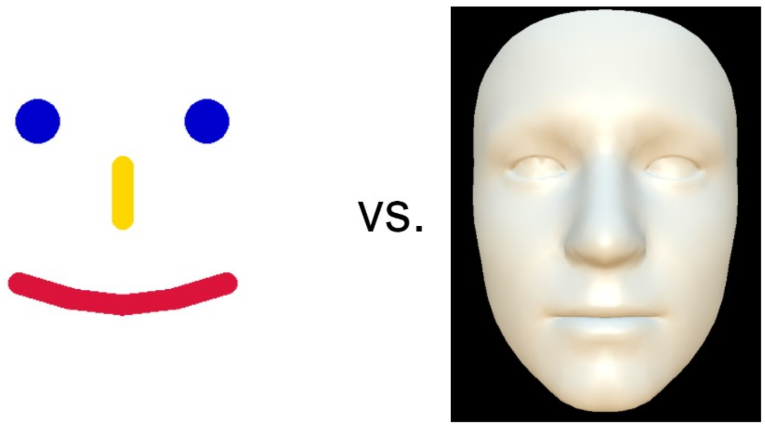
The data matrix X, which is composed of the 200 face vectors of the data base, is now a 3879×200 matrix and thus significantly larger than

in the case of the 2D-smileys.
However, apart from the graphical representation of the data points and this fact, there are no further differences between the two examples. In particular, the procedure for the principal component analysis and the application of the mathematical formulae in the individual steps of the PCA do not change.
Our knowledge about PCA gained from the smiley example can therefore be transferred to the face example and to all other use cases as long as the structure of the data matrix complies with the same specifications.

The principal component analysis is now performed as in the smiley example.
We first calculate the arithmetic mean μ of the data points in the data base and obtain the average face.
Below we see the comparison of the average smiley (2D) and the average face (3D):

Average smiley vs. average face

[15] Show code



The further steps of the PCA are now carried out analogously on the face data. This means:

- Determine normalised data matrix $\tilde{X} \in \mathbb{R}^{d \times n}$.
- Calculate covariance matrix $S := \frac{1}{n} \tilde{X} \tilde{X}^T$.
- Calculate eigenvalues λ_i of S.

Now we take the $m = 10$ eigenvectors $w_i, \ldots, w_{10}$ of the largest eigenvalues from the resulting 199. These yield our 10 principal components and thus our new basis.
The faces can now be described by 10 principal components, i.e. 10-dimensional vectors (instead of 3879-dimensional vectors).

The conversion of a face from the representation in PCA space (10-dimensional) to that in data space (3879-dimensional) and vice versa is also mathematically analogous to the smiley example. The only difference is our new basis matrix $W \in \mathbb{R}^{d \times m}$, which now consists of 10 principal components in the case of the faces:

$$W = \begin{pmatrix} \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \tilde{w_1} & \tilde{w_2} & \tilde{w_3} & \tilde{w_4} & \tilde{w_5} & \tilde{w_6} & \tilde{w_7} & \tilde{w_8} & \tilde{w_9} & \tilde{w_{10}} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix}, \qquad \text{where } \tilde{w_j} := \sqrt{\lambda_j} \cdot \vec{w_j} \qquad \forall j \in \{1, \ldots, 10\}$$

The vector

$$\vec{s} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

accordingly represents the average face with our new basis.

In order to transform the 10-dimensional vector s from the PCA space back into a 3879-dimensional vector s' from the data space, which can then be visualised as a face, we again need the following formula:

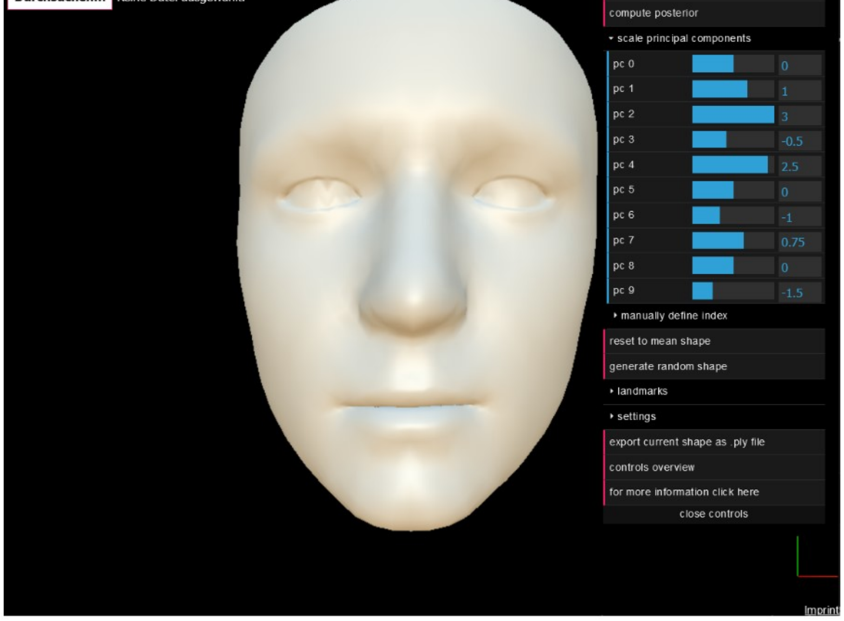
$$s' = W \cdot s + \mu$$

For example, we can generate the face for the vector

$$\vec{s} = \begin{pmatrix} 0 \\ 1 \\ 3 \\ -0.5 \\ 2.5 \\ 0 \\ -1 \\ 0.75 \\ 0 \\ -1.5 \end{pmatrix}$$

Show face in 3D tool

[16] Show code



pc 0 0
pc 1 1
pc 2 3
pc 3 -0.5
pc 4 2.5
pc 5 0
pc 6 -1
pc 7 0.75
pc 8 0
pc 9 -1.5

The influence of the individual principal components can also be tested manually in the 3D tool using the control elements on the right-hand side ("scale principal components"). Resetting to the average face ("reset to mean shape") and generating a normally distributed random face ("generate random shape") is also possible there.

A simple manipulation of the entire face as in the smiley example with the smile vectors would be just as easy for the 3D faces. However, this function, including these vectors, is not provided by the 3D tool, as its functionalities are general and designed to be used with different models. However, each of the 1293 points of the face can be moved individually. As this quickly leads to "edges" in the face and hence to an unrealistic face model, this is not enough to create a meaningful face.
To achieve that, the so-called posterior model of the face matching a few selected points can be calculated and displayed. All missing points that are not defined as known are completed in such a way that, together with the specified points, they result in a face that is as probable as possible.

The calculation of the posterior model uses the mathematical formula $\tilde{s} = W^{-1} \cdot (s' - \mu)$ to convert the representation from data space to PCA space.
Now, however, we have not given the entire data vector s', but only part of this data as vector s'_g. The missing data must therefore be added. In order to apply the formula, W now also has to be reduced to W_g as well as μ to μ_g. To do so, the lines or entries that correspond to the missing points are omitted [2].
The following example, in which two points ($(s_1|s_2|s_3)$ and $(s_4|s_5|s_6)$) are known, shows how this works in concrete terms.

$$\vec{s'} = \begin{pmatrix} \boxed{?} \\ s_1 \\ s_2 \\ s_3 \\ \boxed{?} \\ s_4 \\ s_5 \\ s_6 \\ \boxed{?} \end{pmatrix} \longrightarrow \vec{s'_g} = \begin{pmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ s_6 \end{pmatrix}, \qquad \vec{\mu} = \begin{pmatrix} \vdots \\ \mu_{s_1} \\ \mu_{s_2} \\ \mu_{s_3} \\ \vdots \\ \mu_{s_4} \\ \mu_{s_5} \\ \mu_{s_6} \\ \vdots \end{pmatrix} \longrightarrow \vec{\mu_g} = \begin{pmatrix} \mu_{s_1} \\ \mu_{s_2} \\ \mu_{s_3} \\ \mu_{s_4} \\ \mu_{s_5} \\ \mu_{s_6} \end{pmatrix}$$

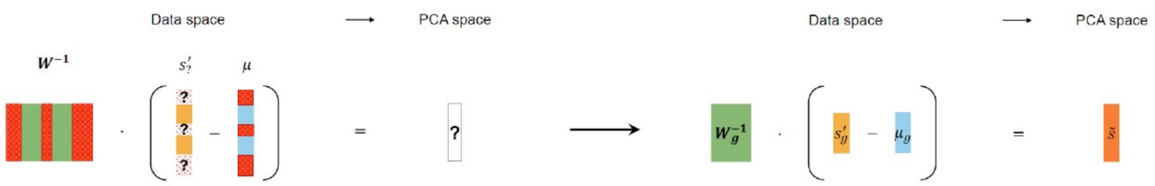
$$W = \begin{pmatrix} \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ w1_{s_1} & w2_{s_1} & w3_{s_1} & \cdots & w8_{s_1} & w9_{s_1} & w10_{s_1} \\ w1_{s_2} & w2_{s_2} & w3_{s_2} & \cdots & w8_{s_2} & w9_{s_2} & w10_{s_2} \\ w1_{s_3} & w2_{s_3} & w3_{s_3} & \cdots & w8_{s_3} & w9_{s_3} & w10_{s_3} \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ w1_{s_4} & w2_{s_4} & w3_{s_4} & \cdots & w8_{s_4} & w9_{s_4} & w10_{s_4} \\ w1_{s_5} & w2_{s_5} & w3_{s_5} & \cdots & w8_{s_5} & w9_{s_5} & w10_{s_5} \\ w1_{s_6} & w2_{s_6} & w3_{s_6} & \cdots & w8_{s_6} & w9_{s_6} & w10_{s_6} \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \end{pmatrix} \longrightarrow W_g = \begin{pmatrix} w1_{s_1} & w2_{s_1} & w3_{s_1} & \cdots & w8_{s_1} & w9_{s_1} & w10_{s_1} \\ w1_{s_2} & w2_{s_2} & w3_{s_2} & \cdots & w8_{s_2} & w9_{s_2} & w10_{s_2} \\ w1_{s_3} & w2_{s_3} & w3_{s_3} & \cdots & w8_{s_3} & w9_{s_3} & w10_{s_3} \\ w1_{s_4} & w2_{s_4} & w3_{s_4} & \cdots & w8_{s_4} & w9_{s_4} & w10_{s_4} \\ w1_{s_5} & w2_{s_5} & w3_{s_5} & \cdots & w8_{s_5} & w9_{s_5} & w10_{s_5} \\ w1_{s_6} & w2_{s_6} & w3_{s_6} & \cdots & w8_{s_6} & w9_{s_6} & w10_{s_6} \end{pmatrix}$$

With our modified data, the posterior model for the given points can now be computed as follows:

$$\tilde{s} = W_g^{-1} \cdot (s'_g - \mu_g)$$

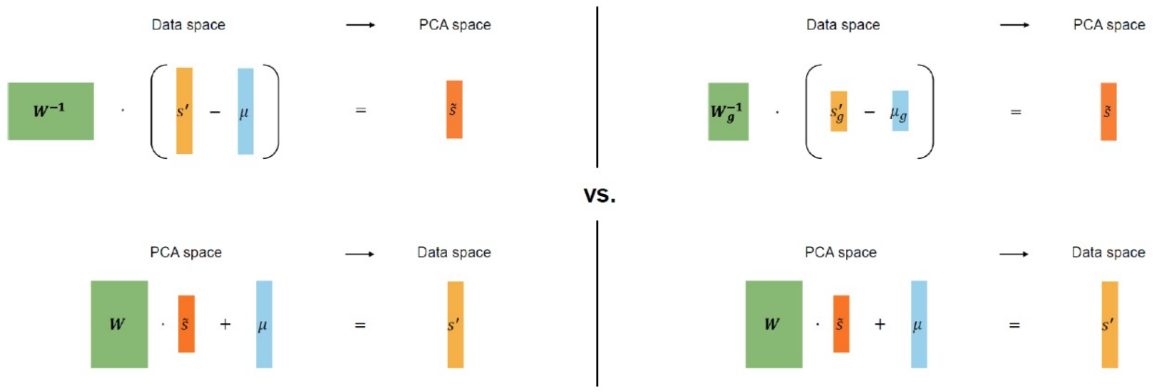
› In pictures: adjust the vectors and matrices for partial data vectors

[17] Show code



› Transformation workflow: complete data vectors vs. partial data vectors

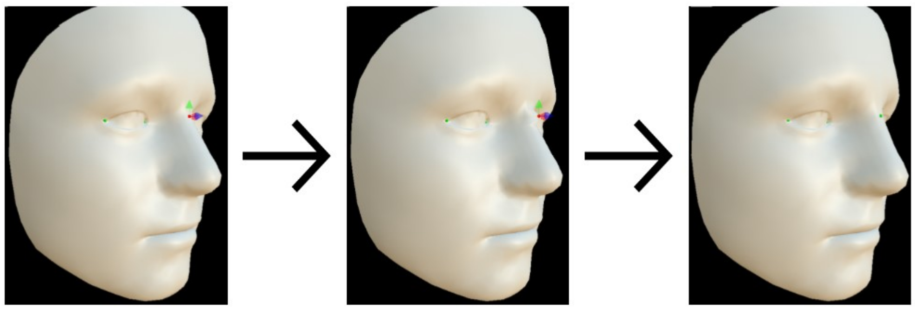
[18] Show code


Data space
PCA space
$W^{-1} \cdot (s' - \mu) = \tilde{s}$
Data space
PCA space
$W_g^{-1} \cdot (s'_g - \mu_g) = \tilde{s}$
vs.
PCA space
Data space
$W \cdot \tilde{s} + \mu = s'$
PCA space
Data space
$W \cdot \tilde{s} + \mu = s'$

For this realisation in the 3D tool, the selected points are manually designated as so-called landmarks and, if necessary, slightly modified according to the user's wishes. By clicking on the "compute posterior" button, the landmarks are now retained as points of the new face and the rest of the face is suitably added and then displayed. An example of modifying the bridge of the nose while retaining the position of the corners of the eyes can be found here.

Calculate posterior

[19] Show code



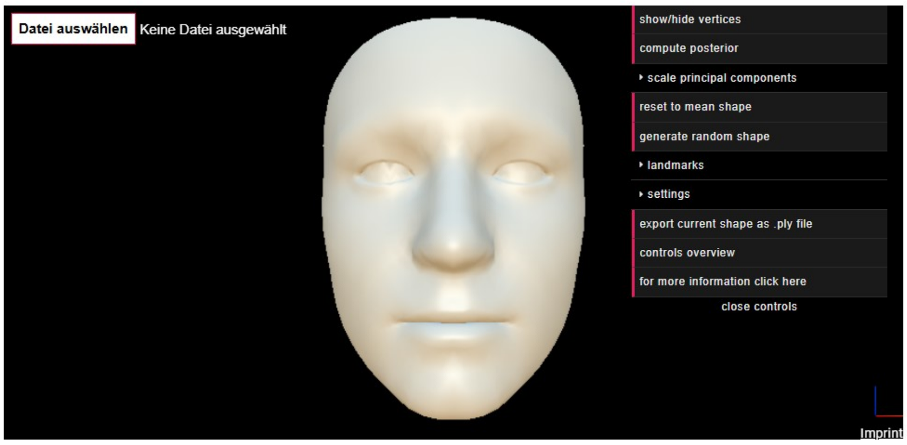
With this option of supplementing missing data using principal component analysis, another important function of PCA is covered and can be tested interactively.

The interactive 3D tool offers the functions described so far and a few more, such as different lighting and camera settings.
When using the programme for the first time, do not close the tutorial displayed by default, but read it carefully. If you encounter any problems with the operation, we recommend reading the GitHub repository [3] (linked under "for more information click here").

The best way to try out the tool is directly on the website: https://maximilian-hahn.github.io/exploreCOSMOS/ .
Alternatively, you can also test everything right here. Have fun with it!

3D tool

[20] Show code



Sources:

[1] University of Basel. "Universität Basel Morphface". Accessed: 2024-04-05. URL: https://faces.dmi.unibas.ch/bfm/
[2] Maximilian Hahn and Bernhard Egger. “Interactive Exploration of Conditional Statistical Shape Models in the Web-browser: exploreCOSMOS”. In: BVM Workshop, pp. 108–113. URL: https://arxiv.org/abs/2402.13131.
[3] Maximilian Hahn. "Interactive Creation and Modification of Statistical Shape Models". Accessed: 2024-04-05. URL: https://github.com/maximilian-hahn/BA.