Probabilistic algorithms for constructing approximate matrix decompositions

Quentin Aristote

January the 24th, 2021

1. Introduction

Given an $m \times n$ matrix A, the problem of *low-rank matrix factorization* consists in finding matrices B and C such that

$$A = BC \tag{1}$$

The hope is that the matrices B and C have dimensions $m \times k$ and $k \times n$ with k small compared to m and n, so that A can be stored, multiplied by and more generally manipulated more efficiently. An example of how this may be useful is multpliying a square matrix by A : when A is not factorized, it costs around $O(m^2 \times n)$ depending on the chosen algorithm. This cost becomes huge when dealing with increasingly large matrices.

Of course, in the factorization (1), the smallest value for k that can be found is the rank of A. Therefore when A is invertible or almost invertible this factorization will not lead to very big differences in computational times. We instead tackle the problem of *low-rank matrix approximation*, which consists in finding matrices B and C with dimensions $m \times k$ and $k \times n$ that give a good compromise between having k be small before n and m and the approximation error $||A - BC||_F$ ($||\cdot||_F$ denoting the Frobenius norm) being small.

This document gives an overview of how this problem can be solved using probabilistic techniques, themselves derived from more costly deterministic techniques. These algorithms were first introduced in the article *Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions* [1]. Section 2 introduces the theory behind them and tries to show how their step-by-step development is natural, while section 3 introduces an OCaml library that implements them.

2. Theory

2.1. Deterministic matrix factorization

Let us first give a rapid overview of different matrix factorization methods that we will use to derive algorithms for probabilistic matrix approximation.

2.1.1. QR factorization

The most basic one is the *QR factorization* : given an $m \times n$ matrix A, it is possible to find an $m \times n$ matrix Q with orthogonal columns and an $n \times n$ upper triangular matrix R such that $A = QR$. This can be done using the Gram-Schmidt process in $O(mn^2)$. We will rapidly recall how to build Q with this process, as it will be further used in an algorithm in section 2.2.

Let us write $A = [a_1, \dots, a_n]$ in columns. Finding Q amounts to finding an orthogonal basis for $\langle a_1, \dots, a_n \rangle$, the subspace generated by $a_1, \dots, a_n$. We can thus first choose $q_1 = a_1 / ||a_1||$, then compute q_2 by renormalizing the orthogonal projection of a_2 on $\langle q_1 \rangle^\perp$ and so on, computing q_k by renormalizing the orthogonal projection of a_k on $\langle q_1, \dots, q_{k-1} \rangle^\perp$, that is the part of subspace that is left to be generated by our target orthogonal family. The resulting matrix Q is then $[q_1, \dots, q_n]$. In practice, there are two ways to do this computation : one may either compute the q_k *consecutively*, using the formulæ

$$\begin{aligned} u_1 &= a_1 & q_1 &= \tfrac{u_1}{||u_1||} \\ &\dots \\ u_k &= a_k - [q_1, \dots, q_{i-1}][q_1, \dots, q_{i-1}]^* a_k & q_k &= \tfrac{u_k}{||u_k||} \\ &\dots \end{aligned}$$

or *dynamically*, using the formulæ

$$\begin{aligned} [u_1^{(1)}, \dots, u_n^{(1)}] &= [a_1, \dots, a_n] & q_1 &= \frac{u_1^{(1)}}{\left|\left|u_1^{(1)}\right|\right|} \\ &\dots \\ [u_k^{(k)}, \dots, u_n^{(k)}] &= [u_k^{(k-1)}, \dots, u_n^{(k-1)}] - q_{k-1} q_{k-1}^* [a_k, \dots, a_n] & q_k &= \frac{u_k^{(k)}}{\left|\left|u_k^{(k)}\right|\right|} \\ &\dots \end{aligned}$$

since in both cases, we end up with

$$q_k = \frac{a_k - \sum_{i=1}^{k-1} (a_k, q_i) q_i}{\left|\left| a_k - \sum_{i=1}^{k-1} (a_k, q_i) q_i \right|\right|}$$

Also note that those two techniques may be used alternatively, for example computing an orthogonal family with the dynamic technique and then adding an additional vector to the family using the consecutive technique.

2.1.2. Singular value decompositions

The *singular value decomposition* (SVD) and its variants is probably the most-used method to factorize a matrix. Given an $m \times n$ matrix A of rank r, it is possible to find $m \times r$ and $r \times n$ matrices U and V with orthogonal columns and a diagonal matrix $\Sigma = \mathrm{diag}(\sigma_1, \dots, \sigma_r)$ of size r whose diagonal elements (the *singular values*) are all non-zero, and such that

$$A = U \Sigma V^*$$

Computing this decomposition takes $O(mn^2)$ time. The SVD may be used to solve the problem of matrix approximation deterministically by keeping only the singular values with highest module : if $|\sigma_1| \geq \ldots \geq |\sigma_r|$, taking $B = U$ and $C = \text{diag}(\sigma_1, \ldots, \sigma_k)V^*$ yields the matrix BC that minimizes the error $||A - BC||_2$ among all rank k matrices : this method is called the *principal component analysis* (PCA).

If A is a square matrix of size n, U and V can be chosen to be equal : this is called the *eigenvalue decomposition*, and the singular values are now called *eigenvalues*. However, the resulting eigenvalues $(\sigma_1, \ldots, \sigma_r)$ may be non-real, even if A itself is real. The eigenvalues are guaranteed to be real when A is hermitian, that is when $A^* = A$.

2.1.3. Factorizing a matrix using its range

There are many more kinds of matrix factorizations but they all usually involve matrices with orthogonal columns as the leftmost factor. This means that if we can find an $m \times k$ matrix Q with orthogonal columns such that $A = QQ^*A$, factorizing A can be done more efficiently the smaller k is. For example, one may first compute the SVD of $Q^*A = U\Sigma V^*$ in $O(kn^2)$ time, yielding the SVD $A = (QU)\Sigma V^*$ after an additional matrix multiplication taking $O(mkn)$ time, for an overall complexity of $O(kn \min(m, n))$ time.

One may also solve the low-rank factorization problem using this technique : if we can find an $m \times k$ matrix Q such that $||A - QQ^*A||_2 \leq \epsilon$, and if $Q^*A = U\Sigma V^*$, then we get

$$||A - (QU)\Sigma V^*||_2 = ||A - QQ^*A||_2 \leq \epsilon$$

A good candidate for the matrix Q case is a matrix that captures (or approximately captures) the range of A, that is such that its columns generate the range of A (because QQ^* is then the orhtogonal projector on the range of A). This is the case of the matrix Q given by the QR factorization. Similarly, the $m \times k$ matrix Q with orthogonal columns that minimizes $||A - QQ^*A||_2$ among all matrices of rank k can be computed using a PCA. But finding Q these ways doesn't help because it means the overall complexity of the chosen decomposition stays the same.

This is not surprising, as finding such a matrix Q is itself an instance of the low-rank factorization problem (or the low-rank approximation problem). Fortunately, it is also possible to find good candidates for Q (good approximations of the range of A) more efficiently, using probablistic methods. This is the topic of section 2.2.

2.2. Range finding

We now seek to find matrices Q of fixed dimensions $m \times k$ such that $||A - QQ^*A||_F$ is small, with the intuition that Q should approximate the range of A. Note that the results described here also work with the spectral norm instead of the Frobenius norm ; since the original article [1] uses the spectral norm in its proofs we use the Frobenius norm here.

2.2.1. Randomized range finder

The first method we can use to find such a Q is the *randomized range finder* :

- sample $k \leq n$ random vectors $\omega_1, \ldots, \omega_k$ of size n and form the matrix $\Omega = [\omega_1, \ldots, \omega_k]$;
- derive a generating family $y_1, \ldots, y_k$ of an approximation of the range of A by taking $[y_1, \ldots, y_k] = Y = A\Omega$;
- make this family orthogonal by choosing $Q = [q_1, \ldots, q_k]$ as in the QR factorization $Y = QR$.

It is then possible to give an upper bound on the error $||A - QQ^*A||_F$ using only Ω and A.

Let us first write the (full-size) SVD $A = U\Sigma V^*$ with U a unitary of size m, Σ a diagonal matrix of dimensions $m \times n$ and V of size $n \times n$. Assume that $\Sigma = \mathrm{diag}(\sigma_1, \ldots, \sigma_n)$ is such that $|\sigma_1| \geq \ldots \geq |\sigma_n|$. Write

$$\Sigma V^*\Omega = \begin{pmatrix} \Sigma_1 & \\ & \Sigma_2 \end{pmatrix} \begin{pmatrix} V_1^* \\ V_2^* \end{pmatrix} \begin{pmatrix} \Omega_1 \\ \Omega_2 \end{pmatrix}$$

with Σ_1 of size $k \times k$ and assume it is invertible, that is $|\sigma_k| \geq 0$. Finally, assume that the rows and columns of Ω_1 are linearly independant. Then, since the Frobenius norm is invariant under unitaries,

$$\begin{aligned} ||A - QQ^*A||_F^2 &= ||U^*(U\Sigma V^* - QQ^*U\Sigma V^*)||_F^2 \\ &= ||\Sigma V^* - (U^*Q)(U^*Q)^*\Sigma V^*||_F^2 \\ &= \left|\left|\tilde{A} - \tilde{Q}\tilde{Q}^*\tilde{A}\right|\right|_F^2 \end{aligned}$$

where we write $\tilde{A} = \Sigma V^*$, $\tilde{Q} = U^*Y$ and $\tilde{Y} = U^*Y$. Note that $\tilde{Q}R = \tilde{Y}$ and therefore $\tilde{Q}$ is just an orthogonalization of $\tilde{Y}$, therefore $\tilde{Q}\tilde{Q}^*$ is none other than $P_{\tilde{Y}}$, the orthogonal projection on the range of $\tilde{Y}$, which is itself an approximation of the range of $\tilde{A}$. But the range of $\tilde{Y}$ itself contains the range of

$$Z = \begin{pmatrix} I \\ \Sigma_2 V_2^* \Omega_2 (V_1^*\Omega_1)^\dagger \Sigma_1^{-1} \end{pmatrix} = \tilde{Y}(V_1^*\Omega_1)^\dagger \Sigma_1^{-1}$$

(with $(V_1^*\Omega_1)^\dagger$ the pseudo-inverse of $V_1^*\Omega_1$ which also happens to be one of its right inverse as the rows of $V_1^*\Omega_1$ are linearly independant), and therefore

$$\begin{aligned} \left|\left|\tilde{A} - \tilde{Q}\tilde{Q}^*\tilde{A}\right|\right|_F^2 &= \left|\left|\tilde{A} - P_{\tilde{Y}}\tilde{A}\right|\right|_F^2 \\ &\leq \left|\left|\tilde{A} - P_Z\tilde{A}\right|\right|_F^2 \\ &= \mathrm{Tr}(((I - P_Z)\tilde{A})^*(I - P_Z)\tilde{A}) \\ &= \mathrm{Tr}(\tilde{A}^*(I - P_Z)\tilde{A}) \\ &= \mathrm{Tr}(V\Sigma^*(I - P_Z)\Sigma V^*) \\ &= \mathrm{Tr}(\Sigma^*(I - P_Z)\Sigma) \end{aligned}$$

(the last equalities come from the fact that the trace is cyclic). But since Z has linearly independant columns, P_Z itself can be written as

$$P_Z = Z(Z^*Z)^{-1}Z^*$$

(since $(Z(Z^*Z)^{-1}Z^*)^2 = Z(Z^*Z)^{-1}Z^*$ and since $Z(Z^*Z)^{-1}Z^*$ is orthogonal and has range the range of Z, it is indeed the orthogonal projector on the range of Z), and thus

$$P_Z = \begin{pmatrix} I \\ F \end{pmatrix} (I + F^*F)^{-1} \begin{pmatrix} I \\ F \end{pmatrix}^* = \begin{pmatrix} (I+F^*F)^{-1} & (I+F^*F)^{-1}F^* \\ F(I+F^*F)^{-1} & F(I+F^*F)^{-1}F^* \end{pmatrix}$$

(where we write $F = \Sigma_2 V_2^* \Omega_2 (V_1^* \Omega_1)^\dagger \Sigma_1^{-1}$). Therefore, $I - P_Z$ can also be rewritten :

$$\begin{aligned} \Sigma^*(I - P_Z)\Sigma &= \Sigma^* \begin{pmatrix} I - (I+F^*F)^{-1} & \cdots \\ \cdots & I - F(I+F^*F)^{-1}F^* \end{pmatrix} \Sigma \\ &= \begin{pmatrix} \Sigma_1^*(I - (I+F^*F)^{-1})\Sigma_1 & \cdots \\ \cdots & \Sigma_2^*(I - F(I+F^*F)^{-1}F^*)\Sigma_2 \end{pmatrix} \end{aligned}$$

And thus,

$$\mathrm{Tr}(\Sigma^*(I - P_Z)\Sigma) = \mathrm{Tr}(\Sigma_1^*(I - (I+F^*F)^{-1})\Sigma_1) + \mathrm{Tr}(\Sigma_2^*(I - F(I+F^*F)^{-1}F^*)\Sigma_2)$$

But all the matrices involved in those traces are positive semi-definite and it is thus possible to show using classic arguments on the partial ordering of positive semi-definite matrices that

$$\begin{aligned} \mathrm{Tr}(\Sigma_1^*(I - (I+F^*F)^{-1})\Sigma_1) &\leq \mathrm{Tr}(\Sigma_1^* F^* F \Sigma_1) \\ \mathrm{Tr}(\Sigma_2^*(I - F(I+F^*F)^{-1}F^*)\Sigma_2) &\leq \mathrm{Tr}(\Sigma_2^*\Sigma_2) \end{aligned}$$

We thus finally get

$$\begin{aligned} ||A - QQ^*A||_F^2 &\leq \mathrm{Tr}(\Sigma_1^* F^* F \Sigma_1) + \mathrm{Tr}(\Sigma_2^*\Sigma_2) \\ &= ||F\Sigma_1||_F^2 + ||\Sigma_2||_F^2 \\ &= \left|\left|\Sigma_2 V_2^* \Omega_2 (V_1^*\Omega_1)^\dagger\right|\right|_F^2 + ||\Sigma_2||_F^2 \end{aligned}$$

This proof is very technical but the overall intuition to get is that if the the smallest $n-k$ singular values (contained in Σ_2) are small enough, sampled vectors will, on average, have most of their norm be induced by their components along the vectors corresponding to the biggest k singular values, which also happen to generate the best approximation of the range of A by a k-dimensional subspace (as given by a PCA).

The question that is left to answer is how to actually sample the vectors before passing them through A : this is the topic of section 2.3. The sampled vectors should verify the assumption that Ω_1 has linearly independant column and rows (this is true with probability 1 as soon as the distribution is absolutely continuous with respect to the Lebesgue measure), and that on average $\left|\left|V_2^*\Omega_2(V_1^*\Omega_1)^\dagger\right|\right|_F$ is not too big (this really only depends on the distribution and not on V_1 and V_2 as V is orthogonal). Also note that if the distribution is indeed absolutely continuous with respect to the Lebesgue measure, then if k is greater than the rank of A (that is $|\sigma_k| = 0$), we immediately get that $A = QQ^*A$ almost surely because Q has rank the rank of A.

2.2.2. Randomized subspace iteration

Following the previous intuition, it may be so that the first k singular values are not big enough compared to the last $n-k$ ones, possibly resulting in the image by A of the sampled vectors not being concentrated enough on the vectors corresponding to the first k singular values (when $||V_2^*\Omega_2||_F$ is big enough with respect to $\left|\left|(V_1^*\Omega_1)^\dagger\right|\right|_F^{-1}$), thus degrading the approximation of the range of A. To dampen the impact of this problem, one may instead use a *randomized subspace iteration*, that tries to approximate the range of the matrix $B = (AA^*)^qA$, where q is a non-negative integer. Since B has the same range as A (its range is included in that of A and its rank is that of k), this approximation will also happen to be an approximation of the range of A.

In practice, to decrease the impact of approximation errors resulting from finite-precision arithmetic, we renormalize the range between each application of A and A^*, and proceed thus as follows.

- sample $k \leq n$ random vectors of size n and form the matrix Ω ;
- derive a generating family of an approximation of the range of A by taking $Y_0 = A\Omega$;
- make this family orthogonal by choosing Q_0 as in the QR factorization $Y_0 = Q_0R_0$;
- then, for each $1 \leq i \leq q$,
  - derive a generating family of an approximation of the range of $A^*(AA^*)^{i-1}A$ by taking $\tilde{Y}_i = A^*Y_i$;
  - make this family orthogonal by choosing $\tilde{Q}_i$ as in the QR factorization $\tilde{Y}_i = \tilde{Q}_i\tilde{R}_i$;
  - derive a generating family of an approximation of the range of $(AA^*)^iA$ by taking $Y_i = A\tilde{Y}_i$;
  - make this family orthogonal by choosing Q_i as in the QR factorization $Y_i = Q_iR_i$.

Note that $(AA^*)A\Omega = Q_q(\Pi_iR_i\tilde{R}_i)R_0$ and so the columns of $Q = Q_q$ indeed make an orthogonal family that generates the approximation $(AA^*)A\Omega$ of the range of $(AA^*)A$. In particular, QQ^* is the orthogonal projector on the subspace generated by the columns of $(AA^*)A\Omega$ and therefore the inequality derived in section 2.2.1 yields

$$\begin{aligned}
||(I-QQ^*)(AA^*)^qA||_F^2 &\leq \left|\left|(\Sigma_2\Sigma_2^*)^q\Sigma_2V_2^*\Omega_2(V_1^*\Omega_1)^\dagger\right|\right|_F^2 + ||(\Sigma_2\Sigma_2^*)^q\Sigma_2||_F^2 \\
&\leq \left(1 + \left|\left|V_2^*\Omega_2(V_1^*\Omega_1)^\dagger\right|\right|_F^2\right) ||(\Sigma_2\Sigma_2^*)^q\Sigma_2||_F^2 \\
&\leq \left(1 + \left|\left|V_2^*\Omega_2(V_1^*\Omega_1)^\dagger\right|\right|_F^2\right) ||\Sigma_2||_F^{4q+2}
\end{aligned}$$

This does not yet give a bound on $||A - QQ^*A||_F$, though. Fortunately, writing $P = I - QQ^*$,

$$\begin{aligned} ||PA||_F^{2(2q+1)} &= \mathrm{Tr}(A^*P^*PA)^{(2q+1)} \\ &= \mathrm{Tr}(A^*PA)^{2q+1} \\ &= \mathrm{Tr}(V\Sigma^*U^*PU\Sigma V^*)^{2q+1} \\ &= \mathrm{Tr}(U^*PU\Sigma V^*V\Sigma^*)^{2q+1} \\ &= \mathrm{Tr}(U^*PU\Sigma\Sigma^*)^{2q+1} \end{aligned}$$

But using Jensen's inequality, one may show that

$$\mathrm{Tr}(U^*PU\Sigma\Sigma^*)^{2q+1} \leq \left(\frac{n-k}{n}\right)^{2q} \mathrm{Tr}(U^*PU(\Sigma\Sigma^*)^{2q+1})$$

And therefore using similar equalities in reverse order

$$\begin{aligned} ||PA||_F^{2(2q+1)} &\leq \mathrm{Tr}(U^*PU(\Sigma\Sigma^*)^{2q+1}) \\ &= ||P(AA^*)^qA||_F^2 \end{aligned}$$

Which finally gives

$$||A - QQ^*A||_F^2 \leq \left(1 + \left|\left|V_2^*\Omega_2(V_1^*\Omega_1)^\dagger\right|\right|_F^2\right)^{1/(2q+1)} ||\Sigma_2||_F^2$$

Compared to the randomized range finder of section 2.2.1, we additionally take the $(2q+1)$th-root of $1 + \left|\left|V_2^*\Omega_2(V_1^*\Omega_1)^\dagger\right|\right|_F^2$, which in practice implies the randomized subspace iteration is, as expected, less sensitive to the concentration of the sampled vectors on the subspace corresponding to the last $n-k$ singular values.

2.2.3. Adaptive range finders

In practice, one may not necessarily look for a fixed-sized matrix Q but instead for one that achieves a precision $||A - QQ^*A||_F \leq \epsilon$. The adaptive versions of the algorithms described in sections 2.2.1 and 2.2.2 are designed for this task. Here we only introduce the adaptive version of the randomized range finder because the randomized subspace iteration works more or less the same way but is slightly more complicated.

The idea is to build an approximation of the range of A by adding vectors one-by-one until the desired precision is reached. To do so, we use both the consecutive and the dynamic method for computing a QR factorization, as seen in section 2.1.1. To ensure the desired precision was reached, we test that the orthogonal projection of random vectors on the orthogonal of our current approximation are small enough ; if not, we add one of these vectors to the our approximation and start again. We thus proceed as follows :

- sample r random vectors $Y = [y_1, \ldots, y_r]$ of size m in the range of A ;

- set Q to be the empty matrix of size $m \times 0$ (the empty family of vectors of size m) ;
- as long as one of $y_1, \ldots, y_r$ has norm greater than ϵ, do the following :
  - take q to be the renormalized orthogonal projection of y_1 on $\langle Q \rangle^\perp$;
  - sample a new vector y of size n in the range of A and project it onto $\langle Q \rangle^\perp$ (using the consecutive method since it is a fresh vector) ;
  - add q to Q (we still get an orthogonal family since we just used the consecutive method for computing the QR factorization) ;
  - set Y to be $[y_2, \ldots, y_r, y]$;
  - project each column of Y onto $\langle Q \rangle^\perp$ (using the dynamic method as, prior to this step, they were already projected against the previous version of $\langle Q \rangle^\perp$, and we thus only need to remove the projection along q).

We thus end up with a matrix Q verifying that, for r random vectors y_i in the range of A, $||(I - QQ^*)y_i|| \leq \epsilon/(10\sqrt{2/\pi})$. If p is the probability that, for a random vector y in the range of A, $||A - QQ^*A||_F \geq ||y - QQ^*y||$, then with probability at least $1 - p^r$,

$$||A - QQ^*A||_F \leq \epsilon$$

Thus our sampling distribution should also verify that the probability that $||A - QQ^*A||_F \geq ||y - QQ^*y||$ is as low as possible. Note that, to decrease the probability of failure of the algorithm, it is also possible to use the probability that $||A - QQ^*A||_F \geq C\,||y - QQ^*y||$, where C is a well-chosen constant by which ϵ should be divided.

2.3. Range sampling

In section 2.2, we sampled vectors in the range of a matrix A to derive probabilistic matrix approximation algorithms. The sampling distribution should be absolutely continuous with respect to the Lebesgue measure, and should be the push-forward by A of a distribution on the domain of A (meaning we sample a vector by computing the image by A of a random vector sampled in the domain of A).

The first distribution on the domain of A one may push forward is the classic gaussian distribution : to sample a vector in the range of A, first sample a vector ω from $\mathcal{N}(\mathbf{0}, \mathbf{1})$, and output $A\omega$. But push-forward of distributions have the disadvantage of being costly to sample, as they require a matrix product. For example, sampling k vectors from the push-forward of the gaussian distribution requires $O(mnk)$ time.

Instead, one may used more advanced distributions to allow for the optimization of the matrix product. One such distribution uses the Fourier transform (for complex matrices) or the Hadamard transform (for real matrices) to make this computation faster. The idea is to multiply A by an $n \times n$ diagonal matrix with coefficients sampled on the unit circle (this costs $O(n)$ time), compute a fast Fourier or fast Hadamard transform of its rows ($O(mn \log n)$ time), and output k random columns extracted from the resulting matrix. The complexity of the Fourier / Hadamard transform step may even be reduced to $O(mn \log k)$ using a subsampled fast Fourier transform.

For both of these sampling methods, analysis of the probabilistic error bounds provided in section 2.2 are given in the original article [1]. They are not detailed here as they do not help get a better intuition of how the algorithms work.

3. Practice

The algorithms described in section 2 were implemented into an OCaml library [2]. OCaml was chosen as the target language because it makes it easy to create modular programs (it is easy to combine different algorithms for different phases of the computation together), which is one of the features of these probabilistic methods that were put forward by the authors of the original article [1]. Although it also makes it easier to write correct code, it has the disadvantage of making it difficult to work with different types of matrices : for instance, working with Fourier transforms on real matrices can be quite cumbersome.

The documentation for the libary can be found in the appendix : the `Fact` module (appendix A) implements the probabilistic algorithms for low-rank matrix approximation, the `Matrix` module (appendix B) provides the interface for linking with any linear algebra library, and the `Test` module (appendix C) provides functions that may help for checking the output values. Note that this documentation may also be generated directly from the source code.

The following is a Jupyter notebook whose aim is to introduce the library to newcomers. As an example, it realizes statistical tests to emprically confirm the theoretical bound on the probability of failure of the adaptive algorithms, and compares the empirical means of the different algorithms that are provided.

Introduction

PMA is an OCaml library for Probabilistic Matrix Approximation. It implements the algorithms from the paper Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions. Some algorithms still need to be implemented or optimized, and the code still needs to be fully tested.

The documentation can be generated with `dune build @doc-private` and can then be found in `_build/default/_doc/_html`.

```
#use "topfind"
#require "owl,owl-top,owl-plplot,jupyter.notebook"

#use_output "dune top" ;;

Jupyter_notebook.clear_output ()
```

The main feature of PMA is modularity : although it is built with the goal to be used along with the Owl library, it can in practice be used with any linear algebra library by providing an interface module of signature `Pma.Matrix.Matrix` to the functor `Pma.Fact.Make`.

```
#show Pma.Matrix.Matrix
#show Pma.Fact.Make
```

```
module type Matrix =
  sig
    type elt
    type mat
    type complex_mat
    val to_float : elt -> float
    val cast : mat -> complex_mat
    val shape : mat -> int * int
    val zeros : int -> int -> mat
    val eye : int -> mat
    val gaussian : ?mu:elt -> ?sigma:elt -> int -> int -> mat
    val sub : mat -> mat -> mat
    val dot : mat -> mat -> mat
    val dot_complex : complex_mat -> complex_mat -> complex_mat
    val mul : mat -> mat -> mat
    val div_scalar : mat -> elt -> mat
    val triangular_solve : mat -> mat -> mat
    val max' : mat -> elt
    val l2norm : ?axis:int -> ?keep_dims:bool -> mat -> mat
    val l2norm' : mat -> elt
    val get_slice : int list list -> mat -> mat
    val transpose : mat -> mat
    val ctranspose : mat -> mat
    val concat_horizontal : mat -> mat -> mat
    val qr : mat -> mat * mat
    val svd : mat -> mat * mat * mat
    val eig : mat -> complex_mat * complex_mat
    val chol : mat -> mat
```

```
  end
module Make = Pma.Fact.Make
module Make : Pma.Matrix.Matrix -> Pma.Fact.Fact
```

Such an implementation is already provided for the submodules of `Owl.Dense.Matrix`.

```
module RNdarray = Owl.Dense.Ndarray.D
module RMat = Owl.Dense.Matrix.D
module RLinalg = Owl.Linalg.D ;;
module RFact = Pma.Fact.D
module RFactTest = Pma.Test.D

module C = Complex
module CMat = Owl.Dense.Matrix.Z
module CFact = Pma.Fact.Z
module Fft = Owl.Fft.D

module Ndarray = Owl.Dense.Ndarray.Generic ;;
module Maths = Owl.Maths
module Plot = Owl_plplot.Plot
```

Content of the library

Let us now review the content of a module of signature `Pma.Fact.Fact`. Throughout this review, we will work with 64-bit complex matrices implemented in the `Owl.Dense.Matrix.Z` module.

```
#show Pma.Fact.Fact
```

```
module type Fact =
  sig
    type elt
    type mat
    type complex_mat
    type range_sampler = mat -> int -> mat
    val gaussian_range_sampler : range_sampler
    type range_finder = mat -> mat
    val randomized_range_finder :
      ?sampler:range_sampler -> int -> range_finder
    val adaptive_randomized_range_finder :
      ?sampler:range_sampler -> ?r:int -> float -> range_finder
    val randomized_subspace_iteration :
      ?sampler:range_sampler -> int -> int -> range_finder
    val adaptive_randomized_subspace_iteration :
      ?sampler:range_sampler -> ?r:int -> int -> float -> range_finder
    val direct_svd : mat -> mat -> mat * mat * mat
    val direct_eig : mat -> mat -> complex_mat * complex_mat
    val nystrom_eig : mat -> mat -> mat * mat
    val approximate : range_finder -> (mat -> mat -> 'a) -> mat -> 'a
  end
```

Three types of functions are defined : *range samplers*, *range finders* and *postprocessors*.

Range samplers

Range samplers are functions that sample random vectors in the range of a matrix. The default range sampler uses gaussian sampling.

```
#show CFact.range_sampler ;;
#show CFact.gaussian_range_sampler ;;
```

```
type nonrec range_sampler = CFact.mat -> int -> CFact.mat
val gaussian_range_sampler : CFact.range_sampler
```

```
let a = CMat.ones 3 3 in
let range_sampler = CFact.gaussian_range_sampler a in
range_sampler 4
```

```
- : CFact.mat =

             C0              C1             C2             C3
R0 (1.1803, 0i) (-0.519549, 0i) (1.76975, 0i) (2.45171, 0i)
R1 (1.1803, 0i) (-0.519549, 0i) (1.76975, 0i) (2.45171, 0i)
R2 (1.1803, 0i) (-0.519549, 0i) (1.76975, 0i) (2.45171, 0i)
```

Range samplers are easy to add. As an example, we implement the Fourier range sampler (Algorithm 4.5), not implemented by default in PMA because the Hadamard transform (necessary for real matrices) is not implemented in Owl.

The following version only supports sampling at most n vectors from an $n \times m$ matrix, but this is not a problem for matrices of rank $r \ll n$ as they usually only require the sampling of around r vectors.

When implementing a range sampler, it is important to keep in mind that, during the execution of a range finder, the sampler is instantiated once for a given matrix and the resulting function may then be called several times.

```
let fourier_range_sampler (a : CMat.mat) : int -> CMat.mat =
    let _, n = CMat.shape a in

    (* d ~ U(unit circle) *)
    let d : CMat.mat = CMat.polar (RMat.ones 1 n) (RMat.uniform ~b:(2. *.␣
 ↪Float.pi) 1 n) in

    let samples = CMat.shuffle_rows (CMat.mul a d) in
    let i = ref 0 in
    let sampler k =
        let samples_chosen = try
            CMat.get_slice [ [] ; [!i ; !i+k-1] ] samples
            with _ -> failwith "too many samples required" in
        i := !i + k ;
```

```
      CMat.mul_scalar (Fft.fft ~axis:1 samples_chosen) C.{re = Stdlib.sqrt␣
↪(float n /. float k) ; im = 0.} in
   sampler
```

```
val fourier_range_sampler : CMat.mat -> int -> CMat.mat = <fun>
```

```
let a = CMat.ones 3 3 in
let range_sampler = fourier_range_sampler a in
range_sampler 3
```

```
- : CMat.mat =

                        C0                     C1                     C2
R0 (0.695991, -0.0327037i) (0.657222, 0.0672092i) (-0.408123, 2.81274i)
R1 (0.695991, -0.0327037i) (0.657222, 0.0672092i) (-0.408123, 2.81274i)
R2 (0.695991, -0.0327037i) (0.657222, 0.0672092i) (-0.408123, 2.81274i)
```

Range finders

Range finders are functions that find an orthogonal family that generates the range of a matrix.

They do so by first finding a generating family of the range of the matrix using a range sampler (that can be specified as an optional parameter) and then making that family orthogonal using orthogonalization procedures such as the Gram-Schmidt process.

Let us for example find a basis for the range of a simple matrix whose coefficients are all ones. We know that its rank is 1, and thus we can ask our range finder to find a family of size 1.

Note that for matrices with bigger ranks r, because of the probabilistic nature of the algorithms, you should ask the range finder to find a family of size $l = k + p$ with $p \sim 5$ or $p \sim 10$ and k the expected size of the family of vectors. Of course, when $k < r$ one may only expect to get an approximation of the range of the matrix.

```
#show CFact.randomized_range_finder ;;
```

```
val randomized_range_finder :
  ?sampler:CFact.range_sampler -> int -> CFact.range_finder
```

```
let a = CMat.ones 3 3 in
let range_finder = CFact.randomized_range_finder 1 in
range_finder a
```

```
- : CFact.mat =

               C0
R0 (-0.57735, 0i)
R1 (-0.57735, 0i)
R2 (-0.57735, 0i)
```

What if the rank of the matrix is unknown ? Each range finder has an *adaptive* version : instead of looking for a generating family of fixed size, it grows the family until sufficient precision ϵ has been reached. If A is the $m \times n$ matrix whose range we are looking for, the resulting orthogonal family Q verifies

$$||(I - QQ^*)A|| \leq \epsilon$$

with probability at least $1 - \min(m, n)10^{-r}$, where r is an optional parameter that defaults to 10.

```
#show CFact.adaptive_randomized_range_finder ;;
```

```
val adaptive_randomized_range_finder :
  ?sampler:CFact.range_sampler -> ?r:int -> float -> CFact.range_finder
```

```
let a = CMat.ones 3 3 in
let epsilon = 0.01 in
let range_finder = CFact.adaptive_randomized_range_finder epsilon in
let q = range_finder a in
assert ((CMat.sub a (CMat.dot q @@ CMat.dot (CMat.ctranspose q) @@ a) |>␣
 ↪CMat.l2norm').re <= epsilon) ;
q
```

```
- : CFact.mat =

              C0
R0 (0.57735, 0i)
R1 (0.57735, 0i)
R2 (0.57735, 0i)
```

`randomized_range_finder` is the most basic kind of range finder. For matrices whose singular values decay slowly, its performance can be improved by finding the range of powers of the matrix (effectively making the singular values decay faster). This procedure is called *subspace iteration*. In practice, it only requires the additional specification of the power and it can then be used just as the basic range finder.

```
#show CFact.randomized_subspace_iteration
#show CFact.adaptive_randomized_subspace_iteration
```

```
val randomized_subspace_iteration :
  ?sampler:CFact.range_sampler -> int -> int -> CFact.range_finder
val adaptive_randomized_subspace_iteration :
  ?sampler:CFact.range_sampler ->
  ?r:int -> int -> float -> CFact.range_finder
```

Postprocessors

Postprocessors use the range of a matrix to factorize it. In practice it is only provided with an approximation of the range so the resulting factorization is approximative.

Currently three kinds of factorization are provided : singular value decomposition, eigenvalue decomposition and eigenvalue decomposition using the Nyström mathod (which is more accurate but only works with positive semidefinite matrices).

```
#show CFact.direct_svd
#show CFact.direct_eig
#show CFact.nystrom_eig
```

```
val direct_svd : CFact.mat -> CFact.mat -> CFact.mat * CFact.mat * CFact.mat
val direct_eig :
  CFact.mat -> CFact.mat -> CFact.complex_mat * CFact.complex_mat
val nystrom_eig : CFact.mat -> CFact.mat -> CFact.mat * CFact.mat
```

Here, since our target matrix is positive semidefinite, we use the Nyström method.

```
let a = CMat.ones 3 3 in
let q = CFact.randomized_range_finder 1 a in
let u, lambda = CFact.nystrom_eig a q in
u, lambda, CMat.dot u @@ CMat.mul lambda @@ CMat.ctranspose u
```

```
- : CFact.mat * CFact.mat * CMat.mat =
(
               C0
R0 (-0.57735, 0i)
R1 (-0.57735, 0i)
R2 (-0.57735, 0i)
,

        C0
R0 (3, 0i)
,

        C0      C1      C2
R0 (1, 0i) (1, 0i) (1, 0i)
R1 (1, 0i) (1, 0i) (1, 0i)
R2 (1, 0i) (1, 0i) (1, 0i)
)
```

In practice, there is no need to use the postprocessors directly : they may be passed as an argument to the `approximate` function.

```
#show CFact.approximate
```

```
val approximate :
  CFact.range_finder -> (CFact.mat -> CFact.mat -> 'a) -> CFact.mat -> 'a
```

```
let a = CMat.ones 3 3 in
CFact.approximate (CFact.randomized_range_finder 1) CFact.nystrom_eig a
```

```
- : CFact.mat * CFact.mat =
(
                C0
R0 (-0.57735, 0i)
R1 (-0.57735, 0i)
R2 (-0.57735, 0i)
,

         C0
R0 (3, 0i)
)
```

Checking correctness

Let us now check that the claims that the output of the algorithms verify some conditions with high probability are true. To do so we carry out a binomial test.

We will be using functions imported from the `Pma.Test` submodules.

```
#show Pma.Test.Test
```

```
module type Test =
  sig
    type elt
    type mat
    type range_finder
    val error_factorization : mat list -> mat -> elt
    val error_range_finder : range_finder -> mat -> elt
    val test_error : elt -> float -> bool
    val test_adaptive_range_finder :
      (float -> range_finder) -> float -> mat -> bool
  end
```

Adaptive randomized range finder

We first study the case of the adaptive algorithm : we will check that

$$||(I - QQ^*)A|| \leq \epsilon$$

with probability at least $1 - \min(m, n)10^{-r}$. We will do so for both fixed and random A.

We now use real 64-bit matrices.

```
type int_ndarray = (int, Bigarray.int16_unsigned_elt) Ndarray.t ;;
let int_ndarray_kind = Bigarray.Int16_unsigned ;;
```

```
let int_range
    ?start:(start=0)
```

```
    ?step:(step=1)
    (n : int)
    : int_ndarray =
    Ndarray.init int_ndarray_kind [|n-start|] (fun i -> start + step * i)
```

```
val int_range : ?start:int -> ?step:int -> int -> int_ndarray = <fun>
```

```
let count_adaptive_range_finder_failures
    ?show_steps:(show_steps=false)
    (rs : int_ndarray)
    (epsilons : RNdarray.arr)
    (iters : int)
    (gen_mat : int -> float -> int -> RMat.mat)
    : int_ndarray =
    let shape = Array.concat [Ndarray.shape rs ; Ndarray.shape epsilons] in
    let failures_range_finder = Ndarray.create int_ndarray_kind shape 0 in
    Ndarray.iteri_nd (fun r_index r ->
        RNdarray.iteri_nd (fun eps_index eps ->
            for k = 1 to iters do
                if show_steps then (
                    Jupyter_notebook.printf "r = %d, eps = %f, iter = %d /␣
 ↪%d@." r eps k iters ;
                    ignore (Jupyter_notebook.display_formatter "text/html") ;
                ) ;
                let a = gen_mat r eps k in
                if not (RFactTest.test_adaptive_range_finder (RFact.
 ↪adaptive_randomized_range_finder ~r:r) eps a) then
                    let index = Array.concat [r_index ; eps_index] in
                    Ndarray.set failures_range_finder index (Ndarray.get␣
 ↪failures_range_finder index + 1)
            done)
        epsilons)
    rs ;
    failures_range_finder
```

```
val count_adaptive_range_finder_failures :
  ?show_steps:bool ->
  int_ndarray ->
  RNdarray.arr -> int -> (int -> float -> int -> RMat.mat) -> int_ndarray =
  <fun>
```

Approximating a Laplacian

We first start by approximating a Laplacian in dimension $n = 100$. Laplacians are matrices that correspond to discrete second-order derivatives ; as such, they are widely used as soon as differential equations come into play.

```
let n = 100
```

```
let laplacian =
    let laplacian_upper = RMat.bidiagonal
                            (RMat.scalar_mul 2. (RMat.ones 1 n))
                            (RMat.scalar_mul (-1.) (RMat.ones 1 (n-1))) in
    RMat.set laplacian_upper 0 (n-1) (-1.) ;
    RMat.symmetric laplacian_upper
```

```
val laplacian : RMat.mat =

     C0  C1  C2  C3  C4     C95 C96 C97 C98 C99
 R0   2  -1   0   0   0 …   0   0   0   0  -1
 R1  -1   2  -1   0   0 …   0   0   0   0   0
 R2   0  -1   2  -1   0 …   0   0   0   0   0
 R3   0   0  -1   2  -1 …   0   0   0   0   0
 R4   0   0   0  -1   2 …   0   0   0   0   0
    … … … … … … … … … … …
R95   0   0   0   0   0 …   2  -1   0   0   0
R96   0   0   0   0   0 …  -1   2  -1   0   0
R97   0   0   0   0   0 …   0  -1   2  -1   0
R98   0   0   0   0   0 …   0   0  -1   2  -1
R99  -1   0   0   0   0 …   0   0   0  -1   2
```

We chose to compute 100 approximations for each set of parameters of the adaptive randomized range finders. Here we constrain r to range from 2 to 5 and ϵ from 0.0001 to 1.

It doesn't make sense to go with greater values of r since the probability of failure should be lower than $n10^{-r}$ which is below $1/1000$ for $r \geq 5$, thus making it unlikely to have any failure on a sample size of 100.

```
let epsilons = (RNdarray.logspace ~base:10. 0. (-4.) 5) ;;
let rs = (int_range ~start:2 6) ;;

let failures = count_adaptive_range_finder_failures
    ~show_steps:true
    rs
    epsilons
    100
    (fun _ _ _ -> laplacian)
in Jupyter_notebook.clear_output () ;
failures
```

```
- : int_ndarray =

   C0 C1 C2 C3 C4
R0  0  0  0  0  0
R1  0  0  0  0  0
R2  0  0  0  0  0
R3  0  0  0  0  0
```

We find that the procedure never failed, so the statistical test to carry out is very simple : the p-value for the hypothesis that, for a fixed r, the probability of failure is $\pi \in [0,1]$ is $2*(1-\pi)^{500}$ (we carried out 100 experiments for each value of ϵ). In particular, we can reject (for the chosen parameters) the hypothesis that the probability of failure is greater than $1 - \min(n,m)10^{-r}$ with confidence $1 - 2 * (1-\pi)^{500}$.

```
let new_get_color_fun () : unit -> Plot.spec =
    let cycle = Array.map (fun (r,g,b) -> Plot.RGB (r,g,b))
        [|(31,119,180) ; (255,127,14) ; (44,160,44) ; (214,39,40) ;␣
 ↪(148,103,189) ; (140,86,75) ; (227,119,194) ; (127,127,127) ;␣
 ↪(188,189,34) ; (23,190,207)|] in
    let i = ref 0 in
    let n = Array.length cycle in
    (fun () -> let color = cycle.(!i mod n) in i := !i + 1 ; color)
```

```
val new_get_color_fun : unit -> unit -> Plot.spec = <fun>
```

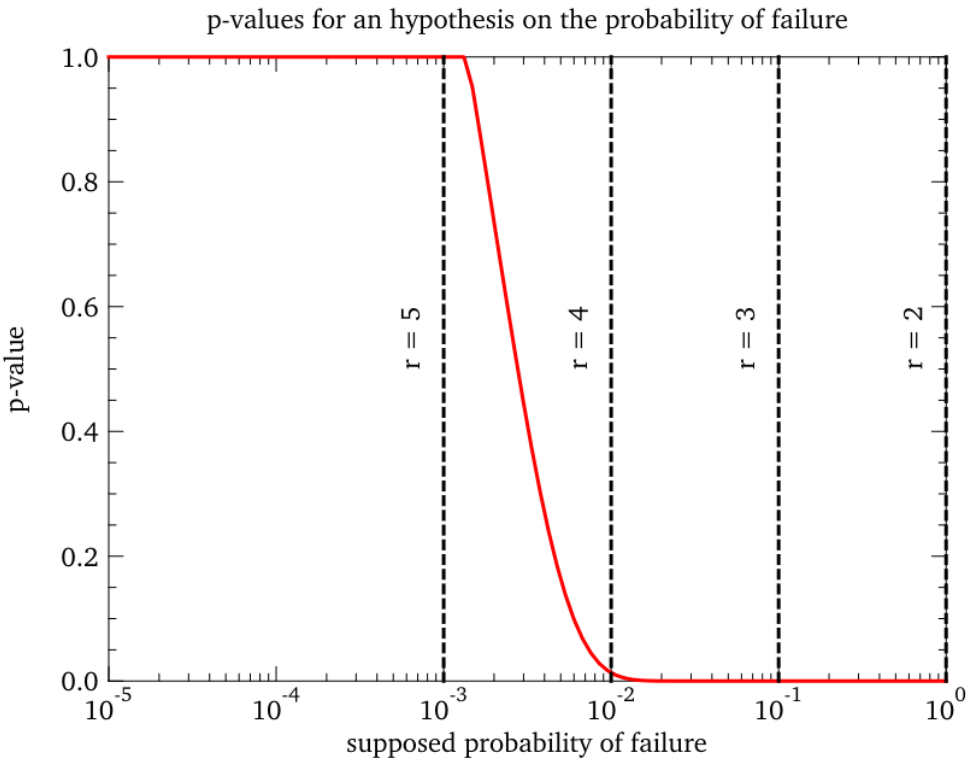
```
let display_plot
    ?path:(path="/tmp/owl.png")
    (h : Plot.handle)
    : Jupyter_notebook.display_id =

    Plot.output h ;

    let ic = open_in_bin path in
    let n = in_channel_length ic in
    let data = really_input_string ic n in
    close_in ic ;
    Jupyter_notebook.display ~base64:true "image/png" data
```

```
val display_plot : ?path:string -> Plot.handle -> Jupyter_notebook.
 ↪display_id =
  <fun>
```

```
let h = Plot.create "/tmp/owl.png" ;;

Plot.set_yrange h 0. 1. ;
let xs = RNdarray.logspace ~base:10. (-5.) 0. 100 in
let pvalues = RNdarray.map (fun x -> min 1. (2. *. (1. -. x) ** 500.)) xs in
Plot.semilogx ~h ~spec:[RGB (255,0,0) ; LineWidth 3.] ~x:xs pvalues ;

Ndarray.iter (fun r ->
    let log_probability = 2. -. (float r) in
    Plot.draw_line ~h ~spec:[LineStyle 3 ; LineWidth 3.] log_probability 0.␣
 ↪log_probability 1. ;
    let legend = Format.sprintf "r = %d" r in
```

```
    Plot.text ~h (log_probability -. 0.2) 0.5 ~dy:1. legend ;
) rs ;

Plot.set_xlabel h "supposed probability of failure" ;
Plot.set_ylabel h "p-value" ;
Plot.set_title h "p-values for an hypothesis on the probability of failure" ;
```

```
display_plot h
```




```
- : Jupyter_notebook.display_id = <abstr>
```

As expected, we confirmed with high confidence that the probability of failure is indeed below the theoretical bound for $r = 2, 3, 4$. For $r = 5$ more experiments need to be carried out to confirm the hypothesis (this is expected as the theoretical bound is 1/1000).

Approximating random matrices

We can do the same test but with random matrices. To ensure they don't have full rank, we take them of size $m \times n$ where $m = 100$ and n is chosen randomly between 10 and 90.

```
let random_mat _ _ _ =
    let m = 100 in
    let n = 10 + Random.int 80 in
    RMat.gaussian m n
```

```
val random_mat : 'a -> 'b -> 'c -> RMat.mat = <fun>
```

```
let failures = count_adaptive_range_finder_failures
    ~show_steps:true
    (int_range ~start:2 6)
    (RNdarray.logspace ~base:10. 0. (-4.) 5)
    100
    random_mat
in Jupyter_notebook.clear_output () ;
failures
```

```
- : int_ndarray =

   C0 C1 C2 C3 C4
R0  0  0  0  0  0
R1  0  0  0  0  0
R2  0  0  0  0  0
R3  0  0  0  0  0
```

Once again, we find that the procedure never failed, so we've confirmed empirically that

$$||(I - QQ^*)A|| \leq \epsilon$$

with probability at least $1 - \min(m, n)10^{-r}$ for random matrices and $r = 2, 3, 4$, and that more experiments need to be carried out for $r = 5$.

Computing the average error

We now set out to compute an empirical average for the error $||(I - QQ^*)A||$. Once again, we will do so with both fixed and random A.

```
let errors_range_finder
    ?show_steps:(show_steps=false)
    (range_finder : int -> RFact.range_finder)
    (ls : int_ndarray)
    (sample_size : int)
    (gen_mat : int -> int -> RMat.mat)
    : RMat.mat =
    let ls = Ndarray.flatten ls in
    RMat.init_2d (Ndarray.shape ls).(0) sample_size (fun i iter ->
        let l = Ndarray.get ls [|i|] in
        if show_steps then (
            Jupyter_notebook.printf "iter = %d / %d, l = %d" (iter + 1)␣
 ↪sample_size l ;
            ignore (Jupyter_notebook.display_formatter "text/html") ;
        ) ;
        let a = gen_mat l iter in
        RFactTest.error_range_finder (range_finder l) a
```

```
    )
```

```
val errors_range_finder :
  ?show_steps:bool ->
  (int -> RFact.range_finder) ->
  int_ndarray -> int -> (int -> int -> RMat.mat) -> RMat.mat = <fun>
```

We will compare these empirical errors to the optimal error we can get when approximating a matrix by another fixed-rank one, which can be achieved by carrying out a principal component analysis (PCA).

```
let errors_pca (a : RMat.mat) : RMat.mat =
    let u, _, v = RLinalg.svd a in
    let _, n = RMat.shape a in
    RMat.init 1 n (fun i ->
        let new_u = RMat.get_slice [[] ; [0;i]] u in
        let new_v = RMat.get_slice [[0;i] ; []] v in
        let q = RMat.dot new_u new_v in
        RFactTest.error_range_finder (fun _ -> q) a
    )
```

```
val errors_pca : RMat.mat -> RMat.mat = <fun>
```

Finally we will also compare these average errors to the theoretical bound given in the paper.

```
let errors_theory (a : RMat.mat) : RMat.mat =
    let _, eigvals, _ = RLinalg.svd a in
    let _, n = RMat.shape a in
    RMat.init 1 n (fun i ->
        let l = i + 1 in
        if l <= 4 then
            nan
        else if l < n then
            let ps = int_range ~start:2 (l-2) in
            RMat.init 1 (l-4) (fun i ->
                let p = Ndarray.get ps [|i|] in
                let k = l - p in
                sqrt (1. +. float k /. (float p -. 1.)) *. RMat.l2norm'␣
 ↪(RMat.get_slice [[];[k+1;n-1]] eigvals)
            ) |> RMat.min'
        else
            0.
    )
```

```
val errors_theory : RMat.mat -> RMat.mat = <fun>
```

```
let compute_average_errors
    ?show_steps:(show_steps=false)
    (a : RMat.mat)
    (sample_size : int)
    (subspace_iterations : int_ndarray)
    : RMat.mat * RMat.mat * RMat.mat =

    let m, n = RMat.shape a in
    let dim = max m n in

    let ls = int_range ~start:1 (dim+1) in

    let errors_range_finder_a = RNdarray.empty [|(Ndarray.shape␣
 ↪subspace_iterations).(0) ; dim ; sample_size|] in
    Ndarray.iteri (fun i power ->
        if show_steps then (
            Jupyter_notebook.printf "subspace iteration = %d" power ;
            ignore (Jupyter_notebook.display_formatter "text/html") ;
        ) ;
        RNdarray.set_slice
            [[i] ; [] ; []]
            errors_range_finder_a
            (RNdarray.reshape
                (errors_range_finder
                    ~show_steps:false
                    (RFact.randomized_subspace_iteration power)
                    ls
                    sample_size
                    (fun _ _ -> a))
                [|1 ; dim ; sample_size|])
    ) subspace_iterations ;

    if show_steps then
        Jupyter_notebook.clear_output () ;

    let errors_pca_a = errors_pca a in
    let errors_theory_a = errors_theory a in

    errors_range_finder_a, errors_pca_a, errors_theory_a
```

```
val compute_average_errors :
  ?show_steps:bool ->
  RMat.mat -> int -> int_ndarray -> RMat.mat * RMat.mat * RMat.mat = <fun>
```

```
let plot_errors
    ?path:(path="/tmp/owl.png")
    (subspace_iterations : int_ndarray)
    (errors_range_finder, errors_pca, errors_theory : RNdarray.arr * RMat.
 ↪mat * RMat.mat)
```

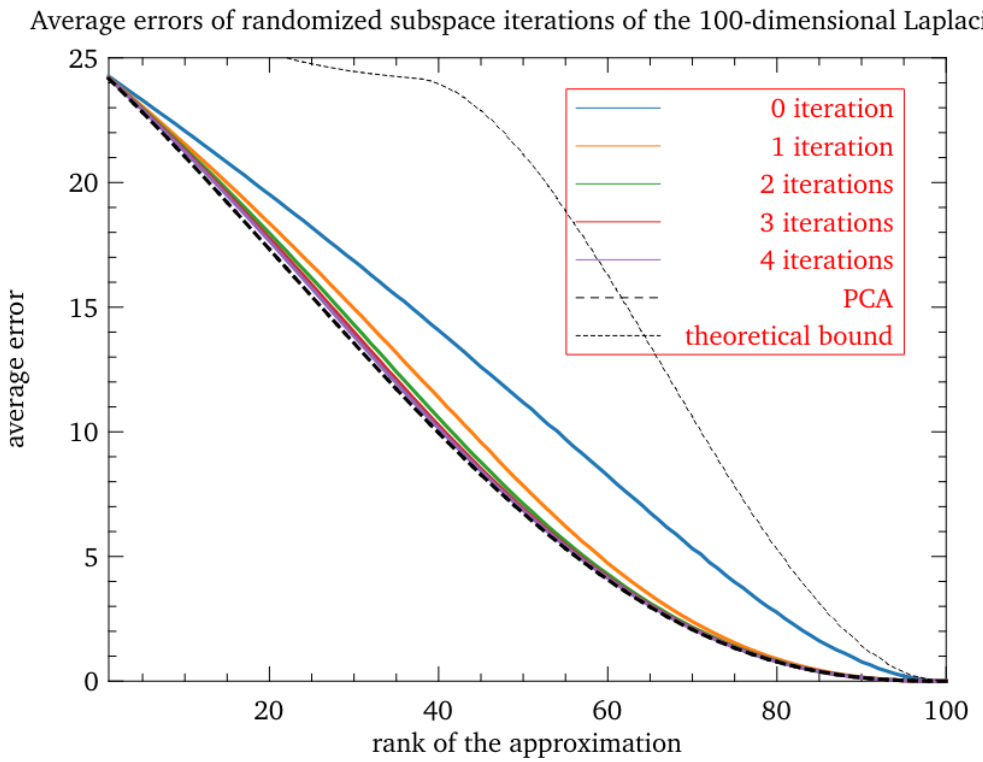
```
    : Plot.handle =

    let _, dim = RMat.shape errors_pca in

    let ls = RMat.sequential ~a:1. dim 1 in
    let get_color = new_get_color_fun () in
    let names = Ndarray.to_array subspace_iterations |> Array.map (fun power␣
↪-> Format.sprintf "%d iteration%s" power (if power >= 2 then "s" else␣
↪"")) in
    let h = Plot.create path in

    Ndarray.iteri (fun i _ ->
        let errors = RNdarray.get_slice [[i] ; [] ; []] errors_range_finder␣
↪in
        (* Plot.scatter ~h (RMat.flatten @@ RMat.mul (RMat.ones 1 100) @@␣
↪ls) (RNdarray.flatten errors) ; *)
        let average_errors = RNdarray.mean ~axis:2 errors in
        Plot.plot ~h ~spec:[get_color () ; LineStyle 1 ; LineWidth 3.] ls␣
↪average_errors ;
    ) subspace_iterations ;
    Plot.plot ~h ~spec:[LineStyle 3 ; LineWidth 3.] ls errors_pca ;
    Plot.plot ~h ~spec:[LineStyle 2] ls errors_theory ;

    Plot.set_xlabel h "rank of the approximation" ;
    Plot.set_ylabel h "average error" ;
    Plot.legend_on h (Array.concat [names ; [|"PCA";"theoretical bound"|]]) ;

    h
```

```
val plot_errors :
  ?path:string ->
  int_ndarray -> RNdarray.arr * RMat.mat * RMat.mat -> Plot.handle = <fun>
```

Approximating a Laplacian

```
let subspace_iterations = int_range 5 ;;

Jupyter_notebook.clear_output () ;;

let errors_laplacian,
    errors_pca_laplacian,
    errors_theory_laplacian = compute_average_errors
                                ~show_steps:true
                                laplacian
                                100
                                subspace_iterations
```

```
let h = plot_errors
            subspace_iterations
            (errors_laplacian, errors_pca_laplacian, errors_theory_laplacian)

in
Plot.set_yrange h 0. 25. ;
Plot.set_title h "Average errors of randomized subspace iterations of the␣
 ↪100-dimensional Laplacian." ;

display_plot h
```



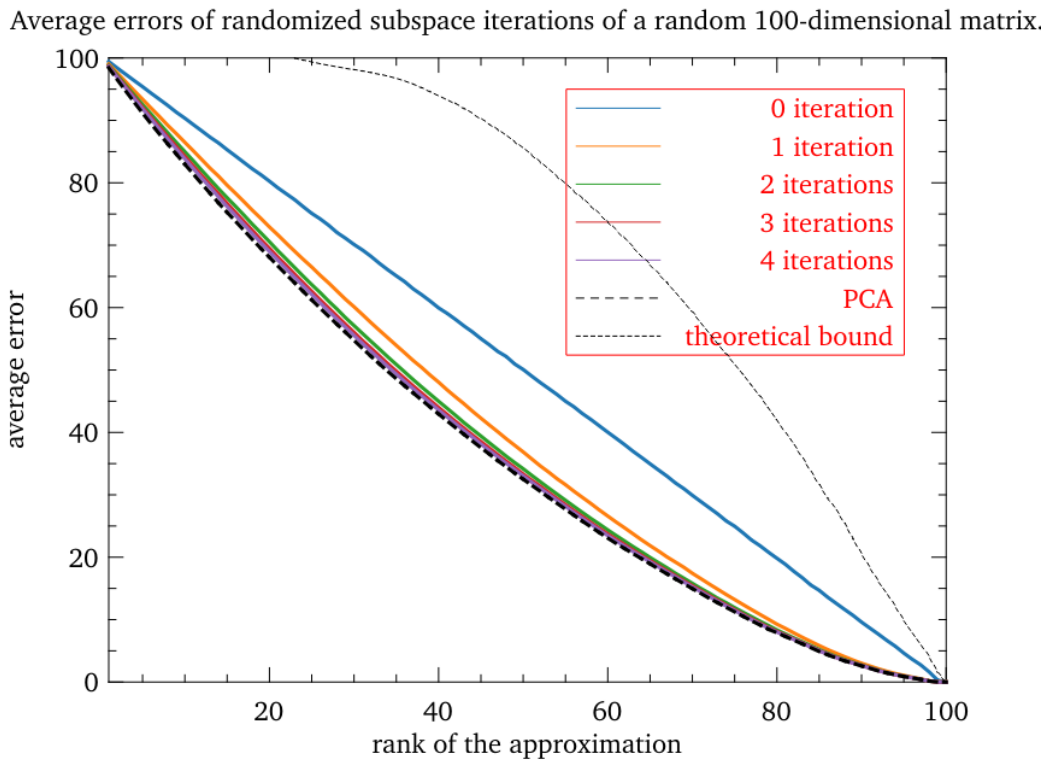

```
- : Jupyter_notebook.display_id = <abstr>
```

Here we can see how the randomized subspace iteration method can be used to decrease the influence of the lowest singular values on the output : the randomized range finder (the randomized subspace iteration with 0 iterations) is not really precise (although already way better than expected in theory, especially for small ranks), as it is quite far from the optimal error given by the principal component analysis (PCA), but only a few iterations are needed to get significantly closer to the optimal possible error.

Approximating a random matrix

```
let random_mat = RMat.gaussian 100 100 ;;

Jupyter_notebook.clear_output () ;;

let errors_random,
    errors_pca_random,
    errors_theory_random = compute_average_errors
                                ~show_steps:true
                                random_mat
                                100
                                subspace_iterations
```

```
let h = plot_errors
            subspace_iterations
            (errors_random, errors_pca_random, errors_theory_random)

in
Plot.set_title h "Average errors of randomized subspace iterations of a␣
 ↪random 100-dimensional matrix." ;
Plot.set_yrange h 0. 100. ;
display_plot h
```



```
- : Jupyter_notebook.display_id = <abstr>
```

Doing the same but with a random matrix, we reach the same conclusions as with the Laplacian.

4. Conclusion

Sampling the range of a matrix A probabilistically, it is possible to find a low-ranks matrices Q whose columns are orthogonal and generate subspaces that approximate the range of A almost optimally. Using deterministic factorization methods on Q^*A, it is then possible to get good low-rank approximations of A into factors faster than it would take to factorize A itself. This document described only the general ideas behind this process, and the reader may read the original article [1] for more advanced optimizations and more precise proofs.

These methods were implemented in an OCaml library that is modular : it may be used with any predefined linear algebra library. Although it does not implement all the algorithms and optimizations described in the original article [1], and is not fully tested, it is still completely functional, and was empirically shown to work correctly, and better than predicted by theory.

References

[1] N. Halko, P. G. Martinsson, and J. A. Tropp. "Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions". In: *SIAM Review* 53.2 (May 1, 2011), pp. 217–288. ISSN: 0036-1445. DOI: 10.1137/090771806. URL: https://doi.org/10.1137/090771806.

[2] Quentin Aristote. *OCaml Probabilistic Matrix Approximation*. URL: https://git.eleves.ens.fr/qaristote/ocaml-probabilistic-matrix-approximation.

A. Module `Fact` : The signature of the algorithms implemented in this library.

```
module type Fact =
  sig
    type elt
    type mat
    type complex_mat
```

A.1. Sampling the range of a matrix.

```
    type range_sampler = mat -> int -> mat
    val gaussian_range_sampler : range_sampler
```

`gaussian_range_sampler a n` returns `n` random independant gaussian vectors in the range of `a`.

A.2. Finding an approximation of the range of a matrix.

```
type range_finder = mat -> mat
val randomized_range_finder : ?sampler:range_sampler -> int -> range_finder
```

`randomized_range_finder ~sampler l a` returns an orthonormal family of size `l` that approximates the range of `a`.

It is obtained by sampling a family of `l` vectors in the range of `a` using `sampler` and computing its QR factorization. In particular, if `l` is greater than the rank of `a`, the outputed family will be truncated.

If `k` is the desired rank of the family, a good value for `l` is around `k+5~k+10`.

See also *Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions*, Algorithm 4.1[https://doi.org/10.1137/090771806]

```
val adaptive_randomized_range_finder :
  ?sampler:range_sampler -> ?r:int -> float -> range_finder
```

`adaptive_randomized_range_finder ~sampler ~r eps a` returns an orthonormal family of that approximates the range of `a` with precision `eps`.

If `a` is of size `m x n`, `q` is the resulting orthonormal matrix and `q^H` its adjoint, then `(id - qq^H)a` has norm smaller than `eps` with probability at least `1 - min(m,n)1e-r`.

It is obtained by sampling a vectors in the range of `a` using `sampler` and computing the QR factorisation of the resulting family until sufficient precision is reached.

See also *Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions*, Algorithm 4.2[https://doi.org/10.1137/090771806]

```
val randomized_subspace_iteration :
  ?sampler:range_sampler -> int -> int -> range_finder
```

`randomized_subspace_iteration ~sampler l q a` returns an orthonormal family of size `l` that approximates the range of `a`.

It is obtained by sampling a family of `l` vectors in the range of `(aa^H)^qa` (where `a^H` is the adjoint of `a`) using `sampler` and computing its QR factorization. In particular, if `l` is greater than the rank of `a`, the outputed family will be truncated. This method improves the accuracy of `Fact.Fact.randomized_range_finder`[A.2] for those matrices whose singular values decay slowly.

If `k` is the desired rank of the family, a good value for `l` is around `k+5~k+10`.

See also *Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions*, Algorithm 4.4[https://doi.org/10.1137/090771806]

```
val adaptive_randomized_subspace_iteration :
  ?sampler:range_sampler ->
  ?r:int -> int -> float -> range_finder
```

`adaptive_randomized_subspace_iteration ~sampler ~r q eps a` returns an orthonormal family of that approximates the range of `a` with precision `eps`.

If `a` is of size `m x n`, `q` is the resulting orthonormal matrix and `q^H` its adjoint, then `(id - qq^H)a` has norm smaller than `eps` with probability at least `1 - min(m,n)1e-r`.

It is obtained by sampling vectors in the range of `(aa^H)^qa` using `sampler` (where `a^H` is the adjoint of `a`) and computing the QR factorisation of the resulting family until sufficient precision is reached. This method improves the rate of accuracy of `Fact.Fact.adaptive_randomized_range_finder`[A.2] for those matrices whose singular values decay slowly.

See also *Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions*, Section 4.5[https://doi.org/10.1137/090771806]

A.3. Factorizing a matrix using (an approximation of) its range.

```
val direct_svd : mat ->
  mat -> mat * mat * mat
```

If `q` approximates the range of `a` with precision `eps`, `direct_eig a q` computes an approximate singular value decomposition `(u,sigma,v)` of `a` with precision `eps`.

See also *Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions*, Algorithm 5.1[https://doi.org/10.1137/090771806]

```
val direct_eig : mat ->
  mat -> complex_mat * complex_mat
```

If `q` approximates the range of a hermitian matrix `a` with precision `eps`, `direct_eig a q` computes an approximate decomposition `(u,lambda)` of `a` into eigenvectors `u` and eigenvalues `lambda` with precision `2eps`.

See also *Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions*, Algorithm 5.3[https://doi.org/10.1137/090771806]

```
val nystrom_eig : mat -> mat -> mat * mat
```

If `q` approximates the range of a positive semi-definite matrix `a` with precision `eps`, `direct_eig a q` computes an approximate decomposition `(u,lambda)` of `a` into eigenvectors `u` and eigenvalues `lambda` with precision `2eps`.

This method improves has roughly the same running time as `Fact.Fact.direct_eig`[A.3] but is typically much more accurate.

See also *Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions*, Algorithm 5.5[https://doi.org/10.1137/090771806]

A.4. Combining a range finder and a postprocessor.

```
val approximate : range_finder ->
  (mat -> mat -> 'a) -> mat -> 'a
```

`postprocessor range_finder postprocessor a` returns a probabilistic approximation of `a` using `range_finder` to find its range and `postprocessor` to derive its approximation.

```
  end

module Make :
  functor (* :   sig


  end ) -> Fact
```

Generate probabilistic matrix approximation functions from a linear algebra library.

```
module S :
  Fact  with type elt = Owl.Dense.Matrix.S.elt and type mat = Owl.Dense.Matrix.S.mat
and type complex_mat = Owl.Dense.Matrix.C.mat
```

Probabilistic matrix approximation functions for Owl's `float32` matrices.

```
module D :
  Fact  with type elt = Owl.Dense.Matrix.D.elt and type mat = Owl.Dense.Matrix.D.mat
and type complex_mat = Owl.Dense.Matrix.Z.mat
```

Probabilistic matrix approximation functions for Owl's `float64` matrices.

```
module C :
  Fact  with type elt = Owl.Dense.Matrix.C.elt and type mat = Owl.Dense.Matrix.C.mat
and type complex_mat = Owl.Dense.Matrix.C.mat
```

Probabilistic matrix approximation functions for Owl's `complex32` matrices.

```
module Z :
  Fact  with type elt = Owl.Dense.Matrix.Z.elt and type mat = Owl.Dense.Matrix.Z.mat
and type complex_mat = Owl.Dense.Matrix.Z.mat
```

Probabilistic matrix approximation functions for Owl's `complex64` matrices.

B. Module `Matrix` : The interface for linking with a linear algebra library.

```
module type Matrix =
  sig
```

Unless specified otherwise, the types and values are the same as in the Owl library.

```
type elt
type mat
type complex_mat
```

B.1. Conversions between real and complex values.

```
val to_float : elt -> float
```

Convert a real number of type `elt` to `float`.

```
val cast : mat -> complex_mat
```

Convert a matrix whose elements are of type `elt` to a complex matrix.

B.2. Basic properties of matrices.

```
val shape : mat -> int * int
```

B.3. Matrix creation.

```
val zeros : int -> int -> mat
val eye : int -> mat
val gaussian : ?mu:elt ->
  ?sigma:elt -> int -> int -> mat
```

B.4. Algebra with matrices.

```
val sub : mat -> mat -> mat
val dot : mat -> mat -> mat
val dot_complex : complex_mat ->
  complex_mat -> complex_mat
val mul : mat -> mat -> mat
val div_scalar : mat -> elt -> mat
val triangular_solve : mat -> mat -> mat
```

`triangular_solve a b` returns the solution `x` of the linear system `x * a = b`.

B.5. Advanced properties of matrices.

```
val max' : mat -> elt
val l2norm : ?axis:int -> ?keep_dims:bool -> mat -> mat
val l2norm' : mat -> elt
```

B.6. Operations on the rows and columns of matrices.

```
val get_slice : int list list -> mat -> mat
val transpose : mat -> mat
val ctranspose : mat -> mat
val concat_horizontal : mat -> mat -> mat
```

B.7. Standard matrix factorizations.

```
val qr : mat -> mat * mat
val svd : mat ->
  mat * mat * mat
val eig : mat -> complex_mat * complex_mat
val chol : mat -> mat
end
```

```
module S :
  Matrix  with type elt = Owl.Dense.Matrix.S.elt and type mat = Owl.Dense.Matrix.S.mat
and type complex_mat = Owl.Dense.Matrix.C.mat
```

Linear algebra for Owl's `float32` matrices.

```
module D :
  Matrix  with type elt = Owl.Dense.Matrix.D.elt and type mat = Owl.Dense.Matrix.D.mat
and type complex_mat = Owl.Dense.Matrix.Z.mat
```

Linear algebra for Owl's `float64` matrices.

```
module C :
  Matrix  with type elt = Owl.Dense.Matrix.C.elt and type mat = Owl.Dense.Matrix.C.mat
and type complex_mat = Owl.Dense.Matrix.C.mat
```

Linear algebra for Owl's `complex32` matrices.

```
module Z :
  Matrix  with type elt = Owl.Dense.Matrix.Z.elt and type mat = Owl.Dense.Matrix.Z.mat
and type complex_mat = Owl.Dense.Matrix.Z.mat
```

Linear algebra for Owl's `complex64` matrices.

C. Module `Test`

```
module type MatrixToTest =
  sig

    include Matrix.Matrix

  end

module type Test =
  sig

    type elt

    type mat

    type range_finder
```

C.1. Computing an error.

```
    val error_factorization : mat list -> mat -> elt

    val error_range_finder : range_finder -> mat -> elt
```

C.2. Testing the precision of an algorithm.

```
    val test_error : elt -> float -> bool

    val test_adaptive_range_finder :
      (float -> range_finder) -> float -> mat -> bool
```

```
  end

module Make :
  functor (M : MatrixToTest) -> functor (F : Fact.Fact  with type elt = M.elt
and type mat = M.mat) -> Test
module S :
  Test  with type elt = Owl.Dense.Matrix.S.elt and type mat = Owl.Dense.Matrix.S.mat
and type range_finder = Fact.S.range_finder
```

Library testing for Owl's `float32` matrices.

```
module D :
  Test  with type elt = Owl.Dense.Matrix.D.elt and type mat = Owl.Dense.Matrix.D.mat
and type range_finder = Fact.D.range_finder
```

Library testing for Owl's `float64` matrices.

```
module C :
  Test  with type elt = Owl.Dense.Matrix.C.elt and type mat = Owl.Dense.Matrix.C.mat
and type range_finder = Fact.C.range_finder
```

Library testing for Owl's `complex32` matrices.

```
module Z :
  Test  with type elt = Owl.Dense.Matrix.Z.elt and type mat = Owl.Dense.Matrix.Z.mat
and type range_finder = Fact.Z.range_finder
```

Library testing for Owl's `complex64` matrices.