Scientific Computing Project
Subspace iteration methods

by Sofiane Fraine
and Felix Foucher de Brandois

Digital Sciences Department - First year 2022-2023

Contents

List of Figures

1 Introduction

Matrix reduction is a mathematical technique that simplifies complex data by eliminating redundant or unimportant information. Principal Component Analysis (PCA) is a dimension reduction method that uses spectral decomposition of the variance/covariance matrix. However, the full spectral decomposition is not required, and only the leading eigenpairs are sufficient to provide necessary information about the data.

One approach to compute the leading eigenpairs is to use the power method, coupled with a deflation process. However, this algorithm may not be efficient in terms of performance.

In this project, we will explore a more efficient method called the subspace iteration method, which is based on an object called Rayleigh quotient. We will examine four variants of this method.

2 Limitations of the power method

Comparison between the method power_v11 and the eig method

Question 1 : We have compared the running time of the `power_v11` to compute a few eigenpairs with the running time of the function `eig` of Matlab.

Table 1: Results of the comparison between both methods (`eig` and `power_v11`)

Dimension and Type		Eig		Basic Power method(v_11)	
		Time	Eigenpairs' quality	Time	Eigenpairs' quality
1000x1000	Type 1	$5.00 \times 10^{-1}s$	$[0.000, 1.881 \times 10^{-13}]$	$\emptyset$	no convergence
	Type 2	$4.02 \times 10^{-1}s$	$[5.862 \times 10^{-16}, 1.442 \times 10^{-6}]$	$4.6s$	$[2.921 \times 10^{-16}, 5.128 \times 10^{-14}]$
	Type 3	$4.00 \times 10^{-1}s$	$[0.000, 1.175 \times 10^{-11}]$	$1.432 \times 10^{2}s$	$[0.000, 8.327 \times 10^{-15}]$
	Type 4	$4.80 \times 10^{-1}s$	$[0.000, 1.631 \times 10^{-14}]$	$\emptyset$	no convergence
200x200	Type 1	$3.00 \times 10^{-2}s$	$[0.000, 8.504 \times 10^{-14}]$	$6.08s$	$[0.000, 1.975 \times 10^{-14}]$
	Type 2	$4.00 \times 10^{-2}s$	$[0.000, 1.119 \times 10^{-7}]$	$1.40 \times 10^{-1}s$	$[1.460 \times 10^{-16}, 1.617 \times 10^{-15}]$
	Type 3	$4.00 \times 10^{-2}s$	$[0.000, 5.623 \times 10^{-12}]$	$2.60 \times 10^{-1}s$	$[1.176 \times 10^{-16}s, 1.332 \times 10^{-15}]$
	Type 4	$4.00 \times 10^{-2}s$	$[0.000, 4.510 \times 10^{-14}]$	$4.97s$	$[0.000, 1.954 \times 10^{-14}]$
100x100	Type 1	$1.00 \times 10^{-2}s$	$[0.000, 9.178 \times 10^{-15}]$	$4.40 \times 10^{-1}s$	$[0.000, 9.948 \times 10^{-15}]$
	Type 2	$2.00 \times 10^{-2}s$	$[0.000, 1.017 \times 10^{-7}]$	$5.00 \times 10^{-2}s$	$[4.381 \times 10^{-16}, 1.595 \times 10^{-15}]$
	Type 3	$2.00 \times 10^{-2}s$	$[0.000, 4.976 \times 10^{-12}]$	$3.00 \times 10^{-2}s$	$[0.000, 8.882 \times 10^{-16}]$
	Type 4	$2.00 \times 10^{-2}s$	$[0.000, 2.235 \times 10^{-14}]$	$3.50 \times 10^{-1}s$	$[0.000, 9.970 \times 10^{-15}]$
50x50	Type 1	$1.00 \times 10^{-2}s$	$[0.000, 9.992 \times 10^{-15}]$	$9.00 \times 10^{-2}s$	$[1.480 \times 10^{-16}, 4.832 \times 10^{-15}]$
	Type 2	$1.50 \times 10^{-1}s$	$[0.000, 1.038 \times 10^{-7}]$	$8.00 \times 10^{-2}s$	$[1.227 \times 10^{-15}, 1.849 \times 10^{-15}]$
	Type 3	$1.00 \times 10^{-2}s$	$[1.132 \times 10^{-16}, 1.042 \times 10^{-12}]$	$2.00 \times 10^{-2}s$	$[2.809 \times 10^{-16}, 7.772 \times 10^{-16}]$
	Type 4	$5.00 \times 10^{-2}s$	$[0.000, 4.493 \times 10^{-14}]$	$1.40 \times 10^{-1}s$	$[0.000, 4.485 \times 10^{-15}]$
10x10	Type 1	$2.00 \times 10^{-2}s$	$[0.000, 3.331 \times 10^{-15}]$	$3.00 \times 10^{-2}s$	$[0.000, 8.882 \times 10^{-16}]$
	Type 2	$1.00 \times 10^{-2}s$	$[0.000, 4.604 \times 10^{-8}]$	$3.00 \times 10^{-2}s$	$[1.311 \times 10^{-16}, 1.311 \times 10^{-16}]$
	Type 3	$1.00 \times 10^{-2}s$	$[0.000, 5.199 \times 10^{-12}]$	$4.00 \times 10^{-2}s$	$[0.000, 0.000]$
	Type 4	$1.00 \times 10^{-2}s$	$[0.000, 2.168 \times 10^{-14}]$	$7.00 \times 10^{-2}s$	$[1.247 \times 10^{-16}, 7.772 \times 10^{-16}]$

We notice that the execution time of Matlab's `eig` method is almost constant whatever the size and the type of matrices. On the contrary, the method `power_v11` depends on the type of the matrix. For example, for the size of array 200×200, there is a ratio 10 between the computation time of the eigenpairs between type 1 and type 2. On average, the `eig` method is 10 times faster than the `power_v11` method. Moreover, the method `power_v11` reaches its limits in terms of convergence size. For instance, we can notice that for the size of array 1000×1000, there are no convergence for types of array 1 and 4 whereas it converges slowly especially for type 3.
Finally, regarding the quality of eigenpairs criteria, it seems that both methods provides similar results.

Implementation of power_v12 method

Question 2 : We rearranged the operations in the algorithm proposed for the power method so that only one *matrix* $\times$ *vector* operation remains in the loop.

```
Algorithm 1 Vector power method V12
Input: Matrix $A \in \mathbb{R}^{n \times n}$, vector $v \in \mathbb{R}^n$
Output: Largest (in module) eigenpair $(\lambda_1, v_1)$
 1: $z = A \cdot v$
 2: $\beta = v^\top \cdot z$
 3: repeat
 4:     $y = z$
 5:     $v = y/\|y\|$
 6:     $z = A \cdot v$
 7:     $\beta_{old} = \beta$
 8:     $\beta = v^\top \cdot z$
 9: until $|\beta - \beta_{old}|/|\beta_{old}| < \epsilon$
10: $\lambda_1 = \beta$ and $v_1 = v$
```

We implemented this new algorithm in a Matlab file (`power_v12`) and compared it to the first version :

Table 2: Comparisons of the execution times of `power_v11` and `power_v12`

Dimension and type		Power method v11	Power method v12
200x200	Type 1	$4.35s$	$2.73s$
	Type 2	$1.10 \times 10^{-1}s$	$8.00 \times 10^{-2}s$
	Type 3	$2.70 \times 10^{-1}s$	$3.60 \times 10^{-1}s$
	Type 4	$4.59s$	$2.85s$

We have checked that the `power_v12` method is two times faster than `power_v11`

Main drawback of the deflated power method

Question 3 : There are several drawbacks to this method. Firstly, the computation time for obtaining eigenpairs can vary greatly depending on the type of matrix, even for matrices of the same size. Additionally, the computation time for certain eigenpairs may be higher than that for all the eigenvalues obtained using the eig function. Furthermore, when using type 1, if the matrix size is too large (e.g. 1000 x 1000), the method fails to converge.

3 Extending the power method to compute dominant eigenspace vectors

Our objective is to extend the power method to compute a block of dominant eigenpairs. The basic version of the subspace iteration method computes the eigenvectors associated with the m largest (in module) eigenvalues of a symmetric matrix A, given a set of m orthonormal vectors V.

Convergence of the matrix V

Question 4 : When we try to apply the algorithm 1 to a set of m vectors, it converges to a matrix V whose m columns correspond to the eigenvector of a same eigenvalue and not m eigenvectors associated to different eigenvalues !
We verified the conjecture by applying the algorithm to a matrix V and we observed that the output vectors were the same with the exception of the sign: they are therefore associated to the same eigenvalue. This is prevented by the orthonormalisation step in the `subspace` algorithm.

Computation of the whole spectral decomposition

Question 5 : We have : $A \in \mathbb{R}^{n\times n}$ where n is a large number. Computing the full spectral decomposition of this matrix can be very computationally expensive and the variants of the power method are used to avoid computing it.
However : $V \in \mathbb{R}^{n\times m}$ and $H = V^\top \cdot A \cdot V \implies H \in \mathbb{R}^{m\times m}$.
Since $m << n$, computing the full spectral decomposition of H is much more efficient than computing the full spectral decomposition of A.

Identification of the steps of the Raleigh-Ritz algorithm

Several modifications are needed to make the basic subspace iteration an efficient code.

Question 7 : The following algorithm is used in `subspace_iter_v1`. Above the algorithm, the red text refers to the localization of the different steps and operations into the Matlab program `subspace_iter_v1`.

Algorithm 2 Subspace iteration method v1 with Raleigh-Ritz projection

```
Input: Symmetric matrix $A \in \mathbb{R}^{n\times n}$, tolerance $\epsilon$, max nb of iterations $MaxIter$, and
    target percentage of the trace of $A$, $PercentTrace$
Output: Dominant eigenvectors $V_{out}$ and corresponding eigenvalues $\Lambda_{out}$
1: Generate an initial set of $m$ orthonormal vectors $V \in \mathbb{R}^{n\times m}$ line 48-49
   $k = 0$ line 38
   $PercentReached = 0$
2: repeat
3:     $k = k + 1$ line 54
4:     Compute $Y$ such that $Y = A \cdot V$ line 56
5:     $V \leftarrow$ orthonormalization of the columns of $Y$ line 58
6:     $Rayleigh - Ritz\ projection$ applied on matrix $A$ and orthonormal vectors $V$ line 61
7:     $Convergence\ analysis\ step$ : save eigenpairs that have converged and update
       $PercentReached$ line 63-117
8: until ($PercentReached > PercentTrace$ or $n_{ev} = m$ or $k > MaxIter$) line 44
```

Toward an efficient solver

Two ways of improving the efficiency of the solver are proposed. Our aim is to build an algorithm that combines both the block approach and the deflation method in order to speed-up the convergence of the solver.

Cost of computations in subspace_iter_v2 method

Orthonormalisation is performed at each iteration and is quite costly. One simple way to accelerate the approach is to perform p products at each iteration (replace $V = A \cdot V$ (first step of the iteration) by $V = A^p \cdot V$).

Question 8 : Let $A \in \mathbb{R}^{m \times p}$ and $B \in \mathbb{R}^{p \times n}$.
To determine the number of flops required for the product between these two matrices, we note that there are $n \times m$ elements in the output matrix. Each element is obtained by taking the dot product of a row of A with a column of B, which involves p multiplications and $p - 1$ additions. Therefore, the number of operations for one element in the output matrix is $2p - 1$ flops.
The total number of flops required for the matrix product is : $n \times m \times (2p - 1)$.

In our algorithm, we have $A \in \mathbb{R}^{n \times n}$ and $V \in \mathbb{R}^{n \times m}$.
The cost of computing A^p is $(p - 1) \times n^2 \times (2n - 1) \approx 2(p - 1)n^3$.
The cost of computing $A^p \cdot V$ is $(p - 1) \times n^2 \times (2n - 1) + n \times m \times (2n - 1) \approx 2(p - 1)n^3$.

If we compute $V = A \cdot V$ p times (i.e., computing $A^p \cdot V$ starting from the right), the number of flops required is $(p - 1) \times n \times m \times (2n - 1) \approx 2(p - 1)mn^2 << 2(p - 1)n^3$.

Moreover, A and p aren't modified in the loop. Therefore, if we compute A^p at the beginning of the program, and just calculate, the number of operations for $A^p \cdot V$ each round is $n \times m \times (2n - 1)$.

Behaviour of the block approach

Question 10 : The following table shows the results of the filet `test_v0v1v2` which highlights the different numbers of iterations of `subspace_iter_v0`, `subspace_iter_v1` and `subspace_iter_v2` by increasing the number p for different sizes and types of arrays. We can notice two distincts phenomemenons.

- The number of iterations decreases and then increases if p is too high.
- Eigenvectors' quality increases proportionnaly to p and is better than the versions v_0 and v_1

This is explained by the fact that in `subspace_iter_v1`, at each turn in the loop, we calculate $V = A \times V$. In `subspace_iter_v2`, we compute directly $V = A^p \cdot V$. So, for the same number of iterations, the quality of the eigenvectors is improved.

Dimension and type	Value of p	subspace_iter_v2
200*200 Type 1	none	1699 for v0, 403 for v1
	1	403
	3	135
	5	81
	7	58
	9	45
50*50 Type 1	none	for v0, for v1
	1	1
	3	1
	5	1
	7	2
	9	4
	11	9
	13	9194
	15	no convergence

Difference of accuracy in subspace_iter_v1

Question 11 : Vectors with the largest eigenvalue converge faster.
In `subspace_iter_v1`, the vectors considered as having converged (convergence criteria having been passed) continue to be updated. At the output of the algorithm, all the vectors will not have the same accuracy.

Accuracy of eigenpairs in subspace_iter_v3

Because the columns of V converge in order, we can freeze the converged columns of V. Suppose the first columns of V have converged, and partition $V = [V_c, V_{nc}]$ where V_c correspond to the vectors that have converged, and V_{nc}, the vectors that have not converged.

Question 12 : The eigenvectors convergence is expected to be faster but may be less precise, as we are only focusing on determining the yet unknown eigenpairs :
In the algorithm of `subspace_iter_v3`, there is no more operations on the vectors that have converged V_c. It reduces the total number of calculus but the vectors that have converged aren't refined anymore.

4 Numerical experiments

Eigenvalue distribution of four types of matrices

Question 14 : We have highlighted the differences between the 4 types of matrices :

- Type 1 : eigenvalues ranging from 1 to n.
- Type 2 : random eigenvalues
- Type 3 : cond**(-(i-1)/(n-1)) avec cond = 1e5

- Type 4 : 1 - ((i-1)/(n-1))*(1 - 1/cond) avec cond = 1e2

The 4 types of matrices that we decided to use for our tests have eigenvalues that are respectively incremented from 1 to n, random, with their logarithm uniformly distributed, and finally uniformly distributed between 0 and 1.

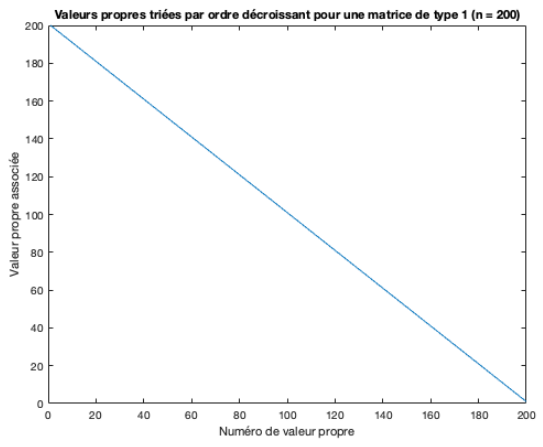

Figure 1: Distribution of eigenvalues for a type 1 matrix of size 200 × 200

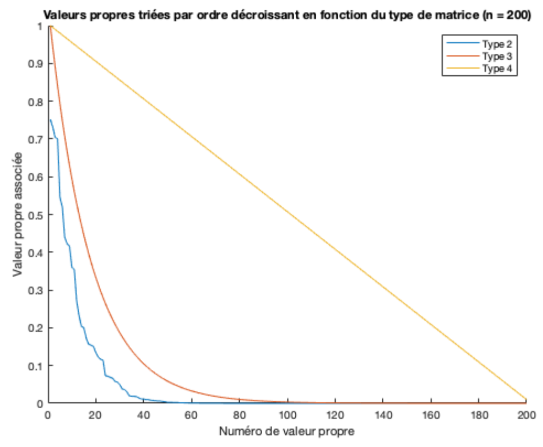

Figure 2: Distribution of eigenvalues for a type 2, 3 and 4 matrix of size 200 × 200

Dimension and type		subspace_iter_v0	subspace_iter_v1	subspace_iter_v2
1000*1000	Type 1	$1.1 \times 10^2 s$	no convergence	no convergence
	Type 2			
	Type 3			
	Type 4			
200*200	Type 1	5.9s	1.92s	$2.5 \times 10^{-1} s$
	Type 2	19.9s	$8.00 \times 10^{-2} s$	$9.00 \times 10^{-2} s$
	Type 3	1.1s	$1.10 \times 10^{-1} s$	$7.00 \times 10^{-2} s$
	Type 4			
100*100	Type 1	1.5s	$2.00 \times 10^{-1} s$	$6.00 \times 10^{-2} s$
	Type 2	$1.00 \times 10^{-1} s$	$8.00 \times 10^{-2} s$	no convergence
	Type 3	$4.20 \times 10^{-1} s$	$8.00 \times 10^{-2} s$	$8.00 \times 10^{-2} s$
	Type 4			

Comparison of the performance of the implemented algorithms

Question 15 : We have compared the performances of the algorithms implemented as well as those provided (`eig`) for different types and sizes of matrix.

The results ot the table shows that the version v_1 and v_2 increases the computing's speed.

5 Image Compression

An image can be described by a matrix I of size $q \times p$. To perform an image compression, we can use the k-low-rank approximation of I, I_k.
In order to generate the matrix I_k :

- We create a matrix $M = II^\top$ (or $M = I^\top I$ if $p < q$)
- We find the k eigenpairs of M
- We create the matrix Σ_k with the k eigenvalues
- We create the matrix U_k with the k eigenvectors (or V_k)
- We create the matrix V_k by using the relation between the vectors of U_k and those of V_k (or the matrix U_k with V_k)
- $I_k = U_k \cdot \Sigma_k \cdot V_k^\top$

Size of the elements in Singular Value Decomposition

Question 1 : We have : $I \in \mathbb{R}^{q \times p}$. and σ_i the singular values of A in descending order(the square roots of the eigenvalues of $A^\top A$ and $AA^\top$).

- If $q < p : M = II^\top$

  $M \in \mathbb{R}^{q\times q}$

  $U \in \mathbb{R}^{q\times q}$ is formed of q orthonormal eigenvectors associated to q eigenvalues of $AA^\top$

  $\implies U_k$ is formed of k orthonormal eigenvectors associated to the k most dominant eigenvalues.

  $\implies U_k \in \mathbb{R}^{q\times k} = (u_1 \ u_2 \ ... \ u_k)$

  $v_i = \dfrac{1}{\sigma_i} I^\top u_i \in \mathbb{R}^{p\times 1}$

  $\implies V_k \in \mathbb{R}^{p\times k} = (v_1 \ v_2 \ ... \ v_k)$

  Let : $\Sigma_k = Diag(\sigma_1, ..., \sigma_k)$
  We have : $I_k = \sum_{i=1}^{k} \sigma_i u_i v_i^\top = U_k \cdot \Sigma_k \cdot V_k^\top$
  $\implies \Sigma_k \in \mathbb{R}^{k\times k}$

- If $p < q : M = I^\top I$
  Same principle but we use the following equation : $u_i = \frac{1}{\sigma_i} I v_i \in \mathbb{R}^{q\times 1}$

  We get the same dimensions for the matrices : $\Sigma_k \in \mathbb{R}^{k\times k}$; $U_k \in \mathbb{R}^{q\times k}$; $V_k \in \mathbb{R}^{p\times k}$

6 References

https://math.stackexchange.com/questions/3512976/proof-of-of-flops-in-matrix-multiplication : Number of flops in a matrix multiplication FAIRE UNE REFERENCE