Contents

2 Introduction

This assignment is build around a data-set of 550 images taken with a omnidirectional camera. Every image has fixed orientation in a room. Our goal is to determine the orientation of the camera by comparing it to a trainingset with known orientations.
In our problem we have to determine which image out of the trainingset is closest to a image of the testset. The images have a size of 150x112. The trainingdata-set is 300 images large and the testdata-set is 250 images large. Intuitively you can calculate the dot-product of one image with every one of the trainingset, but $150 \times 112 = 16800$ pixels and comparing one image with 300 in the trainingset is a very costly operation. We need a alternative. Fortunately PCA is very good option. Principal Components Analysis is a method to abstract data, which makes it a lot easier to compare data-points.
In this assignment we will look as this method, as well as combining it with a simple supervised learning approach: *Nearest Neighbour*. At last we will test the combination on our problem and show the results.

3 Theory

The main purposes of a principal component analysis are the analysis of data to identify patterns and finding patterns to reduce the dimensions of the data-set with minimal loss of information. Because of the large data-set we are using it would be computationally expensive to calculate. With principal component analysis we will be able to compare the test-set with the training set efficiently.
The idea behind PCA is that every data-point in a data-set is build up from a central point, by adding several features to it, which will result in the original data-point. Using PCA, you can calculate this central point and the needed features to abstract the data. The central point in PCA is the mean of the data-set, which means that in our problem, the central point is the mean of all the images. The features needed to add up to every individual image are split in two parts, vectors called the eigenvectors of the covariance-matrix and the vector containing the components of the image in *eigenspace*.

$$x_j = \bar{x} + \sum_{i=1}^{n} g_{ji} e_i \tag{1}$$

With x_j being the vectorised image, $\bar{x}$ the mean of the image data-set, g_j the vector of components and e the matrix of eigenvectors.
To reach this equation, some steps must precede. An grey-scale image is usually defined as a two-dimensional matrix of size M×N0. To calculate the mean of all the images in a data-set of size K, it is efficient to reshape the image from a matrix to a vector. With all the images reshaped a matrix X of size M×N by K can be formed. Taking the mean over the rows results in the mean of all the images in the data-set. From matrix X the covariance-matrix Q can be calculated using the equation:

$$Q = XX^T \tag{2}$$

From the Q the eigenvectors and eigenwaarden are calculated. All the eigenvectors are concatenated into matrix e. From this set of eigenvectors the vector of components of every image is calculated, by taking the dot-product of the normalized image(image with the mean image subtracted).

$$g_{ji} = e_i \cdot (x_j - \bar{x}) \tag{3}$$

With now every variable calculated, the image is now defined in a different way. The distance between images is now the distance between component-vectors. And since the first components are the most important(since their eigenwaarden was the largest), you can approximate the distance between images only by calculating the dot-product of the first components in the component-vector of each image. This is where PCA shows its worth.

Only if certain conditions are met, the sum of squared differences between pixels of two images reduces to a correlation based similarity measure. As Trucco states. (1) The two images need to have a normalized pixel intensities.(2) The images are normalized in size. (3) Each image should only contain one object. (4) The objects are imaged by a fixed camera under weak perspective.

All the images have a maximum pixel intensity of 1, because they represent grey values. Also all images have a equal size. The object in the image is the room itself. So there is only one environment and unoccluded by other objects.

By concatenating the pixels of the image, the correlation measure can become the the dot product, it is the dot because it calculates the distance between two vectors. In Matlab a matrix can be reshaped to a vector by using the reshape function. For instance reshaping matrix A to a column vector can be done by $reshape(A, [1, length(A)]$ When comparing the papers form Shlens and Truco it can be seen that Shlens uses the a transposed version of X relative to Truco. The so called straight forward generalization from Shlens can be explained when we look at the geometrical meaning. A covariance matrix C allows us to measure the covariance in a direction r as $r \cdot (Cr)$. This formula can be derived by taking the variance like $V_r = \frac{1}{N}\sum_{i=1}^{N}(s_i\mu_r)^2$ and substituting in the the scalar value s_i subsequently simplifying the equation results in the teh equation $r \cdot (Cr)$

SVD can intuitively be described as the composition of three operations, rotation scaling and again rotation. The SVD is decomposed of the unitary matrix U, the diagonal matrix $\sum$, and V^* is the conjungate transpose of the unitary matrix V.

4 Algorithm

We divided the program in multiple parts, since the algorithm also has clear steps to follow.

```
1  Load images into cell-array;
2
3  Divide cell-array in trainingset and testset
4
5  Calculate mean of image-vectors
6  Normalize trainingset using mean-image
7
8  Calculate eigenvectors using normalized trainingset
9  Take highest eigenvectors
10 Redefine image-vector in eigenspace using eigenvectors
11
12 Pre-process testset with calculated mean and eigenvectors
13 Compare redefined test-vectors with training-vectors
```

The data handed to us is first divided into two parts, the training set and the test set. The test set is set aside for later, we work our PCA only with the training set. During the division, we also reshape every image from a $M \times N$ array to a $MN \times 1$ vector. Every vector is then concatenated into a array, with in every column an image-vector. From the training set we calculate the mean of every image. The mean is a $MN \times 1$ vector. The training set is then normalized by subtracting the mean vector from every image-vector.
With the resulted normalized trainingset we can calculate the covariance-matrix and the eigenvectors of the

covariance-matrix. These eigenvectors are of the same size as an image, so we can visualise them. Below are the first nine eigenvectors visualised(which are now eigenimages).

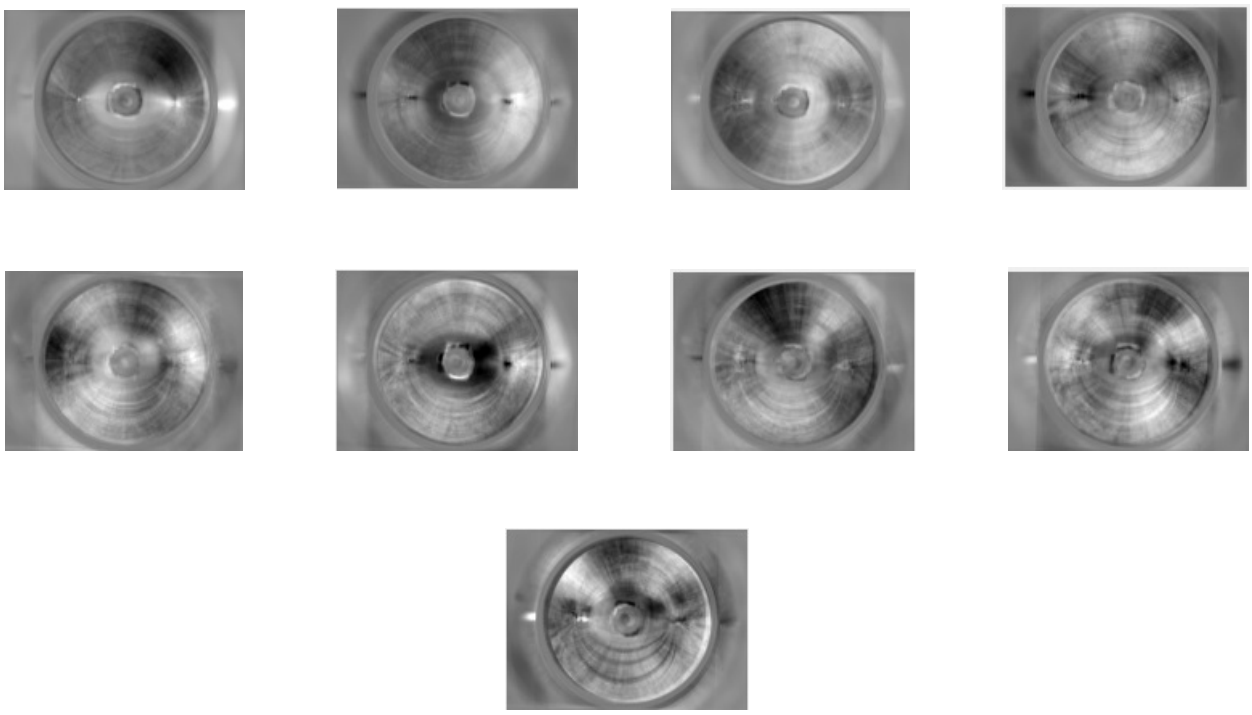

Figure 1: The eigenvector, visualised as eigenimages

Important with PCA is that not every eigenvector is important to determine the final image. It is main principle of PCA. We only take the eigenvectors which are significant. If we take the eigenwaarden corresponding to the eigenvectors, we can look at its significance. Below we have plotted the 50 highest eigenwaarden. Seeing this plot, we can concluded it won't be sufficient to take more than the first 15-20 eigenvectors, since after which the eigenwaarden become more less the same, so the eigenvectors are not that important. With the selected eigenvectors, we

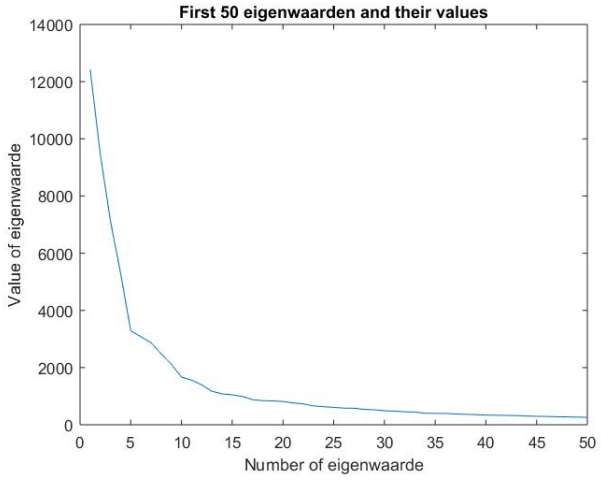

Figure 2: The 50 highest eigenwaarden

can now describe every image in eigenspace, by calculating the projection of the image-vector on the eigenvector. We have now redefined every image, from a 112×155 array to a $1 \times d$ vector, where d is the number of eigenvectors we use. We will compare both in time and accuracy by calcaluting the euclidian distance between one image and its second best match. The results are illustrated below.

```
Number of components:
    50

Elapsed time is 0.004738 seconds.
Using PCA image detection, we found:
   109

Elapsed time is 0.052553 seconds.
Using naive image detection, we found:
   109
```

Figure 3: Comparison between PCA and naive image detection

As we can see, both images choose the same image as the second best match to a random image. Important to notice is the difference between the execution time of both method. The PCA method is more than 10 times faster than the naive method. Not to forget that PCA uses a lot less memory than naive image detection($d \times 1$ vector versus $imagesize \times 1$ vector).

5 Experiments

We have used our PCA program in order to calculate the position of images in the testdata, by comparing component vector of the testdata image with the component vectors of the training images. We diversify in the variable d, the number of eigenvectors used to construct the component vectors. The figure below shows nicely the difference between d-values.

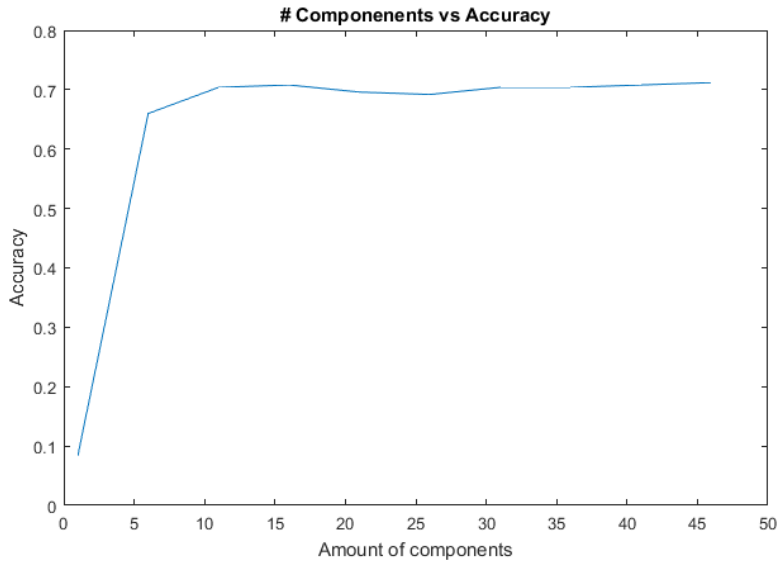

Figure 4: Plot of the accuracy vs amount of PCA componentens used

We reached a accuracy of more less 70% using the PCA components. However we need to keep in mind that a higher number of components needs more memory, so ideally 10 to 20 components are good enough to calculate the position of the robot.
We could optimize the program by removing the black border around the images, but the effect of this operation would be minimal, since the distance between images will be defined by variance in pixels, so a black border does not have any influence on this.

We also used the k-nearestneighbour algorithm to compare the testdata with the trainingdata using only the raw images. The result was comparable with our result of the PCA method in time or accuracy, but in memory the second method is far more costly.

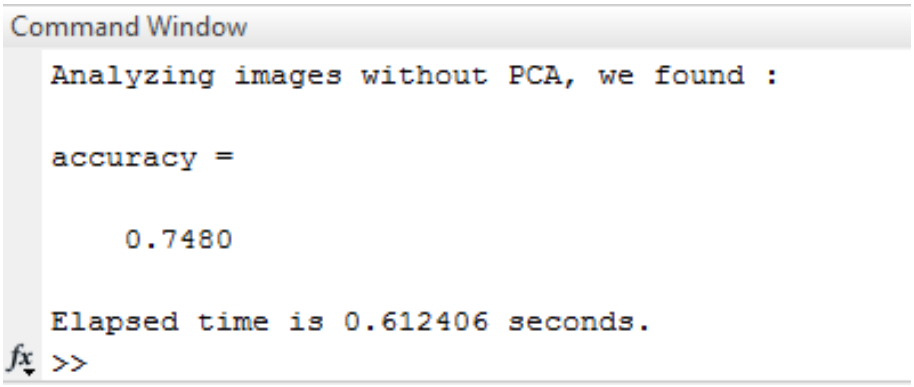

Figure 5: Execution time withouth PCA

6 Conclusion

After we implemented Principal Component Analysis to reduce with the goal to reduce the image dimensions, we experimented with the the great potential it has in image comparison. We saw that by using it there can huge memory savings be done without losing accuracy when comparing images.