École Polytechnique Fédérale de Lausanne

Bachelor's Project

Baptiste LUCAS

EFFICIENT RANDOMIZED MATRIX-BASED ALGORITHMS FOR FUNCTION APPROXIMATION AND LOW-RANK KERNEL INTERACTIONS

Department of Mathematics

Supervisor: Hei Yin LAM

Spring Semester - 2022

Summary

1 Introduction

The main goal of this report is to approximate a bivariate function by using Chebyshev interpolation. Here we only look at two dimensions, but multivariate function approximation (i.e. with more than 2 variables) are used in numerous different domains, such as numerical analysis, optimization, uncertainty quantifications etc.

In our case we will use Chebyshev interpolation to approximate kernel matrices (also called interaction matrices). Their entries correspond to evaluations of a function (called kernel function) which allows to model pairwise interactions within a given set of points. This process makes part of kernel methods which have lots of applications : machine learning, solvers for integral equations, geostatistics, numerical analysis and many others. Since kernels are not the main objective of this project (and we miss some tools to discuss them), we will not go into details.

This Chebyshev approximation will be optimized by using low-rank matrix approximation, which allows to spare computing time while keeping a certain accuracy. Kernel matrices are generally dense (thus hard to compute and store), hence low-rank matrix approximation will accelerate these calculations. We will first decompose the initial interaction matrix into three sub-matrices thanks to Chebyshev approximation (less kernel evaluations will be done), then low-rank will be used on the resulting central matrix to get a better storage. The efficiency of our low-rank method will be verified through several analysis on Matlab.

In addition numerical experiments will support each section of this article by testing algorithms on different functions, observing error convergence or computing time variations.

This report refers to the following paper : Arvind K. Saibaba · Rachel Minster · Misha E. Kilme, "Efficient randomized tensor-based algorithms for function approximation and low-rank kernel interactions"[1].

Since tensors are out of scope in this project we will only consider matrices, that is tensors in two dimensions.

2 Chebyshev approximation

2.1 Chebyshev motivation

Chebyshev interpolation allows to minimize the Runge's phenomenon. The latter is observed when we interpolate a function on $[-1,1]$ with $n+1$ equidistant nodes $x_i = -1+2i/n$ for $i = 0, \ldots, n$. In particular if we interpolate the Runge's function $f(x) = 1/(1+25x^2)$ with the previous discretization we'll always obtain a bigger error $\|f - p_n\|_\infty$ as we increase N, the number of nodes we use.

This can be reformulated as following :
Given $f : [-1,1] \to \mathbb{R}$, we wish to interpolate this function at $n+1$ points $\{x_i\}_{i=0}^n$ by using an interpolation polynomial p_n (of degree n which interpolates each node x_i for $i = 0, \ldots, n$). Then the error is defined as[4] :

$$\|f - p_n\|_\infty \leq \|\omega_{n+1}\|_\infty \frac{\|f^{(n+1)}\|_\infty}{(n+1)!}$$

where $\|\omega_{n+1}\|_\infty = \max\limits_{x\in[-1,1]} |(x-x_0)...(x-x_n)|$.

The latter is minimized by taking the $n+1$ Chebyshev nodes, which are $\xi_i = \cos(\frac{2i+1}{2n+2}\pi)$ for $i = 0, ..., n$. As expected by taking these interpolation points we minimize the Runge's phenomenon.

Now if we want to do the same for a function f defined on $[a,b]$, we just use the mapping function $I_{[a,b]} : [-1,1] \to [a,b]$ defined as :

$$I_{[a,b]}(x) = (x+1)\frac{b-a}{2} + a$$

and $[a,b]$ will be discretized by the $n+1$ points $\eta_i = I_{[a,b]}(\xi_i)$ for $i = 0, ..., n$.

2.2 Chebyshev interpolation

Given an analytic function $f : [-1,1] \to \mathbb{R}$ there exists Chebyshev coefficients[3] $\{a_k\}_{k=0}^{n-1}$ such that

$$f(x) \approx p_{n-1} = \sum_{k=0}^{n-1} a_k T_k(x) \tag{1}$$

where $T_k(x) = \cos(k \arccos(x))$ is the Chebyshev polynomial of the first kind of degree k.

The previous sum is the Chebyshev polynomial of degree $n-1$ interpolating f. Now we rewrite this same expression as in [1] :

We define the bivariate interpolating polynomial $S_n^{[a,b]}$ on $[a,b]^2$ of degree $n-1$ as following[1] :

$$S_n^{[a,b]}(x,y) = \frac{1}{n} + \frac{2}{n}\sum_{k=1}^{n-1} T_k(I_{[a,b]}^{-1}(x))T_k(I_{[a,b]}^{-1}(y)) \tag{2}$$

Given $f : [a,b] \to \mathbb{R}$ interpolated with n Cheyshev nodes we then obtain the following formula, which is a rewriting of (1)

$$\hat{f}_{n-1}(x) = \sum_{i=1}^{n} f(\eta_i) S_n^{[a,b]}(\eta_i, x)$$

where $\{\eta_i\}_{i=1}^n$ are the n Chebyshev nodes discretizing $[a,b]$.

In the coming theorem we will do exactly the same but for a function taking entries in $\mathbb{R}^2$.

2.3 Theorem

Given $f : [a,b] \times [c,d] \to \mathbb{R}$, the Chebyshev interpolation polynomial for f using n Chebyshev nodes on each interval is defined as :

$$\hat{f}_{n-1}(x,y) = \sum_{i=1}^{n} \sum_{j=1}^{n} f(\eta_i, \mu_j) S_n^{[a,b]}(\eta_i, x) S_n^{[c,d]}(\mu_i, y) \tag{3}$$

where $\{\eta_i\}_{i=1}^n$ (resp. $\{\mu_j\}_{j=1}^n$) are the n Chebyshev nodes discretizing $[a,b]$ (resp. $[c,d]$). Also $S_n^{[a,b]}$ and $S_n^{[c,d]}$ are the same as in (2).

Proof.

Let's only consider the x-dimension of our function f, i.e. fix $y \in [c,d]$ and define $g_y(x) : [a,b] \to \mathbb{R}$ such that $g_y(x) = f(x,y)$. We can then make use of the 1-dimension Chebyshev interpolation polynomial on $[a,b]$, i.e.

$$\hat{g_y}(x) = \sum_{i=1}^{n} g_y(\eta_i) S_n^{[a,b]}(\eta_i, x)$$

where $\{\eta_i\}_{i=1}^n$ are the n Chebyshev nodes discretizing $[a,b]$. Now we also interpolate each $g_y(\eta_i) = f(\eta_i, y)$ by making use this time of the 1-dimension Chebyshev interpolation polynomial on $[c,d]$, i.e. we do it w.r.t. y when we look at $\hat{f}(\eta_i, y)$:

$$\hat{g_y}(\eta_i) = \hat{f}(\eta_i, y) = \sum_{j=1}^{n} f(\eta_i, \mu_j) S_n^{[c,d]}(\mu_j, y)$$

where $\{\mu_j\}_{j=1}^n$ are the n Chebyshev nodes discretizing $[c,d]$.

Finally by combining the two interpolations formula above, i.e. we do again an approximation of $\hat{g_y}(x)$, denoted $\hat{\hat{g_y}}(x)$ as following :

$$\hat{f}_{n-1}(x,y) = \hat{\hat{g_y}}(x) = \sum_{i=1}^{n} \hat{g_y}(\eta_i) S_n^{[a,b]}(\eta_i, x) = \sum_{i=1}^{n} \sum_{j=1}^{n} f(\eta_i, \mu_j) S_n^{[c,d]}(\mu_j, y)(\eta_i) S_n^{[a,b]}(\eta_i, x)$$

□

2.4 Interaction matrix

The following is a small motivation concerning the approximation of what we call an interaction matrix, which is an essential branch in machine learning. Since it is not the main goal of this project we will not go into details.

Given $\mathcal{X} = \{x_1, \dots, x_p\}$ and $\mathcal{Y} = \{y_1, \dots, y_q\}$ respectively enclosed by $[a, b] \subset \mathbb{R}$ and $[c, d] \subset \mathbb{R}$ we wish to separate both sets using a reasonable amount of computing time.

Unfortunately this problem is often non linear, hence it requires a lot of time to compute the separation. To avoid this phenomenon we will use the kernel trick[7] : a method which allows to find a linear classifier in a set of higher dimension while avoiding other calculations. Back in the initial set the classifier we found will obviously not be linear. We will not go into details but it spares more computing time than without doing the kernel trick.

Roughly the kernel trick makes use of a function f, which is called kernel function and allows to avoid some computations in the set of higher dimension. Assuming that f verifies different properties (given by the Mercer theorem[8]) we can rewrite the function as a dot product between two elements in the higher dimensional set, without computing it explicitly since f is also defined in another way. Hence the kernel trick avoids the calculations in the set of higher dimension since a certain function f exists and computes the required quantity in a different way.

Going back to our sets $\mathcal{X}$ and $\mathcal{Y}$ we can model the interactions between each possible pair of points as a call to a smooth function f, i.e. for $i = 1, \dots, p$ and $j = 1, \dots, q$ we compute the interaction between $x_i \in \mathcal{X}$ and $y_j \in \mathcal{Y}$ by computing the quantity $f(x_i, y_j)$. Hence we can define the interaction matrix (also called the kernel matrix) as following :

$$\mathcal{K}(\mathcal{X}, \mathcal{Y}) = \begin{bmatrix} f(x_1, y_1) & \dots & f(x_1, y_q) \\ \vdots & & \vdots \\ f(x_p, y_1) & \dots & f(x_p, y_q) \end{bmatrix} \in \mathbb{R}^{p \times q} \tag{4}$$

Therefore by assuming that $\mathcal{X}$ and $\mathcal{Y}$ are well-separated it will be possible to approximate the kernel function f to compute $\mathcal{K}(\mathcal{X}, \mathcal{Y})$ in a faster way. Indeed the goal is to have the less possible kernel evaluations to do (i.e. a call to f is a kernel evaluation since f is the kernel function). Here we would have to do $p \cdot q$ kernel evaluations, which can be quickly too much since the number of interaction points could be really big. The same reasoning works for the complexity and the storage of $\mathcal{K}(\mathcal{X}, \mathcal{Y})$, both being in $O(pq)$.

2.5 Matrix form

Hence we can spare computing time by making use of the Chebyshev interpolation we defined above. Thus we choose n Chebyshev nodes on each interval, i.e. on $[a, b]$ and $[c, d]$.

Now we define

$$U = \begin{bmatrix} s_1(x_1) \\ \vdots \\ s_1(x_p) \end{bmatrix} \in \mathbb{R}^{p \times n} \quad \text{and} \quad V = \begin{bmatrix} s_2(y_1) \\ \vdots \\ s_2(y_q) \end{bmatrix} \in \mathbb{R}^{q \times n}$$

where

$$s_1(\lambda)^\top = \begin{bmatrix} S_n^{[a,b]}(\eta_1, \lambda) \\ \vdots \\ S_n^{[a,b]}(\eta_n, \lambda) \end{bmatrix} \in \mathbb{R}^{n\times 1} \quad \text{and} \quad s_2(\lambda)^\top = \begin{bmatrix} S_n^{[c,d]}(\mu_1, \lambda) \\ \vdots \\ S_n^{[c,d]}(\mu_n, \lambda) \end{bmatrix} \in \mathbb{R}^{n\times 1}$$

with $S_n^{[a,b]}(x,y)$ and $S_n^{[c,d]}(x,y)$ as defined previously[2]. By construction the side matrices U and V map the interaction points to Chebyshev grids.

Finally we let

$$M = \begin{bmatrix} f(\eta_1, \mu_1) & \dots & f(\eta_1, \mu_n) \\ \vdots & & \vdots \\ f(\eta_n, \mu_1) & \dots & f(\eta_n, \mu_n) \end{bmatrix} \in \mathbb{R}^{n\times n}$$

be such that we get the matrix notation for (3) :

$$\mathcal{K}(\mathcal{X}, \mathcal{Y}) \approx UMV^\top$$

On the one hand observe that no kernel evaluations are done within side matrices, but they still depend on the number of Chebyshev nodes. On the other hand M is the only matrix where kernel evaluations are done : precisely we have n^2 kernel evaluations, which is really smaller than the $p \cdot q$ ones we had here[4].

Note that if we keep the same number of Chebyshev nodes, i.e. by fixing n but changing the sizes of $\mathcal{X}$ and $\mathcal{Y}$ then we will not have to compute M again. Indeed the latter only depends on the number of Chebyshev nodes we use. This can be very useful when one always has more interaction points in a model, then this form allows a gain in computing time.

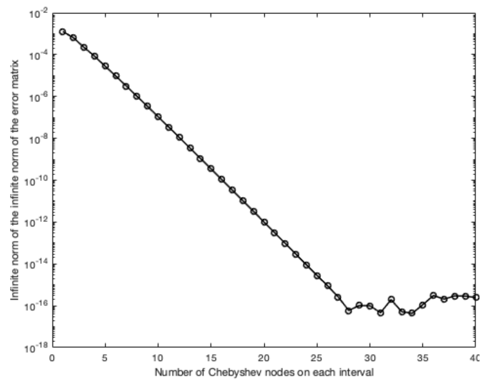
As we keep U, M, V at the end we get a storage (same complexity, which just corresponds to the formations of the three matrices) in $O(n^2 + n(p+q))$.

2.6 Matlab implementation

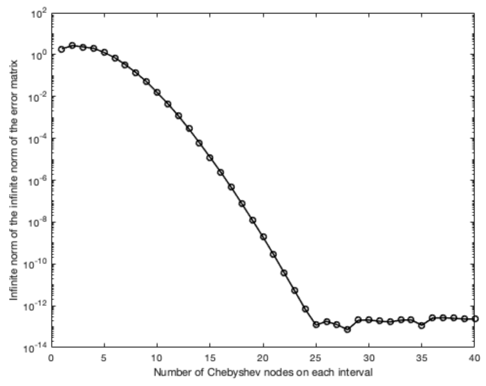
We implemented an algorithm that approximates any function $f : [a,b] \times [c,d] \to \mathbb{R}$ by using the matrix process described above. Now we test our method on 4 different functions (f is the peaks function from Matlab, also used in [6]) :

$$\begin{aligned} f_1(x,y) &= \frac{1}{1 + 25(x^2 + y^2)} \\ f_2(x,y) &= \sin(x+y) \\ f_3(x,y) &= \tanh(3(x+y)) \\ f(x,y) &= 3(1-x)^2 e^{-x^2-(y+1)^2} - 10(\frac{x}{5} - x^3 - y^5)e^{-x^2-y^2} - \frac{1}{3}e^{-(x+1)^2-y^2} \end{aligned}$$

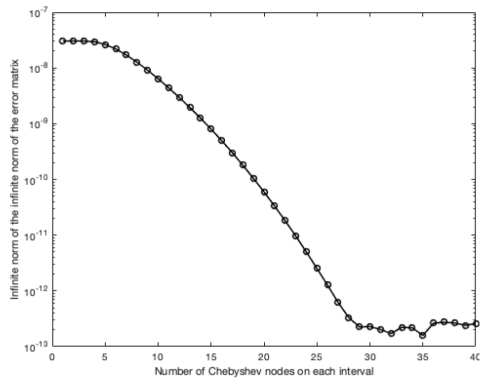
To be precise we test each of them on a 20×20 grid of interaction points. The x-axis is the number of Chebyshev nodes we use on each interval to do our approximation. The y-axis (logarithmic scale) represents the error in infinite norm : we compute the error on the 400 points and then we take the maximum among them in absolute value.



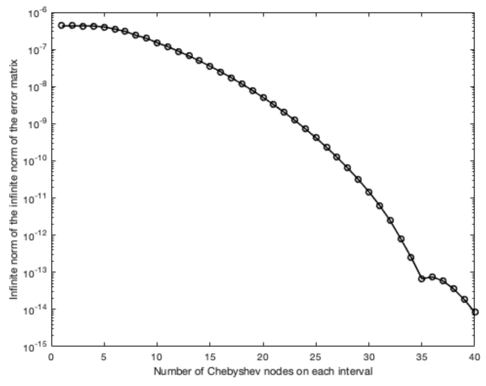
(a) f_1



(b) f_2



(c) f_3



(d) f

The error decay of f_2, f_3 and f is more than exponential whereas the one for f_1 has an exponential convergence speed. This is due to the choice of radius for each Bernstein ellipse, which is in one-to-one correspondence with each function. We will not go into details, but thanks to [5] we have access to a lemma (7.3.3) explaining the different convergence speed regarding the domain of a function and where it can be extended while keeping the function analytic. Here is the statement :

Lemma 7.3.3 :
Let f $\in C^0([-1,1]^2)$ be a function which can be extended to an analytic function f^* on the Cartesian product of Bernstein ellipses $S = \mathcal{E}_{0,1}^{\rho_1} \times \mathcal{E}_{0,1}^{\rho_2}$. Then for $n \in$ with Chebyshev approximation p_n[1] the error is bounded as following :

$$\|f - p_n\|_{C^0([-1,1]^2)} \leq 4\sqrt{2} \cdot \frac{\rho_{\min}^2}{\rho_{\min}^2 - 1} \cdot \frac{1}{\rho_{\min}^n} \cdot M_\rho(f)$$

where $M_\rho(f) = \max_{z \in S} |f^*(z)|$.

We will not prove this statement but the proof is available in [5].

This also works for our approximation, which is more or less constrained by the same properties as the function it interpolates. The more extensible the domain, the faster the convergence of the (approximation) function.

3 Low-rank approximation

3.1 Motivation

The main goal of a low-rank approximation is to find a matrix $Y \in \mathbb{R}^{m\times n}$ of low rank r which approximates well a given matrix $X \in \mathbb{R}^{m\times n}$. In other words we wish to find :

$$Y = \operatorname{argmin}_{\hat{X} \text{ s.t. } \operatorname{rank}(\hat{X})\leq r} \left\| X - \hat{X} \right\|_F$$

However finding the minimum can take quite lot of computing time. One of the best approximations is the truncated SVD. Indeed if we take a small target rank r with $\hat{X} = U(:, 1 : r)\Sigma(1 : r, 1 : r)V(:, 1 : r)^\top$ (where $U\Sigma V^\top = X$ is the SVD decomposition of the latter) then we get[2] :

$$\left\| X - \hat{X} \right\|_F^2 = \sum_{i=r+1}^{m} \sigma_i^2$$

where $\sigma_1, \dots, \sigma_m$ are the singular values of X.

Unfortunately this kind of low-rank approximation requires a lot of computing time (SVD is in $O\big(\max(m, n) \cdot \min^2(m, n)\big)$). Hence we will use another algorithm which approximates well a matrix. It is called `method1.m`[1].

3.2 Our low-rank approximation

The latter returns two matrices A_1, A_2 with two index sets $\mathcal{J}_1, \mathcal{J}_2$ such that we have two low-rank interpolary decompositions (IDs). In other words we get two approximations $X \approx A_1X(\mathcal{J}_1, :)$ and $X^\top \approx A_2X^\top(\mathcal{J}_2, :)$ which combined give $X \approx A_1X(\mathcal{J}_1, \mathcal{J}_2)A_2^\top$.

These two IDs come from two calls to another algorithm within `method1.m`, which is called Randomized Row Interpolatory Decomposition (RRID).

Given $X \in \mathbb{R}^{m\times n}$, target rank r and oversampling parameter p, the whole goal of RRID is to find $\Lambda \in \mathbb{R}^{m\times(r+p)}$ (called factor matrix) and an index set $\mathcal{J}$ s.t. $X \approx \Lambda X(\mathcal{J}, :)$ where $|\mathcal{J}| = r + p$ and $\Lambda(\mathcal{J}, :) = I_{(r+p)}$ is the $(r + p) \times (r + p)$ identity matrix.

This is exactly what an ID[2] does : given $X \in \mathbb{R}^{m\times n}$ of rank k it finds a matrix $\Lambda \in \mathbb{R}^{m\times k}$ s.t. $X \approx \Lambda X(\mathcal{J}, :)$ where $\Lambda(\mathcal{J}, :) = I_k$. In other words it finds a collection $\mathcal{J}$ of k rows spanning the range of X.

Thus we explain now how RRID produces this ID given $X \in \mathbb{R}^{m\times n}$, target rank r and oversampling parameter p. It works in two steps :

Step 1 : find a matrix $Q \in \mathbb{R}^{m\times(r+p)}$ s.t. $X \approx QQ^\top X$

In other words we want $\mathcal{R}(Q) \approx \mathcal{R}(X)$ i.e. the range of X is well approximated by Q.

First we draw a Gaussian random matrix $\Omega \in \mathbb{R}^{n\times(r+p)}$ which we multiply with X, giving $Y = X\Omega \in \mathbb{R}^{m\times(r+p)}$. Hence it gives linear combinations of random columns of X.

Finally we compute a thin QR-decomposition of Y, giving the desired Q s.t. $X \approx QQ^\top X$. This first step can also be named as a randomized range finder[2].

Step 2 : select $(r+p)$ rows of Q s.t. we find an ID of X

The final approximation of step 1 looks like an ID, what is missing is the index selection. What seems natural is to play with Q and X in the last expression of step 1 to make the ID appear.

The idea is to transform $QQ^\top$ into a factor matrix F associated with a certain index set $\mathcal{J}$ s.t. $F(\mathcal{J},:) = I_{(r+p)}$. Usually when doing QR on $Y \in \mathbb{R}^{m\times(r+p)}$ we obtain an orthogonal square matrix $Q \in \mathbb{R}^{m\times m}$. However we did a thin QR on Y, leading to a matrix $Q \in \mathbb{R}^{m\times(r+p)}$ which is neither orthogonal nor square, so it cannot be invertible. To get this factor matrix we will thus search an index set $\mathcal{J}$ s.t. $Q(\mathcal{J},:) \in \mathbb{R}^{(r+p)\times(r+p)}$ is orthogonal.

To do this we operate directly on the “inverse“ i.e. $Q^\top$: we perform a column pivoted thin QR on $Q^\top$ (hence it amounts to select rows on Q). Since we want $|\mathcal{J}| = r+p$ we only keep the first $(r+p)$ columns of the permutation matrix (call it P_1) resulting from this decomposition.

We know that each column of P_1 is a canonical vector. Each of them has only one non-zero value, which is 1. Hence we search the index corresponding to the 1 and store it in an index set, which will form at the end our $\mathcal{J}$. By taking the rows of Q corresponding to this index set we therefore obtain $(r+p)$ well-conditioned rows of Q, making the inversion possible i.e. $Q(\mathcal{J},:)(Q(\mathcal{J},:))^{-1} = I_{(r+p)}$.

This finally gives the factor matrix $F = Q(Q(\mathcal{J},:))^{-1}$ and the ID going with it.

Here is the pseudocode for both algorithms (called `method1.m` and `RRID.m` in Matlab) :

Algorithm 1 Method1 : Randomized Interpolatory Matrix Decomposition

```
Input: matrix $X \in \mathbb{R}^{n\times n}$, target rank $r$, oversampling parameter $p$ such that $r+p \leq n$
Output: factor matrices $A_1, A_2$ and central matrix $X(\mathcal{J}_1, \mathcal{J}_2)$
1: Compute $[A_1, \mathcal{J}_1] = \text{RRID}(X, r+p)$
2: Compute $[A_2, \mathcal{J}_2] = \text{RRID}(X^\top, r+p)$
3: Form $X(\mathcal{J}_1, \mathcal{J}_2)$
```

Algorithm 2 RRID : Randomized Row Interpolatory Decomposition

```
Input: matrix $X \in \mathbb{R}^{m\times n}$, target rank $r$, oversampling parameter $p$ s.t. $r+p \leq \min\{m,n\}$
Output: factor matrix $F$ with index set $\mathcal{J}$ s.t. $|\mathcal{J}| = r+p$
1: Generate standard random Gaussian matrix $\Omega \in \mathbb{R}^{n\times(r+p)}$
2: Form $Y = X\Omega \in \mathbb{R}^{m\times(r+p)}$
3: Apply thinQR on $Y$, that is $[Q,R]$=$Y$
4: Apply column-pivoted thinQR on $Q^\top$ and keep $P_1 \in \mathbb{R}^{m\times(r+p)}$ to get $\mathcal{J}$, i.e.

       $Q^\top[P_1\ P_2] = Q_{\text{new}}[R_1\ R_2]$

5: Form factor matrix $F = Q(Q(\mathcal{J},:))^{-1}$
```

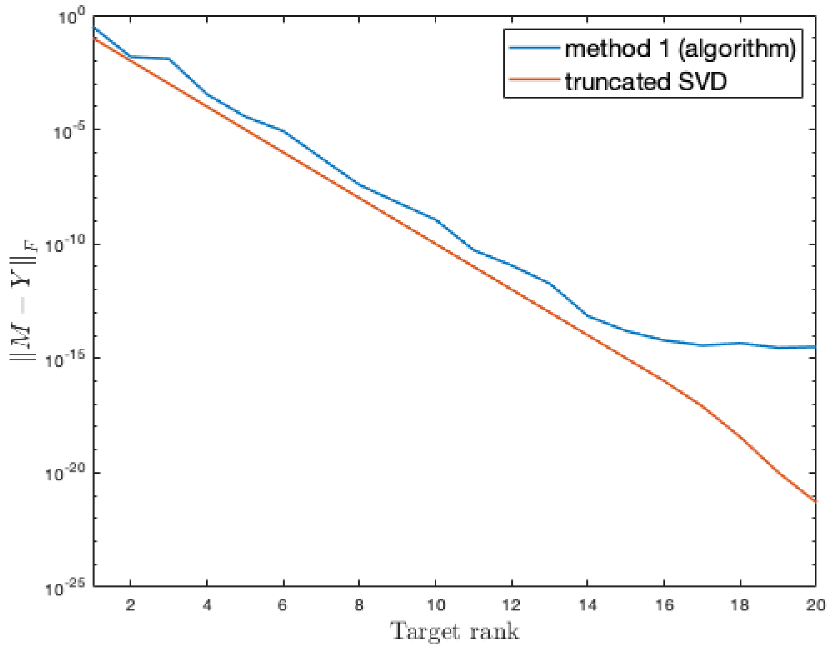
3.3 Matlab implementation

To begin we build a random 200×200 low rank matrix $M = UEV^\top$ where U, V are orthonormal normal matrices and

$$E = \begin{bmatrix} 1 & & & \\ & 10^{-1} & & \\ & & \ddots & \\ & & & 10^{1-n} \end{bmatrix} \in \mathbb{R}^{200\times 200}$$

This matrix construction allows to get a pretty decay of the singular values.

Then we test `method1.m` by calling it 20 times since we increase the target rank r at each call, until $r = 20$. We compare its convergence speed with the one from the truncated SVD of M (i.e. for instance by taking $r = 10$ the truncated SVD will be $U\hat{E}V^\top$ where $\hat{E}$ has the same first 10 coefficients as E and all the rest is zero). It gives rise to :



(a) main_2.m

Hence the error decay in frobenius norm of `method1.m` is reasonable : it has the same convergence speed as the truncated SVD. However the error of the latter is a bit smaller than the one for `method1.m` : it is normal since the low-rank approximation is expected to be better by using truncated SVD.

4 Low-rank on Chebyshev approximation

4.1 Chebyshev and low-rank together

Using `method1.m` as described in the previous section we can apply low-rank approximation to our Chebyshev interpolation matrix form :

Usually for $p \cdot q$ interaction points the matrix algorithm gives

$$U \in \mathbb{R}^{p\times n}, V \in \mathbb{R}^{q\times n} \text{ and } M \in \mathbb{R}^{n\times n} \text{ such that } \mathcal{K}(\mathcal{X},\mathcal{Y}) \approx UMV^\top$$

Now we call `method1.m` on M with a certain target rank r and oversampling parameter k, returning

$$A_1 \in \mathbb{R}^{n\times(r+k)}, A_2 \in \mathbb{R}^{n\times(r+k)} \text{ and } \hat{M} = M(\mathcal{J}_1,\mathcal{J}_2) \in \mathbb{R}^{(r+k)\times(r+k)}$$

where $\mathcal{J}_1, \mathcal{J}_2$ are the index sets found by `RRID.m` inside the call to `method1.m`. Finally we get the following low-rank approximation :

$$\mathcal{K}(\mathcal{X},\mathcal{Y}) \approx \hat{U}\hat{M}\hat{V}^\top \qquad \text{where } \hat{U} = UA_1 \text{ and } \hat{V} = VA_2$$

Again if we keep n fixed we will not have to compute again M (as explained in matrix form section) and thus we call `method1(M,r,k)` only once, even if the kernel matrix changes. Hence in this case we just have to compute again U, V and the matrix products going with it.

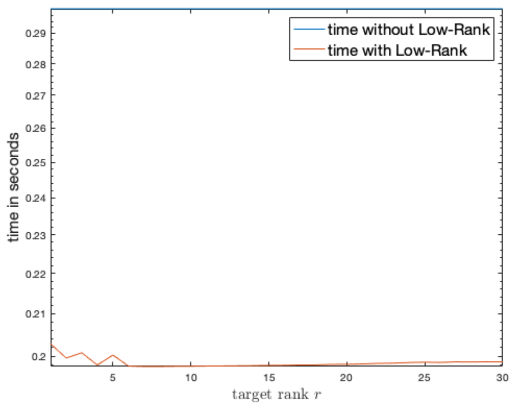
Since we first compute U, M, V (even if then we call `method1.m`) we do n^2 kernel evaluations. At the end we keep $\hat{U}, \hat{M}, \hat{V}$ hence the storage of this method is in $O(r^2 + r(p+q))$. The complexity[1] of the whole operation is in $O(rn^2 + r(1+n)(p+q))$.

4.2 Matlab implementation

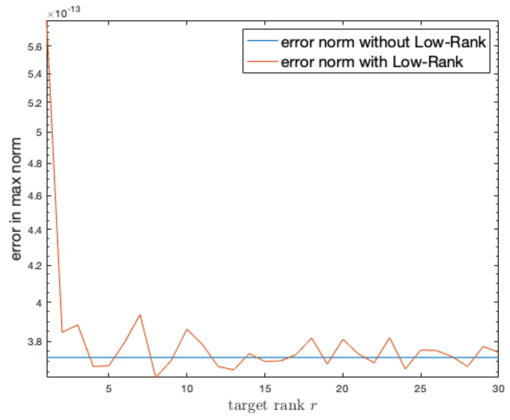
In order to test well the theory we study the computing time with and without Low-Rank approximation, but also the approximation error in max-norm.

In this case we take the peaks function f (which is often used in Matlab) interpolated with 200 Chebyhsve nodes on each interval. The interaction points are between $[-3,3] \times [-3,3]$, distributed equally using `linspace(-3,3,50)` for both dimensions, hence the interaction matrix is such that $\mathcal{K}(\mathcal{X},\mathcal{Y}) \in \mathbb{R}^{50\times 50}$.

Below on the left we plot the computing time w.r.t increasing target rank r until 30, as on the right for the error in max-norm.



(a) Time without Low-Rank is 0.298274



(b) Note the slope before $r = 2$

The blue line is constant since it does not depend on the target rank (on the left figure it is the equation $y = 0.298274$ thus at the top-border of the graph).

Note the decay until $r = 2$: it is because the “real“ target rank of M is approximately 3, you can find it by using `rangefinder.m` (here the decay stops at 2 since we have an oversampling parameter $k = 2$ added to target rank, without it the decay would stop at $r = 3$ or $r = 4$). Hence the approximation error can be important before reaching this rank. Here we were lucky but generally due to the randomness of the algorithm we don't have an important decay but rather oscillations (if there is no oversampling parameter then the behavior should be similar to the one observed here).

The oscillatory behavior on the right figure is due to the randomness coming from our algorithm `method1.m` (line 1 in pseudocode of RRID when we draw Gaussian matrix Ω). This explanation also applies to the left figure : the computing time doesn't only increase or only decrease in a monotonous way but it switches between both behaviors, which is due to randomness within the algorithm. Here we took an oversampling parameter of 2, but if you take 0 then the oscillations will be really small (try on Matlab and zoom on the resulting graph).

Hence we clearly see that Low-Rank saves time but is more or less accurate, which is normal since theoretically Low-Rank is less precise but faster.

5 Conclusion

Hence we introduced what a kernel matrix is, and how to approximate it in a fast way. More precisely we made use of Chebyshev interpolation to get close to the entries of the interaction matrix. It also allowed to avoid additional computing time and storage.

The several numerical experiments showed different error decays depending on the type of function we approximate. These various behaviors were explained thanks to a lemma from [5].

Then in order to optimize this approximation we built an algorithm with an auxiliary program. This calls a smart part of the Chebyshev approximation, i.e. if the number of Chebyshev nodes is fixed then we make use of the algorithm only once, even if the number of interaction points changes.

Finally in this report the fastest way to approximate the kernel matrix is -as expected- by employing the low-rank algorithm described above. The numerical experiments showed that the approximation is still good while we clearly spare more computing time by using this method in addition to the others.

References

[1] Arvind K. Saibaba · Rachel Minster · Misha E. Kilme. *Efficient randomized tensor-based algorithms for function approximation and low-rank kernel interactions. 2021.* URL: https://arxiv.org/pdf/2107.13107.pdf.

[2] P.-G. Martinsson N. Halko and J. A. Tropp. *Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. 2011.* URL: https://epubs.siam.org/doi/epdf/10.1137/090771806.

[3] Lloyd N. Trefethen. *Approximation Theory and Approximation Practice, Extended Edition. 2019.* URL: https://epubs.siam.org/doi/book/10.1137/1.9781611975949?mobileUi=0.

[4] Daniel Kressner. *Numerical Analysis, Chapters 2 and 3 : Polynomial interpolation, EPFL MATH.*

[5] Christoph Schwab Stefan A. Sauter. *Boundary Element Methods. 2004.* URL: https://link.springer.com/content/pdf/10.1007/978-3-540-68093-2.pdf.

[6] URL: http://www.chebfun.org.

[7] URL: https://en.wikipedia.org/wiki/Kernel_method.

[8] URL: https://en.wikipedia.org/wiki/Mercer%27s_theorem.