Question 1: Principal Component Analysis (PCA)

Note: The implementation of the code corresponding to the results shown below can be explored under Appendix.

2D Dataset.

The first task was to implement a PCA algorithm in C++ on a provided 2D data set and try to obtain the same principal components given as reference. Figure 1 illustrates the results of the C++ implementation of the PCA. Both principal components could be found successfully. Note that the inverting of the eigenvalues, which represent the principal components, has nothing to do with the correctness of the result, because only the direction and size of the eigenvectors is relevant. As long as the direction of the eigenvector is the same (or the inverted direction) the principal component is correctly computed. The eigenvalues yielded by the PCA are λ_1 = 3.45358 and λ_2 = 0.438344.

(a) (b)

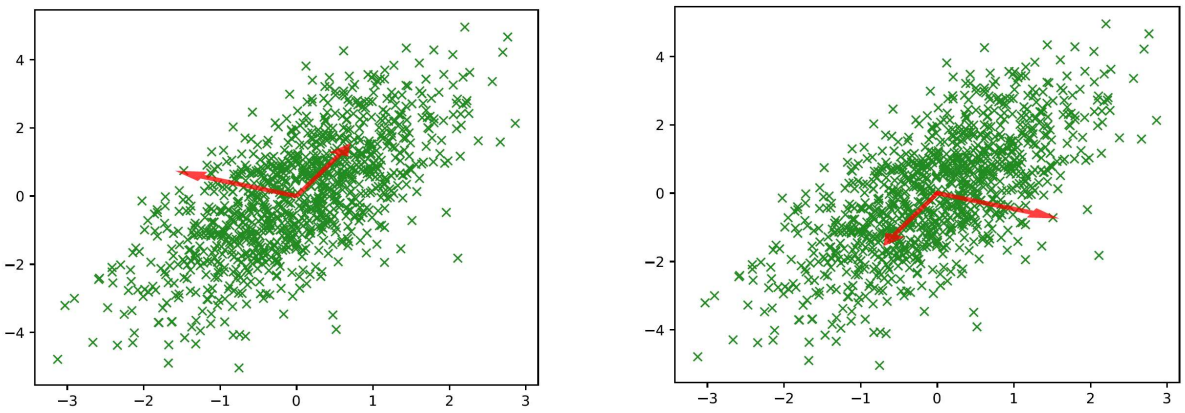
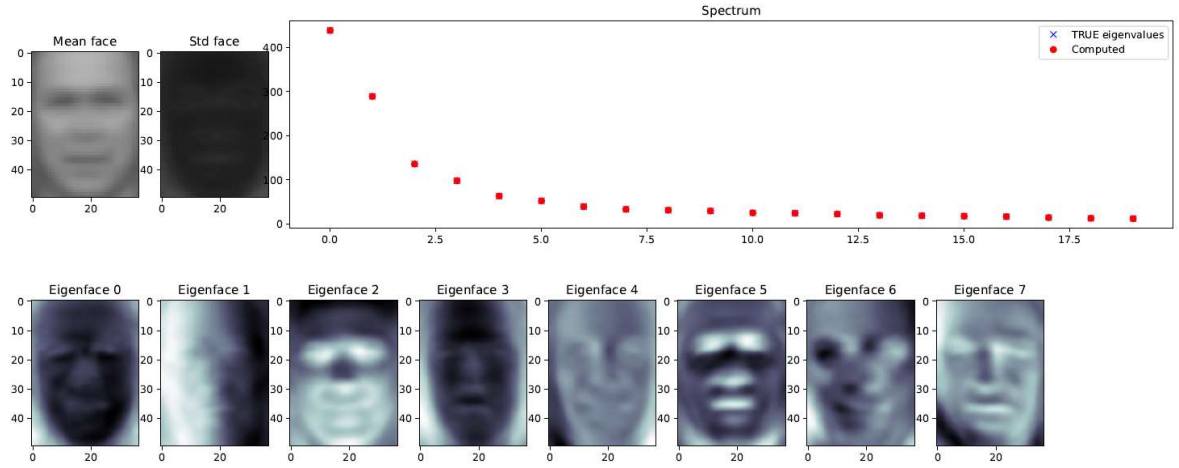


Figure 1. Obtained principal components by **(a)** the C++ implementation and **(b)** by the given Python implementation.

Faces Dataset.

The second task was to apply the PCA algorithm programmed for the 2D data set on a real world data set of images. Figure 2 shows a comparison of the components obtained by the C++ implementation and the components yielded by the Python code. As can be seen from the figure, the C++ PCA code was able to compute all the eigenvalues correctly as they match the correct eigenvalues given.

(a)



(b)

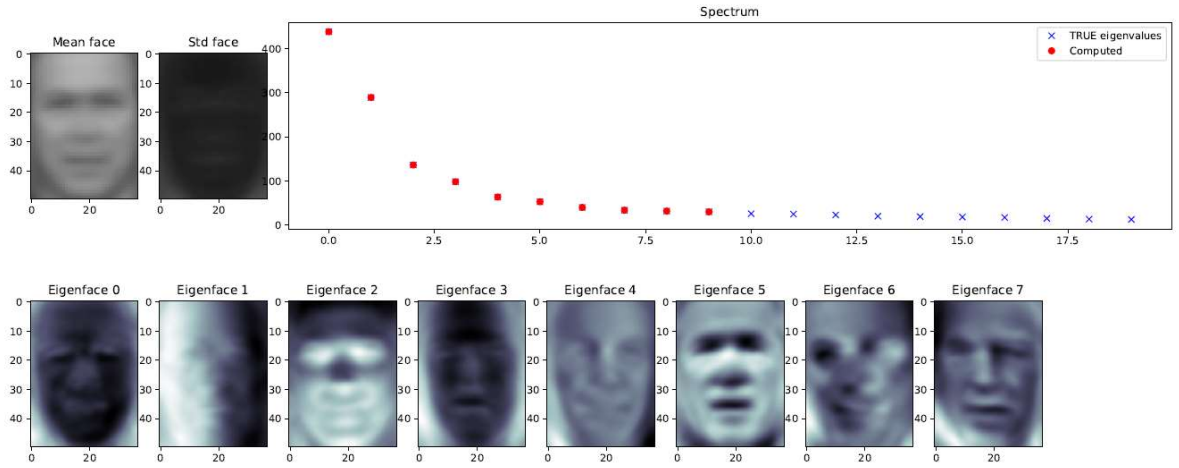
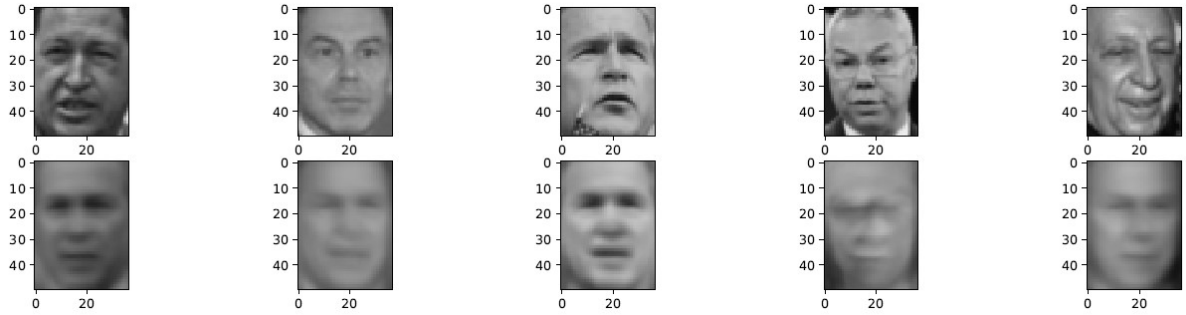


Figure 2. Plots displaying the results of the components yielded by the PCA algorithm implemented in **(a)** C++ and in **(b)** Python. The blue crosses illustrate the correct reference values. As shown in **(a)** the C++ implementation matches the results correctly for all the eigenvalues.

After the correct components were achieved the 10 biggest components were extracted and stored as the vectors of a matrix V_m. This matrix was then used to compress the provided data according to $Y = XV_m$. The obtained matrix Y was the used together with the matrix that stores the first 10 principal components to reconstruct the data. Hereby, the formula $X = YV_m^T$ was used. The following Figure 3 shows the original data faces compared to the reconstructed faces for both the C++ and the Python implementation. Contrasting the reconstructed faces between the two coded PCA implementations (C++ and Python) no difference can be observed, thus, the C++ implementation can be considered as correct.

(a)



(b)

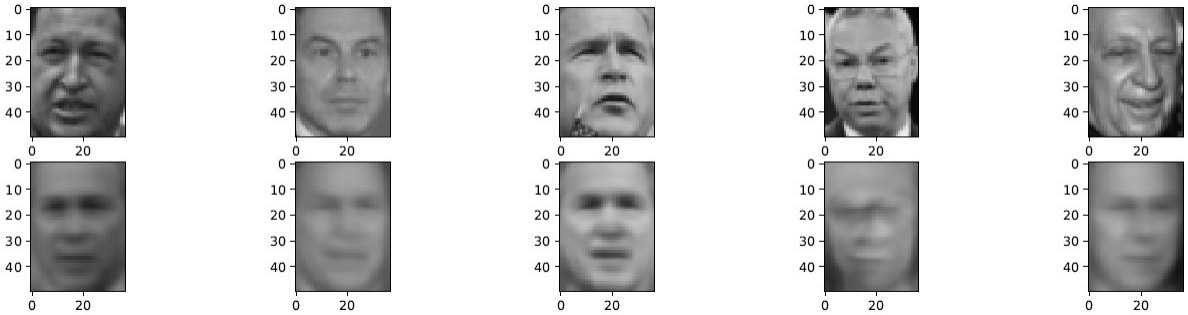


Figure 3. Reconstruction of the original data size using the reduced dataset computed by **(a)** the implemented C++ algorithm and **(b)** the Python based PCA version. The reconstructed faces show no difference in their appearance, thus, it can be stated that the faces were correctly reconstructed.

Compression ratio.

The compression ratio is the amount of data stored for the original data set divided by the amount of data stored to reconstruct the original data set. In this case, the data that was needed to carry out the reconstruction was the data stored in the matrix Y (compressed data of dim [N x num_comp]), the matrix V_m (principal components [D x num_comp]), and the mean [N x 1] and standard deviation [N x 1] of each sample in order to de-standardize and de-centre the reconstructed data. The obtained compression ratio cr = 69.

```
////////////////////////////////////////////////////////////////////////////
//    8. Report the compression ratio

// TODO: Compute the dimension of the original data vs. dimensions of compressed
int dim_old = N*D;
int dim_comp = N*num_comp + D*num_comp + N*2;
std::cout << "COMPRESSION RATIO = " << dim_old/dim_comp << std::endl;
```

Figure 4. C++ implementation for the computation of the compression ratio going from the original data set to the compressed one.

Question 2: Principal Component Analysis with Oja's rule

Note: The implementation of the code corresponding to the results shown below can be explored under Appendix.

2D Dataset.

The first task was to implement Oja's rule and Sanger's rule on the given 2D data set. The results are illustrated in the figures below. Oja's rule allowed to recover the dominant eigenvector of the dataset, whereas Sanger's rule allowed to recover both eigenvectors correctly. This behaviour is as expected.

In order to obtain the correct second principal component using Sanger's rule it was necessary to tune the training parameters. The learning rate was set to 1e-6 instead of the initially provided 1e-7. Generally, bigger learning rates resulted in more accurate eigenvalues. The eigenvalues obtained using Sanger's rule are λ_1 = 3.45358 and λ_2 = 0.438341.

(a) **(b)**

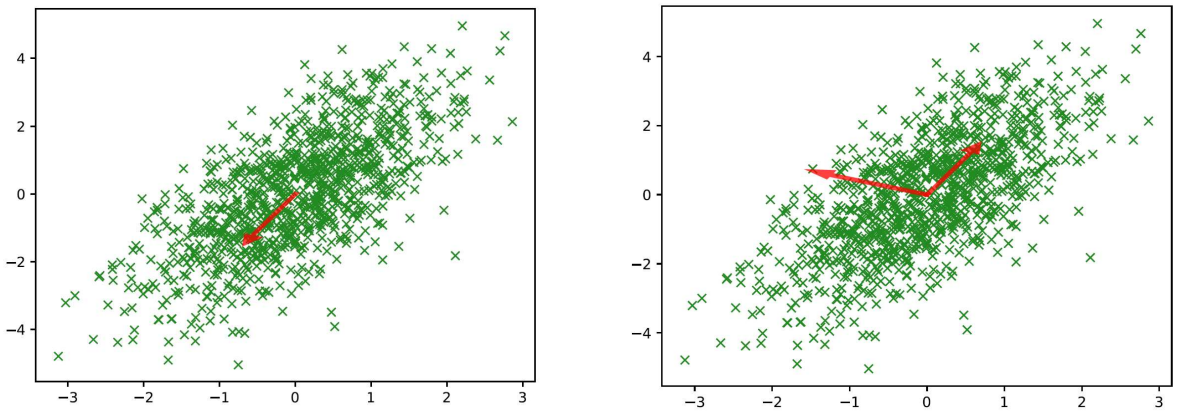


Figure 5. Principal components of the given 2D dataset computed by **(a)** Oja's rule and **(b)** Sanger's rule. The plots clearly reveal that Oja's rule converges to the dominant eigenvector of the data set. On the other hand, Sanger's rule, which is an extension to Oja's rule, allows retrieving both principal components and its corresponding eigenvalues correctly.

```
// DATA PARAMETERS
int D = 2;  // Data dimension
int N = 1024; // Number of training samples
int num_comp = 2; // Number of principal components
std::string data_name = "2D"; // Data path
std::string scaler = "center"; // Scaler type
std::string weight_init = "normal"; // "normal" or "allsame"
std::string method_name = "OJA"; // Method
std::string data_path = "./data/"+data_name+"_dataset.txt"; // Data path
// TRAINING PARAMETERS
const int nepoch = 20000; // Number of epochs
const double learn_rate = 1e-6; // Learning rate
const double tolerance = 0.0; // 1e-18;
const int batch_size = 1; // Batch-size
const int check_every = 10; // Frequency of checking the convergence criterion
```

Figure 6. Training parameters used for the implementation of Oja's and Sanger's rules for the 2D data set.

Faces Dataset.

The second task was to implement the algorithm used for the 2D data set on the provided faces data set as well. The implementation of Oja's rule allowed a correct computation of the dominant eigenvalue and its components, i.e. the first principal component of the dataset. The dominant eigenvalue computed with the training parameters below is λ_1 = 418.89. This result is quite close to the computed dominant eigenvalue yielded by the traditional PCA algorithm, λ_1 = 439.01. It is to say that the value of the dominant principal component obtained by Oja's rule changes by altering the training parameters. With other training parameters than the ones shown in Figure 7 it was possible to get closer to the dominant principal component of the PCA code. The closest eigenvalue was λ_1 = 438.46.

The implementation of Hebb's rule was neither able to recover the correct principal components nor the correct eigenvalues. Using Hebb's rule without the normalization of Oja leads to the dominant eigenvalue growing towards infinity.

Using Sanger's rule it was possible to compute the eigenvalues of the principal components quite well. However, in order to obtain eigenvalues close to the correct ones many iterations changing the training parameters were necessary, because of Sanger's rule being notoriously unstable. The training data values provided below allowed getting relatively close to the real values obtained for the first 5 principal components using the traditional PCA algorithm. The eigenvalue of component number 2 is still slightly off, but any changes in the training set up caused the eigenvalues of the higher principal components being quite far off. The first principal component can be obtained really well with almost all reasonable sets of training parameters. However, larger eigenvalues are more difficult to obtain correctly. With the finally used training parameters shown in Figure 7,

In order to achieve the results in Figure 9 the batch size was increased by a factor 4 from 32 to 128. The learning rate was kept at 1e-6 as both lower and bigger learning rates resulted in more inaccurate results. To achieve even better results the number of epochs was increased by 10, i.e. from 100 to 110. With all the other training parameters in place, as shown in Figure 7, this number of epochs yielded results that came closest to the ones obtained by PCA.

```
// DATA PARAMETERS
int D = 1850;  // Data dimension
int N = 1280; // Number of training samples
int num_comp = 5; // Number of principal components
std::string data_name = "faces"; // Data path
std::string scaler = "standard"; // Scaler type
std::string weight_init = "allsame"; // "normal" or "allsame"
std::string method_name = "OJA"; // Method
std::string data_path = "./data/"+data_name+"_dataset.txt"; // Data path
// TRAINING PARAMETERS
const int nepoch = 110;          // Number of epochs
const double learn_rate = 1e-6; // Learning rate
const double tolerance = 0.0; // Convergence criterion (tolerance)
const int batch_size = 128;      // Batch-size
const int check_every = 1; // Frequency of checking the convergence criterion
```

Figure 7. Training parameters used for the implementation of Oja's and Sanger's rules on the faces data set.

```
Eigenvalues/std
i = 0 - 418.890383
i = 1 - 221.420583
i = 2 - 135.187819
i = 3 - 90.510274
i = 4 - 64.587291
[flindlbauer@eu-login-14 skeleton_code]$
```

Figure 8. Obtained eigenvalues using the training parameters of Figure 07.

Results using Sanger's rule.

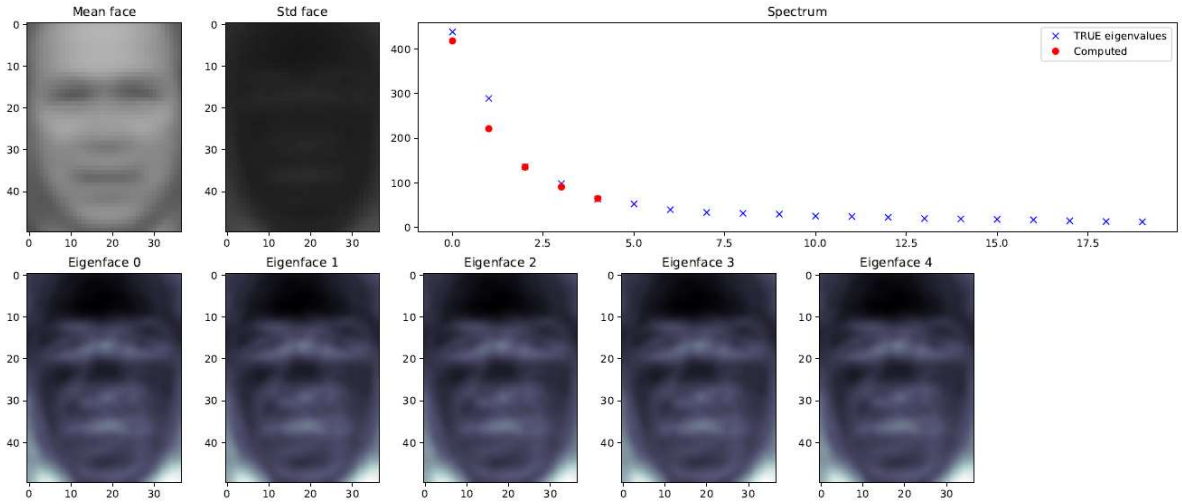
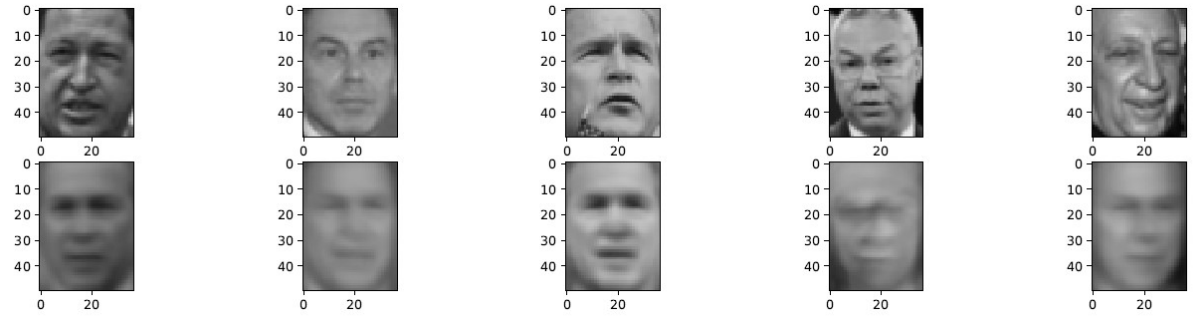


Figure 9. Depiction of the computed eigenvalues using Sanger's rule. Sanger's rule allowed to compute the eigenvalues of the first 5 principal components using the above illustrated training parameters quite well.

Face reconstruction.

(a)



(b)

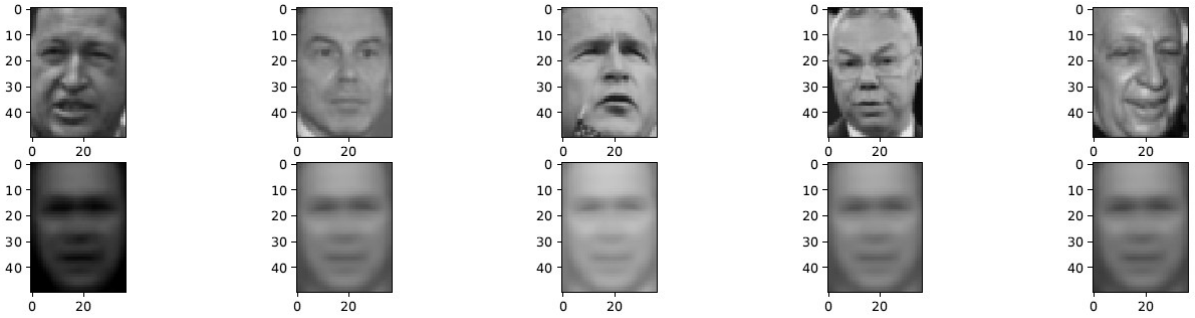


Figure 10. Reconstruction of the original data size using the reduced dataset computed by **(b)** Sanger's rule. The Python implementation of the face reconstruction is given in **(a)**. As can be seen from the result, the implementation of Sanger's rule did not manage to recover the data as well as the Python based PCA algorithm. The reason is the instability of Sanger's rule that made it very difficult to obtain eigenvalues and eigenvectors close to the correct ones. Furthermore, only the first 5 eigenvalues were used.

Appendix

Code main_pca.cpp

The only code to write in this file concerned the compression ratio from the original input data set to the compressed one.

Code main_oja.cpp

No code was to do here. The training set up is shown in the report above.

Code utils.h

```
void transposeData(double* data_T, const double* const data, const int N, const int D)
{
    // Data are given in the form data(n,d)=data[n*D+d]
    // This function transposes the data to data(n,d)=data_T[d*N+n]
    // for the purpose of more efficient memory layout
    // (e.g. calculation of the mean)

    for (int i = 0; i < N; i++)
        for (int k = 0; k < D; k++)
            data_T[i+k*N] = data[k+i*D];

}
```

Figure 11. To make the handling of the input data more convenient throughout the program the matrix was transposed. The matrix was transposed from a row major ordering to a column major ordering.

```
void computeMean(double* mean, const double* const data_T, const int N, const int D)
{
    // Calculation of the mean (over samples) of the dataset
    // data(n,d)=data_T[d*N+n]

    // TODO:
    double sum = 0;
    int start = 0;
    int lim = N;
    // Loop over data dimensions.
    for (int i = 0; i < D; i++) {
        // Accumulate data of dimension i.
        for (int k = start; k < lim; k++)
            sum += data_T[k];
        // Compute mean of dimension i.
        mean[i] = sum/N;

        // Update variables.
        sum = 0;
        start += N;
        lim += N;
    }

    // :TODO
}
```

Figure 12. Computation of the mean of each data component.

```
void computeStd(double* std, const double* const mean, const double* const data_T, const int N, const int D)
{
    // Calculation of the mean (over samples) of the dataset
    // data(n,d)=data_T[d*N+n]

    // TODO:

    int start = 0;
    int lim = N;
    double nom = 0;
    double r = 0;
    // Loop over data dimensions.
    for (int i = 0; i < D; i++) {
        // Loop over data values of dimension i.
        for (int k = start; k < lim; k++) {
            r = data_T[k]-mean[i];
            nom += r * r;
        }
        // Compute std of dimension i.
        std[i] = sqrt(nom/N);
        // Update loop variables.
        start += N;
        lim += N;
    nom = 0;
    }

    // :TODO
}
```

Figure 13. Computation of the standard deviation of each data component.

```
void standardizeColMajor(double* data_T, const double* const mean, const double* const std, const int N, const int D)
{
    std::cout << "Scaling - zero mean, unit variance." << std::endl;
    // COL MAJOR IMPLEMENTATION
    // Data normalization (or standardization)
    // Transormation of the data to zero mean unit variance.
    // data(n,d)=data_T[d*N+n]

    // TODO:

    int start = 0;
    int lim = N;
    double nom = 0;
    double r = 0;
    // Loop over data dimensions.
    for (int i = 0; i < D; i++) {
        // Loop over data values of dimension i.
        for (int k = start; k < lim; k++) {
            data_T[k] = (data_T[k]-mean[i])/std[i];
        }
        // Update loop variables.
        start += N;
        lim += N;
    }

    // :TODO
}
```

Figure 14. Standardization of each data component, i.e. the data is first centred by subtracting its mean and then standardized by dividing by its standard deviation.

```
void centerDataColMajor(double* data_T, const double* const mean, const int N, const int D)
{
    std::cout << "Centering data..." << std::endl;
    // COL MAJOR IMPLEMENTATION
    // data(n,d)=data_T[d*N + n]

    // TODO:
    int start = 0;
    int lim = N;
    double nom = 0;
    double r = 0;
    // Loop over data dimensions.
    for (int i = 0; i < D; i++) {
        // Loop over data values of dimension i.
        for (int k = start; k < lim; k++) {
            data_T[k] = data_T[k]-mean[i];
        }
        // Update loop variables.
        start += N;
        lim += N;
    }

    // :TODO
}
```

Figure 15. Centring of the data by subtracting its mean.

```
void constructCovariance(double* C, const double* const data_T, const int N, const int D)
{
    // Construct the covariance matrix (DxD) of the data.
    // data(n,d)=data_T[d*N+n]
    // For the covariance follow the row major notation
    // C(j,k)=C[j*D+k]

    // TODO:

    int NN = N*D;
    int dim = -1;
    for (int k = 0; k < NN; k+=N) {
        for (int j = 0; j < NN; j+=N) {
        dim++;
        // Init value of C to zero.
        C[dim] = 0;
        for (int i = 0; i < N; i++) {
            C[dim] += data_T[i+k] * data_T[i+j];
        }
        C[dim] = C[dim]/(N-1);
        }
    }

    // :TODO
}
```

Figure 16. Computation of the covariance matrix.

```
void getEigenvectors(double* V, const double* const C, const int NC, const int D)
{
    // Extracting the last rows from matrix C containig the PCA components (eigenvec
    // Be carefull to extract them in order of descenting variance.
    // C(j,d)=C[j*D+d] # ROW MAJOR
    // V(k,d)=V[k*D+d] # ROW MAJOR

    // TODO:

    int lim = D*D;
    int start = lim-D;
    int cnt = 0;
    for (int i = 0; i < NC; i++) {
        for (start; start < lim; start++) {
        V[cnt] = C[start];
        cnt++;
        }
        // Update start and limit.
        lim -= D;
        start = lim-D;
    }

    // TODO
}
```

Figure 17. Code to extract the eigenvectors of the covariance matrix.

```
void reconstructDatasetRowMajor(double* data_rec, const double* const V, const double* const data_red, const int N, const int D, const int NC)
{
    // ROW MAJOR
    // V(c,d)=V[d + c*D]
    // data_red(n,c)=data_red[c + n*NC], C<<D   # ROW MAJOR
    // data_rec(n,d)=data_rec[d + n*D]          # ROW MAJOR

    // TODO:

    for (int d = 0; d < D; d++) {
        for (int r = 0; r < N; r++) {
            double sum = 0.0;
        for (int c = 0; c < NC; c++)
            sum += data_red[r*NC+c] * V[c*D+d];
        data_rec[r*D+d] = sum;
        }
    }

    // :TODO
}
```

Figure 18. Reconstruction of the original data size.

Code perceptron.h

```
void ojasRuleGradient(const double *const input, const int batch_size) {
  forward(input, batch_size);
  memset(gradient, 0.0, sizeof(double) * nOutputs * nInputs);
  // TODO:

  for (int k = 0; k < batch_size; k++) {
      for (int i = 0; i < nInputs; i++) {
          for (int o = 0; o < nOutputs; o++) {
            gradient[i*nOutputs + o] += output[k*nOutputs + o]
              * (input[k*nInputs + i] - output[k*nOutputs + o] * weights[i*nOutputs + o]);
          }
      }
  }

  // :TODO

  // Normalize gradient.
  for (int i = 0; i < nInputs * nOutputs; ++i) {
    gradient[i] = gradient[i] / batch_size;
  }
}
```

Figure 19. Implementation of Oja's rule.

```
void sangersRuleGradient(const double *const input, const int batch_size) {
  forward(input, batch_size);
  memset(gradient, 0.0, sizeof(double) * nOutputs * nInputs);
  // TODO:
  //
  // Sum of Sanger's rule formula.
  double osum;
  int lim;

  for (int k = 0; k < batch_size;  k++) {
      for (int  i = 0; i < nInputs; i++) {
          for (int o = 0; o < nOutputs; o++) {
              osum = 0;
              lim = o+1;
              for (int os = 0; os < lim; os++) {
                  osum += output[k*nOutputs + os] * weights[i*nOutputs + os];
              }
              gradient[i*nOutputs + o] += output[k*nOutputs + o] * (input[k*nInputs + i] - osum);
          }
      }
  }

  // :TODO

  // Normalize gradient.
  for (int i = 0; i < nInputs * nOutputs; ++i) {
    gradient[i] = gradient[i] / batch_size;
  }
}
```

Figure 20. Implementation of Sanger's rule.

```
void computeEigenvalues(const double *const input, const int batch_size) {
  // The eigenvalues are given by the standard deviation at the output
  forward(input, batch_size);
  memset(eigenvalues, 0.0, sizeof(double) * nOutputs);
  memset(mean, 0.0, sizeof(double) * nOutputs);

  // TODO:

  int cnt;

  // Compute the mean.
  for (int i = 0; i < batch_size*nOutputs; i+=nOutputs) {
    cnt = 0;
    for (int k = i; k < i+nOutputs; k++) {
        mean[cnt] += output[k];
        cnt++;
    }
  }
  for (int i = 0; i < nOutputs; i++)
    mean[i] = mean[i]/batch_size;

  // Compute standard deviation, i.e. the eigenvalues.
  for (int i = 0; i < batch_size*nOutputs; i+=nOutputs) {
    cnt = 0;
    for (int k = i; k < i+nOutputs; k++) {
      eigenvalues[cnt] += (output[k] - mean[cnt]) * (output[k] - mean[cnt]);
      cnt++;
    }
  }

  for (int i = 0; i < nOutputs; i++) {
    eigenvalues[i] = eigenvalues[i]/(batch_size-1);
    printf("i = %d - %f\n", i, eigenvalues[i]);
  }

  // :TODO
}
```

Figure 21. Computation of the eigenvalues of the data set. The eigenvalues are given by the standard deviation of the outputs. The outputs are the compressed batch size vectors, i.e. the eigenvectors of each batch sample. These vectors represent the output vector of the neural net for each batch.