NumCSE

Autumn Semester 2017

Prof. Rima Alaifari

Exercise sheet 4

Constrained least squares, singular value decomposition, principal components analysis

A. Dabrowski

Problem 4.1: Matrix least squares in Frobenius norm

We use two different approaches (Lagrange multipliers and augmented normal equations) to solve a constrained minimization problem.

Fix non-zero vectors $\mathbf{z} \in \mathbb{R}^n$ and $\mathbf{g} \in \mathbb{R}^n$. We want to find the matrix $\mathbf{M}^*$ which minimizes the Frobenius norm among all matrices $\mathbf{M}$ which belong to the set

$$\{\mathbf{M} \in \mathbb{R}^{n,n},\ \mathbf{M}\mathbf{z} = \mathbf{g}\}. \tag{4.1}$$

Recall that the Frobenius norm is defined as

$$\|\mathbf{M}\|_F = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n} |\mathbf{M}_{i,j}|^2}.$$

First, we look at the method of Lagrange multipliers to solve Eq. (4.1) analytically.

(a) Let $L : \mathbb{R}^{n,n} \times \mathbb{R}^n \longrightarrow \mathbb{R}$ be the functional defined as

$$L(\mathbf{M}, \boldsymbol{\lambda}) = \|\mathbf{M}\|_F^2 + \boldsymbol{\lambda}^\top(\mathbf{M}\mathbf{z} - \mathbf{g}).$$

Derive simbolically the critical points of L, that is find analytically $\mathbf{M}$ and λ which solve $\nabla L(\mathbf{M}, \boldsymbol{\lambda}) = 0$.

SOLUTION:

The derivative of L with respect to the component $\mathbf{M}_i^j$ is

$$\frac{\partial L}{\partial \mathbf{M}_i^j} = 2\mathbf{M}_i^j + \boldsymbol{\lambda}_i \mathbf{z}_j. \tag{4.2}$$

Imposing that the previous quantity is zero for every i and j is equivalent to asking that

$$2\mathbf{M} + \boldsymbol{\lambda}\mathbf{z}^\top = 0. \tag{4.3}$$

In the same way, deriving L with respet to $\boldsymbol{\lambda}_i$ and imposing that it is zero for every i, we obtain

$$\mathbf{M}\mathbf{z} - \mathbf{g} = 0. \tag{4.4}$$

Plugging (4.3) into (4.4) we obtain:

$$-\frac{1}{2}\boldsymbol{\lambda}\|\mathbf{z}\|_2^2 - \mathbf{g} = 0 \quad \Longrightarrow \quad \boldsymbol{\lambda} = -2\frac{\mathbf{g}}{\|\mathbf{z}\|_2^2} \quad \overset{((4.3))}{\Longrightarrow} \quad \mathbf{M}^* = \frac{\mathbf{g}\mathbf{z}^\top}{\|\mathbf{z}\|_2^2}. \tag{4.5}$$

Notice that the matrix $\mathbf{M}^*$ is the outer product of two vectors.

Another approach, which we now explore, is to recast Eq. (4.1) in a linear least squares form and use the augmented normal equations method to compute the solution.

(b) Reformulate the problem as an equivalent standard linearly constrained least squares problem. That is find suitable matrices $\mathbf{A}$, $\mathbf{C}$ and vectors $\mathbf{b}$ and $\mathbf{d}$, which you should specify based on $\mathbf{z}$ and $\mathbf{g}$, so that the minimization of

$$\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2$$

among all $\mathbf{x} \in \mathbb{R}^{n^2}$ such that

$$\mathbf{C}\mathbf{x} = \mathbf{d}$$

will be equivalent (after an appropriate reshaping of $\mathbf{x}$) to the minimization of (4.1).

Hint: to define $\mathbf{C}$ you may use the Kronecker product.

SOLUTION:

We call $\boldsymbol{\lambda}^* := \text{flatten}(\mathbf{M}^{*\top}) \in \mathbb{R}^{n^2}$ the vector obtained by the concatenation of the rows of $\mathbf{M}^*$. Then the functional to be minimised is just the 2-norm of $\boldsymbol{\lambda}^*$, that is we choose $\mathbf{A} = \mathrm{Id}_{n^2}$ and $\mathbf{b} = \mathbf{0}$. The constraint $\mathbf{M}^*\mathbf{z} = \mathbf{g}$ can be written as $\mathbf{C}\boldsymbol{\lambda}^* = \mathbf{d}$, where $\mathbf{d} = \mathbf{g}$ and the $n \times n^2$ matrix $\mathbf{C} := \mathrm{Id}_n \otimes \mathbf{z}^\top$ is the Kronecker product of the $n \times n$ identity matrix and $\mathbf{z}^\top$. In symbols

$$\mathbf{C} = \begin{bmatrix} [z_1, \dots, z_n] & \mathbf{0}^\top & \cdots & \mathbf{0}^\top \\ \mathbf{0}^\top & \mathbf{z}^\top & \ddots & \vdots \\ \vdots & \ddots & \ddots & \mathbf{0} \\ \mathbf{0}^\top & \cdots & \mathbf{0}^\top & \mathbf{z}^\top \end{bmatrix}.$$

$\mathbf{C}$ has full rank since $\mathbf{z}$ has at least one non-zero component.

(c) State the *augmented normal equations* corresponding to the constrained linear least squares problem of Point (b) and give necessary and sufficient conditions on $\mathbf{g}$ and $\mathbf{z}$ that ensure existence and uniqueness of solutions.

SOLUTION:

The *augmented normal equations* form is:

$$\begin{bmatrix} \mathrm{Id}_{n^2} & \mathbf{C}^\top \\ \mathbf{C} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{\lambda}^* \\ \mathbf{p} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{g} \end{bmatrix} \tag{4.6}$$

By block Gaussian elimination we can rewrite this as

$$\begin{bmatrix} \mathrm{Id}_{n^2} & \mathbf{C}^\top \\ 0 & -\mathbf{C}\mathbf{C}^\top \end{bmatrix} \begin{bmatrix} \boldsymbol{\lambda}^* \\ \mathbf{p} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{g} \end{bmatrix} \tag{4.7}$$

The solution of

$$-\mathbf{C}\mathbf{C}^\top\mathbf{p} = \mathbf{g} \tag{4.8}$$

exists and is unique since by Point (b) the matrix $\mathbf{C}\mathbf{C}^\top$ is regular.

(d) Implement a C++ function

```
MatrixXd min_frob(const VectorXd &z, const VectorXd &g)
```

that computes the solution $\mathbf{M}^*$ of the minimization of Eq. (4.1) given the vectors $\mathbf{z}, \mathbf{g} \in \mathbb{R}^n$. Use the augmented normal equations approach.

Hint: You may use

```
#include <unsupported/Eigen/KroneckerProduct>
```

SOLUTION:

```
MatrixXd min_frob(const VectorXd & z, const VectorXd & g) {

    int n = g.size();

    // Build temporary matrix C
    MatrixXd C = kroneckerProduct(MatrixXd::Identity(n,n),
                                  z.transpose());

    // Build system matrix and r.h.s.
    MatrixXd S(n*n+n,n*n+n);
    S << MatrixXd::Identity(n*n, n*n), C.transpose(),
         C                           , MatrixXd::Zero(n,n);
    VectorXd h(n*n+n);
    h << VectorXd::Zero(n*n), g;

    // Solve augmented system and return only head of solution
    // as a matrix (discard the rest).
    return Map<const MatrixXd>(S.fullPivLu().solve(h).eval().data(),
                               n, n).transpose();
    // It is possible to solve the system more efficiently
    // exploiting the special structure of the matrix.
}
```

(e) Using random vectors $\mathbf{z}$ and $\mathbf{g}$, check with a numerical experiment that the matrix given by

$$\frac{\mathbf{g}\mathbf{z}^\top}{\|\mathbf{z}\|_2^2} \tag{4.9}$$

minimizes Eq. (4.1).

SOLUTION:

```
int main() {

    int n = 10;

    VectorXd z = VectorXd::Random(n),
             g = VectorXd::Random(n);

    MatrixXd M_augmNormal = min_frob(z, g);
    MatrixXd M_lagrange = g*z.transpose() / z.squaredNorm();

    std::cout << "Norm: "
              << (M_augmNormal - M_lagrange).norm()
              << std::endl;
}
```

Problem 4.2: Recompression of the sum of low rank approximations of matrices

Large matrices of low rank can be stored more efficiently using their singular value decomposition. However, adding two low rank matrices usually leads to an increase of the rank, requiring further "recompression" by computing a low-rank best approximation of the sum. We want to implement an efficient approach to recompression.

Template: `lowRankRecompression.cpp`.

(a) Show that for a matrix $\mathbf{X} \in \mathbb{R}^{m,n}$ the following statements are equivalent:

(i) $\text{rank}(\mathbf{X}) = k$

(ii) $\mathbf{X} = \mathbf{A}\mathbf{B}^\top$ for $\mathbf{A} \in \mathbb{R}^{m,k},\ \mathbf{B} \in \mathbb{R}^{n,k},\ k \leq \min\{m,n\}$, both full rank.

Hint: for (i) $\Rightarrow$ (ii) use SVD, for (ii) $\Rightarrow$ (i) the definition of rank and its relation with the dimension of the null-space.

SOLUTION:

The rank of a matrix $\mathbf{X}$ is given by the number of its nonzero singular values, or equivalently by the dimension of its range (the subspace of $\mathbb{R}^m$ generated by the columns of $\mathbf{X}$).

(i) $\Rightarrow$ (ii): If $\text{rank}(\mathbf{X}) = k$, then $\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$, with $\sigma_i = 0,\ \forall\, i > k$.

Thus, we can omit the last rows/columns of the matrices in the decomposition:

$$\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top = \mathbf{U}_{:,1:k}\mathbf{\Sigma}_{1:k,1:k}(\mathbf{V}_{:,1:k})^\top.$$

By defining $\left\{ \begin{array}{l} \mathbf{A} := \mathbf{U}_{:,1:k}\mathbf{\Sigma}_{1:k,1:k} \\ \mathbf{B} := \mathbf{V}_{:,1:k} \end{array} \right\}$, the claim follows.

(ii)$\Rightarrow$(i): The matrix $\mathbf{B}^\top$ corresponds to a linear mapping from $\mathbb{R}^n$ to $\mathbb{R}^k$, with $k \leq n$. Since it has rank k, the range of $\mathbf{B}^\top$ is the whole space $\mathbb{R}^k$.

The matrix $\mathbf{A}$ corresponds to a linear mapping from $\mathbb{R}^k$ to $\mathbb{R}^m$, with $k \leq m$. Since it has rank k, it is injective, i.e. $\dim[\ker(\mathbf{A})] = k - \text{rank}(\mathbf{A}) = 0$, therefore the dimension of its range is k.

Finally, the rank of $\mathbf{X} = \mathbf{A}\mathbf{B}^\top : \mathbb{R}^n \to \mathbb{R}^m$ is equal to the dimension of its range, which is then k.

(b) Implement a C++ function

```
void factorize(const MatrixXd &X, int k, MatrixXd &A,
    MatrixXd &B)
```

that factorizes the matrix $\mathbf{X}$ with $\text{rank}(\mathbf{X}) = k$ into $\mathbf{A}\mathbf{B}^\top$, as in (a).

Hint: Use function **Eigen::svd** and pay attention on how to return $\mathbf{U}$, $\mathbf{\Sigma}$ and $\mathbf{V}$ correctly.

SOLUTION:

```
void factorize(const MatrixXd & X, int k,
                     MatrixXd & A, MatrixXd & B) {

```

```
    int m = X.rows();
    int n = X.cols();

    // You can ask for thin U or V to be computed
    JacobiSVD<MatrixXd> svd(X, ComputeThinU | ComputeThinV);

    VectorXd s = svd.singularValues().head(k);
    MatrixXd U = svd.matrixU();
    MatrixXd V = svd.matrixV();

    int ns = s.size();
    int safeMax = min(k, ns); // in case k > number sing.  values

    A = U.leftCols(safeMax).array().rowwise() *
        s.head(safeMax).transpose().array();
    B = V.leftCols(safeMax);
}
```

(c) Let $\mathbf{A} \in \mathbb{R}^{m,k}$, $\mathbf{B} \in \mathbb{R}^{n,k}$, and assume that k is much smaller than m and n. Write an efficient C++ function that calculates a singular value decomposition of the product $\mathbf{A}\mathbf{B}^\top = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$, where orthogonal $\mathbf{U}, \mathbf{V} \in \mathbb{R}^{n,k}$ and $\mathbf{\Sigma} \in \mathbb{R}^{k,k}$:

```
void svd_AB(const MatrixXd & A, const MatrixXd & B, MatrixXd & U,
    MatrixXd & S, MatrixXd & V);
```

Hint: If you directly compute an SVD of $\mathbf{A}\mathbf{B}^\top$, you will always have to deal with dimensions m and n, regardless of the smallness of k. Exploit the *economical QR decomposition* of both $\mathbf{A}$ and $\mathbf{B}$ separately to reduce the problem to the computation of the SVD of a small matrix.

SOLUTION:

We have the following nested factorization :

$$\mathbf{A}\mathbf{B}^\top = \mathbf{Q}_A \underbrace{\mathbf{R}_A\mathbf{R}_B^\top}_{=\tilde{\mathbf{U}}\mathbf{\Sigma}\tilde{\mathbf{V}}^\top} \mathbf{Q}_B^\top = (\underbrace{\mathbf{Q}_A\tilde{\mathbf{U}}}_{=:\mathbf{U}})\mathbf{\Sigma}(\underbrace{\mathbf{Q}_B\tilde{\mathbf{V}}}_{=:\mathbf{V}})^\top. \tag{4.10}$$

Of course, (4.10) is cheaper than the direct SVD of the $m \times n$ matrix $\mathbf{A}\mathbf{B}^\top$, because the SVD has to be computed only for the $k \times k$ matrix $\mathbf{R}_A\mathbf{R}_B^\top$, with $k \ll m, n$.

```

void svd_AB(const MatrixXd & A, const MatrixXd & B,
                    MatrixXd & U, MatrixXd & S, MatrixXd & V) {

    int m = A.rows();
    int n = B.rows();
    int k = A.cols();

    // QA: m x k; QB: n x k; RA,RB k x k
    HouseholderQR<MatrixXd> QRA = A.householderQr();
    MatrixXd QA = QRA.householderQ() * MatrixXd::Identity(m, min(m, k));
    MatrixXd RA = MatrixXd::Identity(min(m, k), m) *
        QRA.matrixQR().triangularView<Upper>();
    HouseholderQR<MatrixXd> QRB = B.householderQr();
    MatrixXd QB = QRB.householderQ() * MatrixXd::Identity(n, min(n, k));
    MatrixXd RB = MatrixXd::Identity(min(n, k), n) *
        QRB.matrixQR().triangularView<Upper>();

    // U,V: k x k
    JacobiSVD<MatrixXd> svd(RA*RB.transpose(), ComputeFullU | ComputeFullV);
    // Thin matrices are unnecessary here as RA*RB' is
    // a square k times k matrix

    S = svd.singularValues().asDiagonal();
    U = svd.matrixU();
    V = svd.matrixV();

    // U: m x k; V: n x k
    U = QA*U;
    V = QB*V;
}
```

(d) What is the asymptotic computational cost of the function `svd_AB` for a small k and $m = n \to \infty$? Discuss the effort required by the different steps of your algorithm.

SOLUTION:

The algorithm can be divided into the following steps:

1. $(\mathbf{A}, \mathbf{B}) \mapsto (\mathbf{Q}_A, \mathbf{R}_A, \mathbf{Q}_B, \mathbf{R}_B)$, i.e. two economical QR decompositions (k Householder transformations of length $n-1, n-2, \ldots, n-k+1$): each is worth $\mathcal{O}(nk^2)$ operations.
2. $(\mathbf{R}_A, \mathbf{R}_B) \mapsto \mathbf{R}_A\mathbf{R}_B^\top$, i.e. a matrix multiplication $\mathcal{O}(k^3)$, independent of n.
3. $\mathbf{R}_A\mathbf{R}_B^\top \mapsto (\tilde{\mathbf{U}}, \mathbf{\Sigma}, \tilde{\mathbf{V}})$, i.e. an SVD for the $k \times k$ matrix, independent of n.
4. $(\tilde{\mathbf{U}}, \tilde{\mathbf{V}}, \mathbf{Q}_A, \mathbf{Q}_B) \mapsto (\mathbf{Q}_A\tilde{\mathbf{U}}, \mathbf{Q}_B\tilde{\mathbf{V}})$, i.e. two $(n,k) \times (k,k)$ multiplications $\mathcal{O}(nk^2)$.

The total computational effort is linear in n.

(e) If $\mathbf{X}, \mathbf{Y} \in \mathbb{R}^{m,n}$ satisfy $\operatorname{rank}(\mathbf{X}) = \operatorname{rank}(\mathbf{Y}) = k$, show that $\operatorname{rank}(\mathbf{X}+\mathbf{Y}) \leq 2k$.

SOLUTION:

From $\operatorname{rank}(\mathbf{X}) = k$ it follows that there is a basis for the image of $\mathbf{X}$ with k basis vectors, i.e. each column of $\mathbf{X}$ is a linear combination of these k vectors. The same holds for $\mathbf{Y}$ with other k basis vectors, where some can be obtainable from linearcombinations of the basis vectors of $\mathbf{X}$. Hence, each column of $\mathbf{X}+\mathbf{Y}$ is a linear combination of *at most* $2k$ vectors. Thus, $\operatorname{rank}(\mathbf{X}+\mathbf{Y}) \leq 2k$.

(f) Consider $\mathbf{A}_X, \mathbf{A}_Y \in \mathbb{R}^{m,k}$, $\mathbf{B}_X, \mathbf{B}_Y \in \mathbb{R}^{n,k}$, $\mathbf{X} = \mathbf{A}_X\mathbf{B}_X^\top$ and $\mathbf{Y} = \mathbf{A}_Y\mathbf{B}_Y^\top$. Find a factorization of sum $\mathbf{X}+\mathbf{Y}$ as $\mathbf{X}+\mathbf{Y} = \mathbf{A}\mathbf{B}^\top$, with $\mathbf{A} \in \mathbb{R}^{m,2k}$ and $\mathbf{B} \in \mathbb{R}^{n,2k}$.

SOLUTION:

We can define $\mathbf{A} := [\mathbf{A}_X \ \mathbf{A}_Y] \in \mathbb{R}^{m,2k}$ and $\mathbf{B} := [\mathbf{B}_X \ \mathbf{B}_Y] \in \mathbb{R}^{n,2k}$.

Hence, we have that $\mathbf{A}\mathbf{B}^\top = [\mathbf{A}_X \ \mathbf{A}_Y] \begin{bmatrix} \mathbf{B}_X^\top \\ \mathbf{B}_Y^\top \end{bmatrix} = \mathbf{A}_X\mathbf{B}_X^\top + \mathbf{A}_Y\mathbf{B}_Y^\top = \mathbf{X} + \mathbf{Y}$.

(g) Implement the formula derived in the previous subproblems in an efficient C++ function

```
void lowRankApprox(const MatrixXd &Ax, const MatrixXd &Ay,
   const MatrixXd &Bx, const MatrixXd &By, MatrixXd &Az,
   MatrixXd &Bz),
```

where $\mathbf{A}_Z \in \mathbb{R}^{m,k}$ and $\mathbf{B}_Z \in \mathbb{R}^{n,k}$ are the two terms of the decomposition of $\mathbf{Z} = \mathbf{A}_Z\mathbf{B}_Z^\top$, the rank-$k$ best approximation of the sum $\mathbf{X} + \mathbf{Y} = \mathbf{A}_X\mathbf{B}_X^\top + \mathbf{A}_Y\mathbf{B}_Y^\top$:

$$\mathbf{Z} = \underset{\substack{\mathbf{M}\in\mathbb{R}^{m,n} \\ \operatorname{rank}(\mathbf{M})\leq k}}{\operatorname{argmin}} \left\| \mathbf{A}_X\mathbf{B}_X^\top + \mathbf{A}_Y\mathbf{B}_Y^\top - \mathbf{M} \right\|_F$$

Here $\mathbf{A}_X, \mathbf{A}_Y \in \mathbb{R}^{m,k}$ and $\mathbf{B}_X, \mathbf{B}_Y \in \mathbb{R}^{n,k}$.

SOLUTION:

The best low rank approximation of a matrix can be obtained by first calculating the SVD of the matrix and then extracting the first k columns of $\mathbf{U}$ and $\mathbf{V}$ and the upper left $k \times k$ submatrix of $\mathbf{\Sigma}$.

```cpp
void rank_k_approx(const MatrixXd &Ax, const MatrixXd &Ay,
                   const MatrixXd &Bx, const MatrixXd &By,
                   MatrixXd &Az, MatrixXd &Bz) {

    MatrixXd A(Ax.rows(), Ax.cols()+Ay.cols());
    A << Ax, Ay;
    MatrixXd B(Bx.rows(), Bx.cols()+By.cols());
    B << Bx, By;

    // U: m x 2k; S: 2k x 2k; V: n x 2k
    MatrixXd U, S, V;
    svd_AB(A, B, U, S, V);

    int k = Ax.cols();
    Az = U.leftCols(k) * S;
    Bz = V.leftCols(k);
}
```

(h) What is the asymptotic computational cost of the function `rank_k_approx` for a small k and $m = n \to \infty$?

SOLUTION:

`svd_AB` has linear complexity in n and m, as derived in (d). The multiplication of the truncated matrices $\mathbf{U}$ and $\mathbf{S}$ requires $\mathcal{O}(mk)$ operations (since S is diagonal). Hence, the total computational effort is $\mathcal{O}(n+m)$.

Problem 4.3: Face recognition by PCA

We apply principal component analysis to develop a recognition/classification technique for pictures of faces.

Template: `eigenfaces.cpp`

In the folder `basePictures`, you can find $M = 15$ photos encoded with the PGM (Portable GrayMap) ASCII format. They are essentially represented by an $h \times w$ matrix of integers between 0 and 255 (in our case $h = 231$, $w = 195$).

(a) Through the `load_png` function the main function of `eigenfaces.cpp` loads every picture as a flattened EIGEN vector of length hw. Compute the mean of all these vectors, and insert each vector minus the mean as a column of a matrix A of size $hw \times M$.

SOLUTION:

```
    for (int i=0; i<M; i++) {
        string filename = "./basePictures/subject"+to_string(i+1)+".pgm";
        VectorXd flatPic = load_pgm(filename);
        faces.col(i) = flatPic;
        meanFace += flatPic;
    }

    // compute mean face and subtract it from all faces
    meanFace /= M;
    faces.colwise() -= meanFace;
```

The covariance matrix of A is defined as $AA^\top$. The eigenvector of $AA^\top$ corresponding to the largest eigenvalue represents the direction along which the dataset has the maximum variance. Therefore it encodes the features which differ the most among faces. For this reason, since our interest is in recognizing faces from their salient features, we want to compute the eigenvectors of $AA^\top$. We rename such eigenvectors as *eigenfaces* (usually they are known as principal components).

(b) What is the size of $AA^\top$? How many non-zero eigenvalues can the matrix $AA^\top$ have at most?

Hint: see Point (a) of Problem 4.2.

SOLUTION:

The size of $AA^\top$ is $hw \times hw$. By Point (a) of Problem 4.2 the rank of A is at most M, therefore $AA^\top$ can have at most M non-zero eigenvalues (counted with their multiplicities).

(c) What is the size of $A^\top A$? How are the eigenvalues and eigenvectors of the matrix $AA^\top$ related respectively to the eigenvalues and eigenvectors of $A^\top A$, and to the singular values and singular vectors of A?

SOLUTION:

The size of $A^\top A$ is $M \times M$. The eigenvalues of $AA^\top$ and $A^\top A$ are the same and they are the squared singular values of A. The eigenvectors of $AA^\top$ are the left singular vectors of A, while the eigenvectors of $A^\top A$ are the right singular vectors of A.

(d) Use the characterization of the previous point to compute with a C++ code the eigenvectors of $AA^\top$ using the SVD of A.

SOLUTION:

```
    JacobiSVD<MatrixXd> svd(faces, ComputeThinU | ComputeThinV);
    MatrixXd U = svd.matrixU();
```

(e) Given a new face y, implement C++ code which computes its projection on the space spanned by the eigenfaces, that is finds x such that $Ux = (y-$ mean face$)$, where U is the matrix of singular vectors of A.

Hint: instead of solving the linear system, use the properties of the matrix U.

SOLUTION:

```
    MatrixXd projFaces = U.transpose() * faces;
    VectorXd projNewFace = U.transpose() * (newFace - meanFace);
```

(f) Implement C++ code which computes the distance between the projection of the new face and the projection of a column k for a generic k, and print the k which minimizes the distance.

SOLUTION:

```
    int indexMinNorm;
    (projFaces.colwise()-projNewFace).colwise().norm().minCoeff(&indexMinNorm);
    cout << testPicName << " is identified as subject "
         << indexMinNorm+1 << endl;
}
```

(g) Test your code from the previous steps to try to recognize the pictures in the folder `testPictures`.