1. The Covariance Method

We indeed recover the same principal components as are shown in Figure 1 of the exercise sheet. The exact data is outlined in Table 1.

Computed Values	Python Reference Implementation
3.453580	3.453580
0.438344	0.438344

Table 1: Eigenvalues for the 2D data set. Only the first 6 digits after the decimal point are shown.

The original dataset uses $1850 * 1280$ entries, or 2368000 floating point numbers in total. The compressed data only used $10 * 1280$ (PC) $+10 * 1850$ (data) $+2 * 1850$ (scaler), which is equal to 35000. We therefore have a compression rate of ≈ 67.66.
As we can see in Figures 1 and 2 the solutions are qualitatively the same, up to a factor of -1 (note that the faces are the inverses of each other and that the direction of the principal components can be inversed). The reconstructed faces in Figures 5 and 6 show that the chosen number of principal components is too little, as there is quite a big reconstruction error. The reconstructed faces from the C++ code look better, though this can be accredited to the input faces also being of a higher resolution.

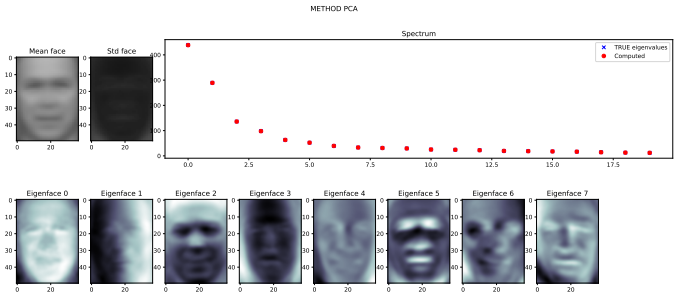

Figure 1: Values of the first 20 eigenvalues as well as the first 8 eigenfaces from our data set, calculated with our C++ code.

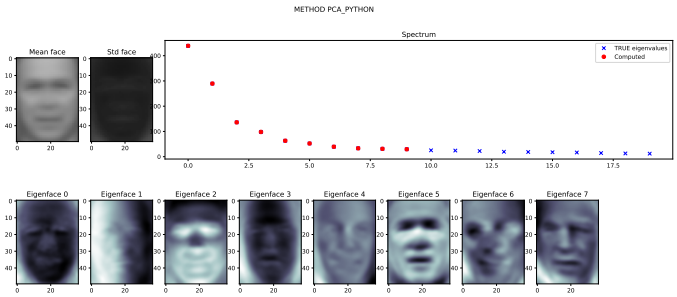

Figure 2: Values of the first 20 eigenvalues as well as the first 8 eigenfaces from our data set, calculated with the reference Python code.

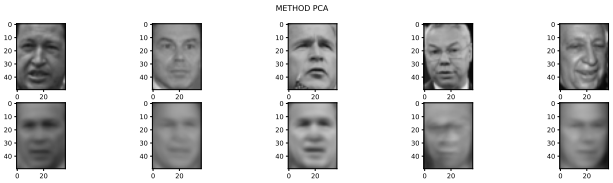

Figure 3: Values of 5 faces from our dataset as well as the reconstruction after PCA with 10 components, calculated with our C++ code.

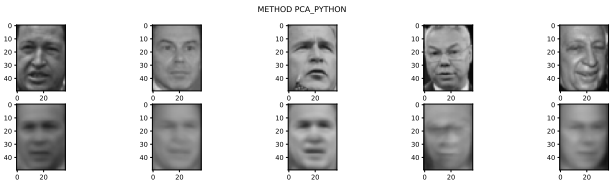

Figure 4: Values of 5 faces from our dataset as well as the reconstruction after PCA with 10 components, calculated with the reference Python code.

2. Principal Component Analysis with Oja's rule

The eigenvalues retrieved from Hebb's rule are dependant on the number of iterations. With Hebb's rule our eigenvalues are diverging (going to $+\inf$), which is not well suited for our purpose. We also get the same eigenfaces for the principal components. With Oja's rule we don't have the problem of divergence anymore, but the issue of recovering the same principal components is still the same. Additionally, due to the stochastic nature of the algorithm (random weight initialization) we recover slightly different numerical results every time we run our code. The principal component we get from Oja's rule (and Hebb's rule) is similar to the one from the Python reference implemenation and the covariance implementation, though not completely the same. Interestingly enough the first principal component is the brightness of the face. We expect this to always be the first principal component, though the extent of it may vary due to several reasons, including numerical differences from the random weight initialization. For Sanger's rule we observe that it takes many iterations to attain good eigenvalues. While the first eigenvalue is similar to the reference implementation after a few hundred iterations, the other eigenfaces take significantly longer to attain similar values. We assume that the algorithm is unstable, as once can most probably find parameter combinations that will make the algorithm diverge or converge to a local minima rather than the global one (principal components). However, we didn't let our code run enough iterations in order to be able to find such scenarios.

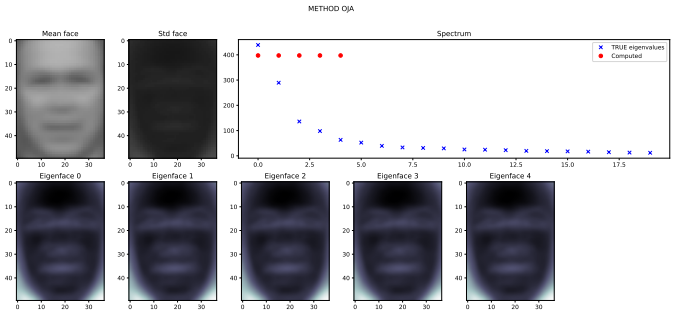

Figure 5: Values of the first 5 eigenvalues as well as the first 5 eigenfaces from our data set,, calculated with Oja's rule.

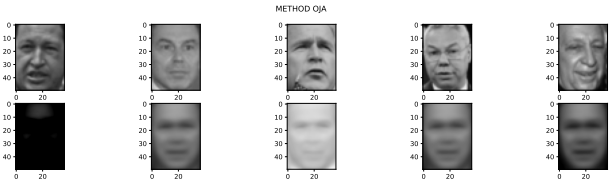

Figure 6: Values of 5 faces from our dataset as well as the reconstruction after PCA with 5 components, calculated with Oja's rule.