# Software and Programming of High Performance Systems

*P. Hadjidoukas, E. Dermatas, S. Gallopoulos*

Winter semester 2022-23

# Set 2 - PCA

Issued: November 21, 2022

## Question 1: Principal Component Analysis

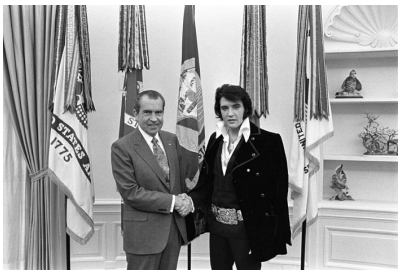
Principal Component Analysis (PCA) is a mathematical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components.

For this exercise you will implement PCA and then use it for compressing a gray-scale image. An image is represented as a 2-D matrix of double-precision numbers and stored row-wise as a compressed binary file. The code does not need to uncompress the files; instead it directly reads the compressed data by means of the zlib library.

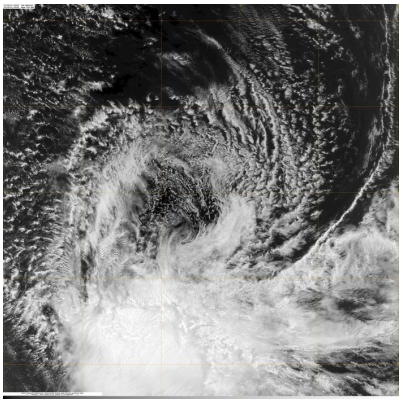
An incomplete source code implementation of PCA is provided in `pca_omp_lapack/pca.cpp`. You will have to implement the parts annotated with `TODO`. An example that uses uses the `dsyev_()` LAPACK routine to compute the eigenvalues of a matrix has been already provided.

We provide three such binary files with varying size:

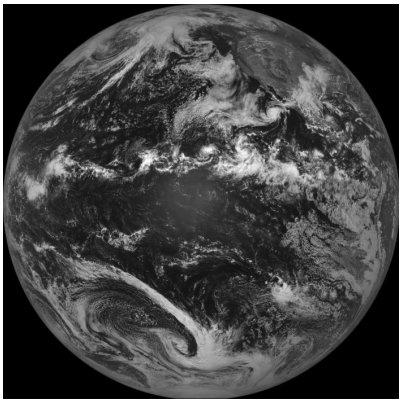
- `elvis.bin.gz`: the well-known Elvis-Nixon image ($469 \times 700$ pixels).
- `cyclone.bin.gz`: image of a tropical cyclone ($4096 \times 4096$ pixels), downloaded from the NASA Earth Observatory (cyclone).
- `earth.bin.gz`: image of planet earth ($9500 \times 9500$ pixels), downloaded from the NASA Earth Observatory (earth).



(a) The King meets with president Nixon ($469 \times 700$)



(b) Tropical cyclone image from NASA ($4096 \times 4096$)



(c) Our planet ($9500 \times 9500$)

The binary files for the corresponding images can be found in the `data` directory in the course Git repository (as well as on the course webpage). In addition, you will find a Matlab script

file (`pca_demo.m`) that uses the Elvis-Nixon image and demonstrates the implementation steps that you have to follow for your C/C++ program. The code in the script is not optimized and its main purpose is to provide guidelines and help you in the verification of your results.

The first stage of PCA includes the construction of the covariance matrix. The program initially reads the image matrix from the disk. Then, it computes the mean and standard deviation of the features (image columns) and appropriately fills the covariance matrix $C$. Next, the program determines the eigenvalues and eigenvectors of the covariance matrix $C$. For this purpose you will use the `dsyev_()` LAPACK routine. Finally, the program computes the first $K$ principal components.

a) Implement the PCA algorithm as described above. Apply PCA to the Elvis & Nixon test image and use the first $K = 1, 30, 50, 100$ principal components to get a series of compressed representations of the image. Compute and report the achieved compression ratio (your compressed data should include all the necessary information for a successful reconstruction). Start with the skeleton code in `pca.cpp`.

   In addition to the course material, you might find useful the provided Matlab script and the "Data Compression" section of the following webpage: putting-pca-to-work. The corresponding implementation of PCA in Matlab is available in the `pca_matlab` directory.

b) Add multithreading capabilities to your PCA implementation and measure its performance for the all the test images. For this question you do not have to reconstruct and save the final image. Report the execution time of your program without including the time spent for reading the input image. How does your PCA implementation scale?

## Question 2: Forces with SIMD and OpenMP

In this question you will have to improve the code that computes *anti*gravitational forces in the system of $N$ planets in the 3D space. The force experienced by the $i$-th planet due to the $j$-th planet is given by the following expression:

$$\mathbf{F}_{ij} = G\frac{m_i m_j(\mathbf{r}_i - \mathbf{r}_j)}{\|\mathbf{r}_i - \mathbf{r}_j\|^3} \tag{1}$$

where $\mathbf{r} = (x, y, z)$ is the 3D coordinate vector, $m$ is mass, $G = 6.67408 \times 10^{-11}$ is the gravitational constant and $\|\cdot\|$ denotes the length of a vector. Note the sign of the expression.

Therefore the total force on the $i$-th planet is

$$\mathbf{F}_i = \sum_{\substack{j=1 \\ j\neq i}}^{N} \mathbf{F}_{ij} = \sum_{\substack{j=1 \\ j\neq i}}^{N} G\frac{m_i m_j(\mathbf{r}_i - \mathbf{r}_j)}{\|\mathbf{r}_i - \mathbf{r}_j\|^3} \tag{2}$$

In `forces_omp_simd/vanilla.cpp` you will find the function
**void** computeGravitationalForces( Particles & particles ) implementing Eq. 2 for all the particles in a naive way. The only parameter of the function, structure `Particles`, is defined in the file `particles.h` and consists of arrays of particle coordinates, masses and forces. The function is passed into the testing machinery that loads a few tests and checks the speed and correctness of the solution.

Your task is to write a *faster* function that computes the same forces. You can only modify **void** computeGravitationalForcesFast( Particles & particles ) in `forces_omp_simd/fast.cpp`.

Use the naive implementation as a starting point and vectorize the code with AVX (or SSE) and parallelize it with OpenMP. Modify the `Makefile` so that it compiles your code.

**Hint:** *AVX double-precision comparison is handled by the following intrinsic:*
`__m256d mask = _mm256_cmp_pd(__m256d a, __m256d b, int cmp).`
`cmp` *argument may take the following values:* `_CMP_EQ_OQ` *(Equal),* `_CMP_NEQ_OQ` *(Not-equal),* `_CMP_LT_OQ` *(Less-than),* `_CMP_LE_OQ` *(Less-than-or-equal),* `_CMP_GT_OQ` *(Greater-than),* `_CMP_GE_OQ` *(Greater-than-or-equal).*

*This intrinsic performs a SIMD compare of the packed double-precision floating-point values in the first and second operands and returns the results of the comparison. The result of each comparison is 8 bytes of all 1s (comparison true) or all 0s (comparison false). The return value* `mask` *is a SIMD vector of 4 comparison results.*