SORBONNE UNIVERSITÉ

UE 4IN900 - « PROJET », M1 SPECIALITY SFPN

Project Report
COMPUTATION OF PSEUDOSPECTRUM

Calle Viera Andersson, Zaghouani Slim

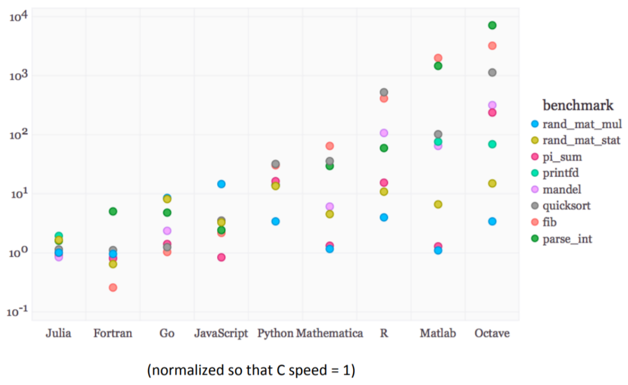
Supervised by
GRAILLAT Stef

Contents

Introduction

Our research subject is the computation of Pseudospectrum. A Pseudospectra is a mathematical tool used to describe the behavior of a linear transformation when its eigenvalue analysis is misleading and when predictions fail to match observations. Specifically, trouble may arise when the associated sets of eigenvectors are ill-conditioned with respect to the norm, here we will focus mainly on the case of the familiar Euclidean or 2-norm. This issue frequently arises in several fields, from atmospheric science to non-Hermitian quantum mechanics, so it is crucial to find a quick way to compute the Pseudospectra of a matrix. During this project we will study the mathematical properties of Pseudospectrums, eigenvalues and Singular Value Decomposition and compare the performances of the sequential and parallel implementations of two algorithms (GRID and prediction-correction).

To implement the algorithms presented through the paper, we will use the **Julia** programming language and more specifically **Julia Pro-1.2.0-2**. Created in 2009, Julia is a compiled language, such as C, and not interpreted for faster runtime performance but is also interactive like Python and offers a pretty much straightforward syntax very much like Python or Matlab. Not only Julia combines the benefits of this languages but it can also call Python, C, and Fortran libraries. One other Julia's most distinctive features is that it supports parallel computing(with or without MPI), which makes it well suited for high performance numerical analysis. Finally it offers a great and simple collection of methods to create complex U.Is.



Performance of different languages (normalized so that C speed = 1).

The code is available at `https://github.com/anders1901/PSFPN`.

1 Pseudo-spectra of a matrix

Definitions :

Before starting with the core of the project let's remind the definition of a matrix norm. In what follows, $\mathbb{K}$ will denote a field.
Let $A, B \in M_n(\mathbb{K})$ and $\alpha \in \mathbb{K}$, the function $\|\cdot\| : M_n(\mathbb{K}) \longrightarrow \mathbb{R}$ is a matrix norm if it satisfies the following properties:

1. $\|A\| \geq 0$ and $\|A\| = 0 \Leftrightarrow A = 0$
2. $\|\alpha A\| = |\alpha|\|A\|$
3. $\|A + B\| \leq \|A\| + \|B\|$
4. $\|AB\| \leq \|A\|\|B\|$

In this project we will be using the 2-norm definided as :

$$\|A\|_2 = \max_{\|x\| \neq 0} \frac{\|Ax\|_2}{\|x\|_2} \tag{1}$$

It exists various equivalent definitions of pseudospectra given and demonstrated in [1] for example. Here we will use the following ones :

Definition 1.1. Let $A \in M_n(\mathbb{C})$ and $\varepsilon > 0$.
The ε-pseudospectrum $\Lambda_\varepsilon(A)$ of A is the set of $z \in \mathbb{C}$ such that :

$$\|(z - A)^{-1}\|_2 > \varepsilon^{-1} \tag{2}$$

Definition 1.2. $\Lambda_\varepsilon(A)$ is the set of $z \in \mathbb{C}$ such that for some $E \in M_n(\mathbb{C})$ with $\|A - E\|_2 \leq \varepsilon$.

$$z \in \Lambda(E) \tag{3}$$

Definition 1.3. $\Lambda_\varepsilon(A)$ is the set of $z \in \mathbb{C}$ such that for some $v \in \mathbb{C}^n$ with $\|v\| = 1$:

$$\|(z - A)v\|_2 < \varepsilon \tag{4}$$

Let E $\in M_n(\mathbb{C})$, to fully grasp the next definition let's remind the reader that we call singular vlaues of E the square roots of the matrix's eigenvalues. We will write σ_{min} the lowest singular value.

Definition 1.4. For $\|\cdot\| = \|\cdot\|_2$, $\Lambda_\varepsilon(A)$ is the set of $z \in \mathbb{C}$ such that :

$$\sigma_{min}(z - A) \leq \varepsilon \tag{5}$$

Theorem 1.1. *For any matrix $A \in M_n(\mathbb{C})$ and for $\|\cdot\|_2$ the four definitions above are equivalent.*

Gershgorin discs :

Definition 1.5. Let $A = (a_{ij}) \in M_n(\mathbb{C})$.
For any $i \in [\![1,\ n]\!]$, we call Gershgorin discs associated to A :

$$D_i = B_f\left(a_{ii}, \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}|\right) = \left\{ z \in \mathbb{C} : |z - a_{ii}| \leq \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}| \right\} \tag{6}$$

Theorem 1.2 (Gerschgorin theorem)**.**
Let $A = (a_{ij}) \in M_n(\mathbb{C})$, every eigenvalue of A lies within at least one of the Gershgorin discs i.e if λ is an eigenvalue of A then $\lambda \in \bigcup_{i=1}^{n} D_i$.

2 GRID-svd algorithm

All the algorithms presented here are to compute the pseudospectra of matrix are based on the formula in (1.4) and. The easiest way, but not the most efficient one, to determine the pseudospectra of A is to use the Basic GRID SVD algorithm which consists of checking if a previously defined set of points on a grid satisfy the condition of (5) and ploting the ones which do.
To determine this grid of points let's use the Theorem 1.2 and extend it to the pseudospectra of A.

Theorem 2.1. Let $A = (a_{ij}) \in M_n(\mathbb{C})$ and $\varepsilon > 0$. For any $z \in \Lambda_\varepsilon(A)$:

$$|z - a_{ii}| \leq \sqrt{n}\varepsilon + \sum_{\substack{j=1 \\ j\neq i}}^{n} |a_{ij}| \tag{7}$$

Proof. Let z $\in \Lambda_\varepsilon(A)$, and $B \in M_n(\mathbb{C})$ such that $\|B - A\|_2 \leq \varepsilon$ as in Definition 1.2.
Let $\|\cdot\|_\infty$ denote the sup norm defined such as $\|A\|_\infty = \max\limits_{0\leq i\leq n} \sum\limits_{j=1}^{n} |a_{ij}|$, using the fact that on a finite-dimensional linear space two norms are equivalent, we have here :

$$\frac{1}{\sqrt{n}}\|A - B\|_2 \leq \|A - B\|_\infty \leq \sqrt{n}\|A - B\|_2 \tag{8}$$

$\implies \|A - B\|_\infty = \max\limits_{0\leq i\leq n} \sum\limits_{j=1}^{n} |a_{ij} - b_{ij}| \leq \sqrt{n}\|A - B\|_2 \leq \sqrt{n}\varepsilon$
$\forall i \geq 1,\ |z - a_{ii}| = |z - b_{ii} + b_{ii} - a_{ii}| \leq |z - b_{ii}| + |b_{ii} - a_{ii}|$
as, $|b_{ii} - a_{ii}| \leq \sum\limits_{j=1}^{n} |b_{ij} - aij| \leq \max\limits_{0\leq i\leq n} \sum\limits_{j=1}^{n} |a_{ij} - b_{ij}| \leq \sqrt{n}\varepsilon$ and, $|z - b_{ii}| \leq \sum\limits_{\substack{j=1 \\ j\neq i}}^{n} |b_{ij}|$ by Definition 1.5.

$\implies |z - a_{ii}| \leq \sum\limits_{\substack{j=1 \\ j\neq i}}^{n} |b_{ij}| + \sqrt{n}\varepsilon$

Using the reverse triangle inequality : $\forall i, j \geq 1,\ |a_{ij} - b_{ij}| \geq \big||a_{ij}| - |b_{ij}|\big|$ we can observe that
$\sqrt{n}\varepsilon \geq \max\limits_{0\leq i\leq n} \sum\limits_{j=1}^{n} |a_{ij} - b_{ij}| \geq \sum\limits_{j=1}^{n} |a_{ij} - b_{ij}| \geq \sum\limits_{j=1}^{n} \big||a_{ij}| - |b_{ij}|\big| \geq \Big|\sum\limits_{j=1}^{n} |a_{ij}| - |b_{ij}|\Big|$
$\implies -\sqrt{n}\varepsilon \leq \sum\limits_{j=1}^{n} \big(|a_{ij}| - |b_{ij}|\big) \leq \sqrt{n}\varepsilon \implies \sum\limits_{j=1}^{n} |b_{ij}| \leq \sum\limits_{j=1}^{n} |a_{ij}| + \sqrt{n}\varepsilon$
$\iff \Big(\sum\limits_{\substack{j=1 \\ j\neq i}}^{n} |b_{ij}|\Big) + |b_{ii}| \leq \Big(\sum\limits_{\substack{j=1 \\ j\neq i}}^{n} |a_{ij}|\Big) + |a_{ii}| + \sqrt{n}\varepsilon \iff \sum\limits_{\substack{j=1 \\ j\neq i}}^{n} |b_{ij}| \leq \sum\limits_{\substack{j=1 \\ j\neq i}}^{n} |a_{ij}| + \sqrt{n}\varepsilon - (|b_{ii}| - |a_{ii}|)$
Note that $|b_{ii}| - |a_{ii}| \leq \big||b_{ii}| - |a_{ii}|\big| \leq |a_{ij} - b_{ij}| \leq \sqrt{n}\varepsilon$
Hence : $\sum\limits_{\substack{j=1 \\ j\neq i}}^{n} |b_{ij}| \leq \sum\limits_{\substack{j=1 \\ j\neq i}}^{n} |a_{ij}| + \sqrt{n}\varepsilon - \sqrt{n}\varepsilon = \sum\limits_{\substack{j=1 \\ j\neq i}}^{n} |a_{ij}|$

$\implies |z - a_{ii}| \leq \sum\limits_{\substack{j=1 \\ j\neq i}}^{n} |a_{ij}| + \sqrt{n}\varepsilon$ □

We can now use this formula to determinate a domain of research for the pseudo-eigenvalues of a matrix and in doing so optimize the time of execution of our algorithm.
One could argue that depending on the localization and density of the pseudospectra, an algorithm based on this ineqality could spend time calculating non relevant values but it is a great first step.

Algorithms:

Gershgorin discs :
For the rest of the section assume that we have $A \in M_n(\mathbb{C})$, to begin with we used the formula in (7) to compute n discs that we stored as a list of centers and rays to be used later on.

Algorithm 1: `cerclesG`

```
Data: $A \in M_n(\mathbb{C})$, $\varepsilon \in \mathbb{R}$
Result: $Disk = ((c_1,\ r_1), \ldots, (c_n,\ r_n))$ a list of centers and rays where
        $\forall i \in [\![1,\ n]\!],\ c_i \in \mathbb{C}, r_i \in \mathbb{R}_+$
Initialization :
$Disk = [\,]$
$n = size(A)$
$k = \sqrt{n}\varepsilon$
for $i = 1$ to $n$ do
    $summ = 0$
    for $j = 1$ to $n$ do
        if $i \neq j$ then
            $summ = summ + |A[i, j]|$
    $A[i] = (A[i,\ i],\ summ + k)$
return $Disk$
```

Once those discs are computed we can calculate the extreme down right corner of the smallest square containing all our discs. Once this point is found we then return and save it's coordinates and also the lenght of the square. This allows us to store less data and makes it easier for the rest of the algorithm. Now that we have limited our search to a much small portion of space we can make a grid of it and use the inequality in (5) to test if it is, indeed, in the pseudospectra. This is the method that was used in 1991 and can be optimized now. Indeed `grid_svd` is not optimal at all because in order to determine wether or not a point of the grid is in the pseudospectra we have to compute a full SVD. Let $m^2 \in \mathbb{N}$ be the size of the grid, we therefore have a complexity of $\mathcal{O}(n^3 m^2)$.

Algorithm 2: `grid_svd`

```
Data: $A \in M_n(\mathbb{C})$, $\varepsilon \in \mathbb{R}$, $nb_{points} \in \mathbb{N}$
Result: $p : plot$
Initialization :
$eig = $ eigvals$(A)$
$circles = $ cerclesG$(A,\ \varepsilon)$
$contour = $ build_contour$(circles)$
$x = $ linspace$(contour[1],\ contour[1] + contour[3],\ nb_{points})$
$y = $ linspace$(contour[2],\ contour[2] + contour[4],\ nb_{points})$
$Z = [\,]$
for $k = 1$ to $nb_{points}$ do
    for $j = 1$ to $nb_{points}$ do
        $Z[j,\ k] = $ min(svd(diag$(x[k] + \mathbf{i}y[j]) - A$))
$p = $ contour$(x,\ y,\ Z,\ levels = \varepsilon)$
$p = $ scatter$(\mathrm{Re}(eigs),\ \mathrm{Im}(eigs))$
return $p$
```

Results:

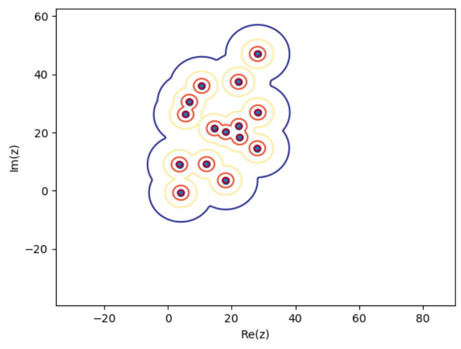


Figure 2: ε-pseudospectre computed with our function for A

We first tested our function for a small random diagonal matrix $A \in \mathbb{C}^{15\times 15}$ and for $\varepsilon \in \{1,\ 2.5,\ 5, 10\}$ and with 200×200 points on the grid. The time of computation is about `1sec` with `1.32 M` allocations, `1.168 GiB` and `21.11% gc time`.

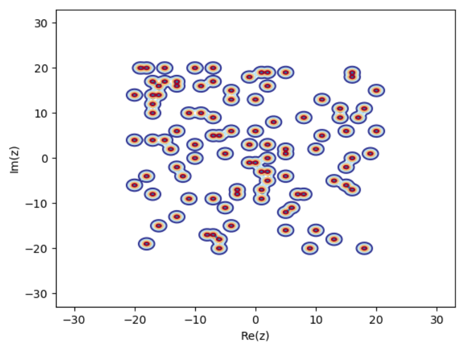


Figure 3: ε-pseudospectre computed with our function for B

Then we tested our function for a bigger random diagonal matrix $B \in \mathbb{C}^{100\times 100}$ and $\varepsilon \in \{0.4,\ 0.7,\ 1, 1.3\}$ and with 200×200 points on the grid. The time of computation is about `23.033569sec` with `1.03 M` allocations, `22.422 GiB` and `15.89% gc time`.

Parallel version :

The GRID-SVD algorithm can be easily parallelised, let's state that we are working with a matrix $A \in \mathbb{C}^{n\times n}$, a grid of $nb_{points} \times nb_{points}$ and nb_{procs} processors.
We can tell each of the $nb_{procs} - 1$ first ones to run exactly $N = \left\lfloor \frac{nb_{points}}{nb_{procs}} \right\rfloor$ lines and the last processor will run the ones left. To put in a nutshell we do not require data movement, no reduction operator is needed and we just want to apply a function to all elements in our collection of points of the grid. Based on this idea we used the predefined function `pmap` of Julia that allows us to do such a thing in the most efficient way possible.
The only small issue we had with parallel map was that in it's easiest way of using it the function shares every single point of the grid to a process in an asynchronously way. Our first results were terrible due to this effect. In order to make it properly work we had to first create two coordinate matrices from our coordinate vectors x and y, for the sake of this explanation let's call them X and Y. But still it wasn't enought, with this new improvement `pmap` shares every single line of X and Y to a new process. We had to explicitly tell the function to divide the arrays in blocks of lines.
To do so we divided the new arrays in sub-arrays as stated in the beginning of this paragraph and we

stored them in $X\_$ and $Y\_$. Finally we used the `pmap` on them and used a simple `map` function for each sub-array.
At the end we will obtain an array Z containing the pseudospectra divided between processes to put it in a correct way we use the function *vcat* that allows to merge the sub-arrays and the key-word "..." to apply it to the entire collection of elements.

```
Algorithm 3: calcul
  Data: $x$, $y$
  min(svd(diag($x + \mathbf{i}y$) $- A$))
```

```
Algorithm 4: repartition
  Data: $x$, $y$
  map(calcul, $x$, $y$)
```

```
Algorithm 5: grid_parallel
  Data: $A \in M_n(\mathbb{C})$, $\varepsilon \in \mathbb{R}$, $nb_{points} \in \mathbb{N}$, $nb_{procs} \in \mathbb{N}$
  Result: $p : plot$
  Initialization :
  $N = \left\lfloor \frac{nb_{points}}{nb_{procs}} \right\rfloor$
  $eig = $ eigvals($A$)
  $circles = $ cerclesG($A,\ \varepsilon$)
  $contour = $ build_contour($circles$)
  $x = $ linspace($contour[1],\ contour[1] + contour[3],\ nb_{points}$)
  $y = $ linspace($contour[2],\ contour[2] + contour[4],\ nb_{points}$)
  $X, Y = $ meshgrid($x,\ y$)
  $X\_ = [\,]$
  $Y\_ = [\,]$
  Initialization of sub-arrays :
  for $i = 1$ to $nb_{procs} - 1$ do
      $X\_[i] = X[(i-1) \times N + 1 : N \times i, :]$
      $Y\_[i] = Y[(i-1) \times N + 1 : N \times i, :]$
  $X\_[nb_{procs}] = X[(nb_{procs} - 1) \times N + 1 : end, :]$
  $Y\_[nb_{procs}] = Y[(nb_{procs} - 1) \times N + 1 : end, :]$
  $Z = $ pmap(($a_1,\ a_2$) $\rightarrow$ repartition($a_1,\ a_2$)$, X\_, Y\_$)
  $p = $ contour($x,\ y,\ $vcat($Z...$)$,\ levels = \varepsilon$)
  $p = $ scatter($\mathrm{Re}(eigs),\ \mathrm{Im}(eigs)$)
  return $p$
```

Comparison :

In order to compare the performances of our algorithms we decided to test them on random diagonale matrices with dimensions varying between 2 and 100 with parameter $\varepsilon = 1$ and $nb_{points} = 200$ points on the grid. For each new matrice we run our programs and measure the different runtimes. The tests bellow are done on a `Dual-Core Intel Core i5`.

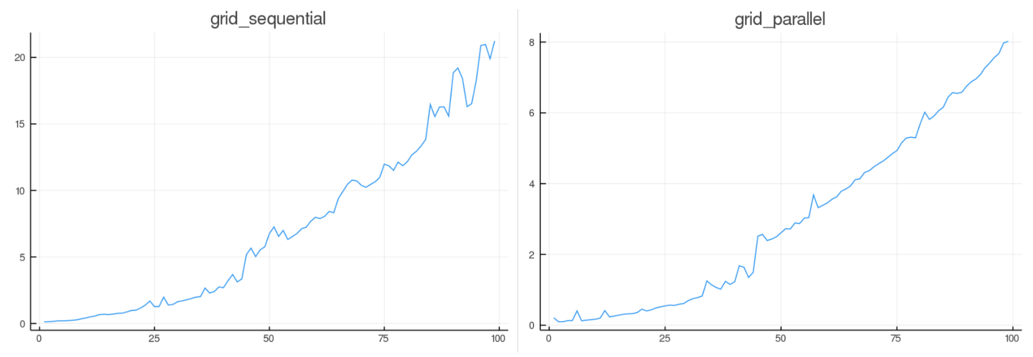


Figure 4: Speed comparison between sequential and parallel GRID algorithm.

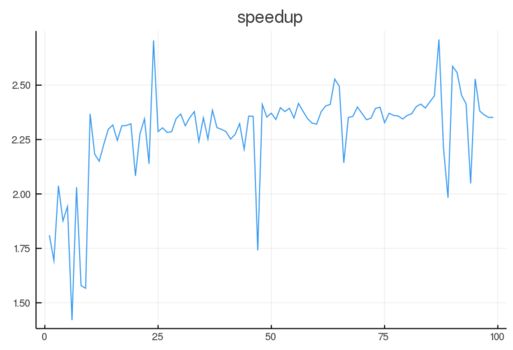


Figure 5: Speedup of our parallel algorithm.

On Figure 13 we ploted the speedup S defined such as $S = \dfrac{T_S}{T_P}$, where T_S is the runtime of the sequential algorithm and T_P is the one of the parallel version to compute the same problem on p processors. Due to the processor used to test the algorithms one could expect a speedup of 2.

We can clearly see that for almost all dimensions the speedup is greater or equal to 2. For dimensions smaller than 15 a reason that could explain a smaller ratio could be the cost of communication between processes. For all the other dimensions some kind of accelerations due to Julia could be the reason that the speedup is greater than 2.

3 Prediction-correction

Another view on the ε-pseudospectrum $\Lambda_\varepsilon(A)$ is that it can also be obtained by tracing directly its boundary curve. By using def. (1.1) it is the set $\{z \in \mathbb{C} : \|(z-A)^{-1}\| = \varepsilon^{-1}\}$ or equivalently by using def. (1.4), $\{z \in \mathbb{C} : \sigma_{min}(z-A) = \varepsilon\}$. Therefore computing the boundary of $\Lambda_\varepsilon(A)$ reduces to determining the curve in the complex plane implicitly defined by g(z)=ε , where g(z)= $\sigma_{min}(zI-A)$,σ_{min} being the smallest singular value of $(zI-A)$. In order to obtain this boundary curve we use a predictor-corrector path following algorithm which works like this :

1. We determine z_1 on $\partial\Lambda_\varepsilon(A)$ (*i.e* it is on the boundary curve)
2. for $k \geq 2$:
   (a) (Prediction) : determine a direction $r_k \in \mathbb{C}$, $|r_k| = 1$ and a steplength τ_k, then compute the point $\widetilde{z}_k = z_{k-1} + \tau_k r_k$.
   (b) (Correction) : determine a direction $d_k \in \mathbb{C}$, $|d_k| = 1$ and θ_k, then compute the corrected point $z_k = \widetilde{z}_k + \theta_k d_k$.

Before formulating the algorithm, let us take a closer look at the function g. the function g(z) is real valued and nonnegative. Moreover, since the zeros of g are the eigenvalues of A, we can deduce that for $x+iy \in \mathcal{C}\backslash\Lambda(A)$, $g(x,y)$ is real analytic in the neighbourhood of (x,y), if $\sigma_{min}((x+iy)I-A)$

is a simple singular value. Then the gradient $\nabla g(x,y) = v^*_{min}u_{min}$. Where u_{min} and v_{min} denote respectively the left and right singular vectors associated with $g(x,y)$

Algorithms:

In step 0 the first point z_1 is determined by the application of Newton's method to univariate nonlinear equations of the type $f(\theta) = 0$ where $f(\theta) = g(\lambda + \theta d) - \varepsilon$. In other words, we look for a solution of $g(z) = \varepsilon$ along a straight line $\{\lambda + \theta d : \theta \in \mathbb{C}\}$. The first iterate θ_1 is then defined by

$$\theta_1 = -\frac{g(\lambda) - \varepsilon}{h'(0)} = -\frac{\sigma_{min} - \varepsilon}{Re(\bar{d}v_{min}u_{min})} \tag{9}$$

which corresponds to the point

$$z_1 = \lambda + \theta_1 d = \lambda - \frac{\sigma_{min} - \varepsilon}{Re(\bar{d}v^*_{min}u_{min})}d \tag{10}$$

Then in step 1 We set a direction $r_k \in \mathbb{C}$ to compute our prediction point $\hat{z_k}$

$$r_k = i\nabla g(\hat{z_{k-1}})/|g(\hat{z_{k-1}})| \tag{11}$$

This allow us to enforce r_k to be orthogonal to the gradient of g at $\hat{z_{k-1}}$. Finally the correction step consists of one single Newton step with respect to the equation

$$g(\hat{z_k} + \theta d_k) - \varepsilon = 0$$

Therefore we can express, by formula (10), the correction z_k in terms onf the smallest singular value of $\hat{z_k}I - A$. Moreover,in the special case $d = \nabla g(\lambda) = v^*_{min}u_{min}$ where d is the direction of the steepest ascent in the point λ equation (10) can be simplified to :

$$z_1 = \lambda - \frac{\sigma_{min} - \varepsilon}{v_{min}u_{min}} \tag{12}$$

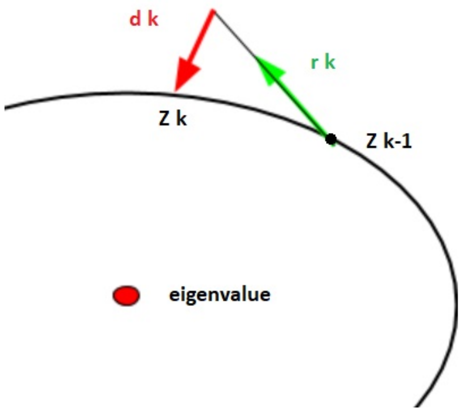


Figure 6: Directions choosed in the prediction step (green) and correction step(red)

Algorithm 6: `Prediction_Correction`

```
Data: $A \in M_n(\mathbb{C})$, $\varepsilon \in \mathbb{R}$, $nb_{points} \in \mathbb{N}$, $tol \in \mathbb{R}$ and $d_0$ the relative accuracy and the search
      direction for the first point
Result: $Pseudospectre : nb_{points}$ size list of coordinates
$eigval$ = eigvals($A$)
for $i = 1$ to $size(eigval, 1)$ do
    $eig = eigval[i]$
    Step 0 :
    $\theta_0 = \varepsilon$
    $z_1^{new} = \lambda_0 + \theta_0 d_0$
    $\sigma_{min}, u_{min}, v_{min}$ = singular_triplet($z_1^{new}, A$)
    while $/ \sigma_{min} - \varepsilon| > tol \times \varepsilon$ do
        $z_1^{old} = z_1^{new}$
        $\sigma_{min}, u_{min}, v_{min}$ = singular_triplet($z_1^{old}, A$)
        $z_1^{new} = \lambda - \frac{\sigma_{min} - \varepsilon}{Re(\bar{d} v_{min}^* u_{min})} d$
        $\sigma_{min}, u_{min}, v_{min}$ = singular_triplet($z_1^{new}, A$)
    $z_1 = z_1^{new}$
    for $k = 1$ to $nb_{points}$ do
        Step 1:
        $r_k = i v_{min}^* u_{min} / |v_{min}^* u_{min}|$
        We chose to keep a fixed steplenght throughout the algorithm
        $z_k^{pred} = z_{k-1} + steplenght \times r_k$
        Step 2:
        $\sigma_{min}, u_{min}, v_{min}$ = singular_triplet($z_k^{pred}, A$)
        $z_k = z_k^{pred} - \frac{\sigma_{min} - \varepsilon}{u_{min}^* v_{min})}$
        push!($Pseudospectre$, $z_k$)
return $Pseudospectre$
```

To compute the singular triplet we use the function svd from the LinearAlgebra julia librairy to perform the singular value decomposition of $(A - zI)$.

Algorithm 7: `Singular_triplet`

```
Data: $A \in M_n(\mathbb{C})$, $z \in \mathbb{C}$,
Result: $\sigma_{min}, u_{min}, v_{min}$
$C$ = diagm(0 => fill($z$, size($A$, 1)))
$U, S, V$ = svd($C - A$)
$\sigma_{min} = S$[size($A$, 1)]
$u_{min} = U$[:, size($A$, 1)]
$v_{min} = V$[:, size($A$, 1)]
return $\sigma_{min}, u_{min}, v_{min}$
```

Results:

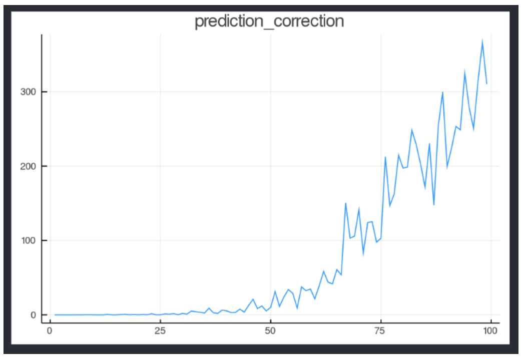


Figure 7: Performance of the prediction correction algorithm

It appears that our implementation of the prediction correction algorithm is very slow when compared to the GRID algorithm, which is very unexpected. The most likely cause is that the algorithm is computing too many points. At every iteration it is computing a new point on the boundary curve of the pseudospectrum without checking if it has already been computed before. This might result in a situation where the algorithm plots a curve going around the pseudospectrum several times. To remedy that, we should consider adding a break statement in the correction step if the current point is almost equal to the first point computed in step 0.

Parallel version :

To write a parallel version of the prediction correction algorithm we used a different approach than grid svd.Let's state that we are still working with a matrix $A \in \mathbb{C}^{n\times n}$ and nb_{procs} processors.Our matrix has a number of eigenvalues $nb_{eigenvalues} \leq n$ this time we are going to tell each of the $nb_{procs} - 1$ first processors to run the algorithm for only $N = \left\lfloor \frac{nb_{eigenvalues}}{nb_{procs}} \right\rfloor$ eigenvalues and the last processor will run the ones left.
To do so we tried using the built in Julia's multi-threading capabilities by writing a base.Threads@threads for loop going through the list of all the eigenvalues of A

Comparison :

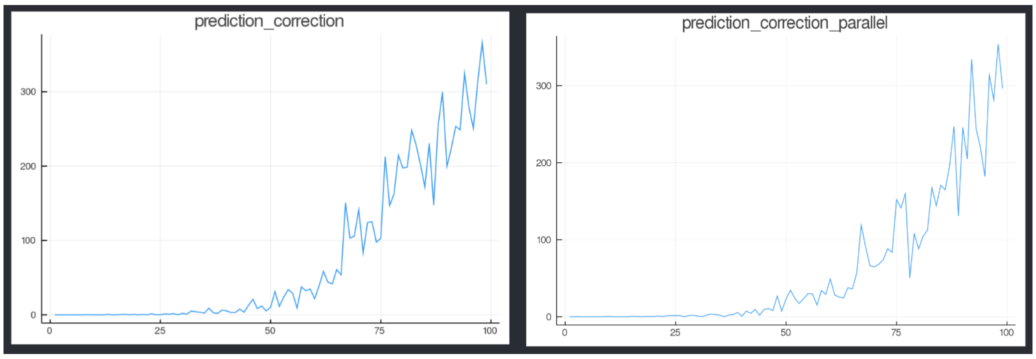


Figure 8: Speed comparison between sequential and parallel Prediction Correction algorithms.

We tested the algorithms on random matrices with dimensions varying from 2 to 100 with parameter $\varepsilon = 1$. This figures show us the differents runtimes for the sequential and parallel version. At first it seems that we can not observe a significant speed up

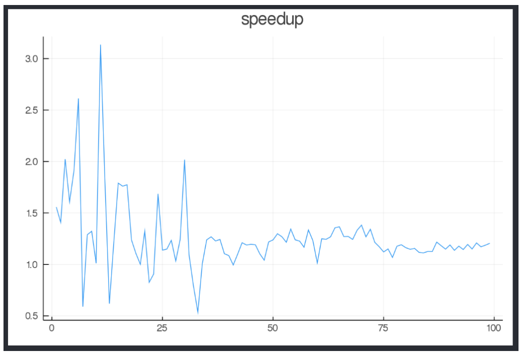


Figure 9: Speed up of the parallel prediction correction algorithm

Considering the fact that the tests above were done on a `Dual-Core Intel Core i5`, we can say that the speed up is way lower than expected at 1.2 approximately, when it should be closer to 2.

4 Componentwise GRID

The main idea for this algorithm is to take into consideration a perturbation for each coefficient of the matrix. Given a weights matrix E we will define the componentwise pseudospectrum of a matrix such as a contour sets of a function.

Algorithms:

As stated before the focus of this approach is to handle a perturbation on each coefficient. We had just to modify a little our code from the Algorithm 2 in order to make it work. The main thing we changed is inside the double fr loops.

Algorithm 8: `grid_par_composante`

```
Data: $A \in M_n(\mathbb{C})$, $\varepsilon \in \mathbb{R}$, $nb_{points} \in \mathbb{N}$
Result: $p : plot$
Initialization :
$eig = \texttt{eigvals}(A)$
$circles = \texttt{interclesG}(A,\ \varepsilon)$
$contour = \texttt{build\_contour}(circles)$
$x = \texttt{linspace}(contour[1],\ contour[1] + contour[3],\ nb_{points})$
$y = \texttt{linspace}(contour[2],\ contour[2] + contour[4],\ nb_{points})$
$E = (1)_{n \times n}$
$Z = [\,]$
for $K = 1$ to $nb_{points}$ do
    for $j = 1$ to $nb_{points}$ do
        $M = |(\texttt{diag}(x[k] + \mathbf{i}y[j]) - A)^{-1}|$
        $N = ME$
        $Z[j,\ k] = \texttt{max}(\texttt{eigvals}(N))$
$p = \texttt{contour}(x,\ y,\ Z,\ levels = \varepsilon)$
$p = \texttt{scatter}(\mathrm{Re}(eigs),\ \mathrm{Im}(eigs))$
return $p$
```

Results:

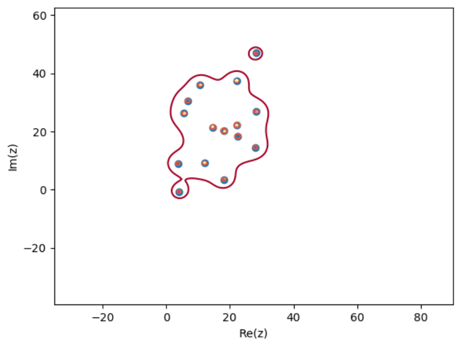


Figure 10: ε-pseudospectre computed with our function for B

For our first matrix A the time of computation is about `3.552562sec` with `1.23 M` allocations, `2.030 GiB` and `16.45% gc time`.

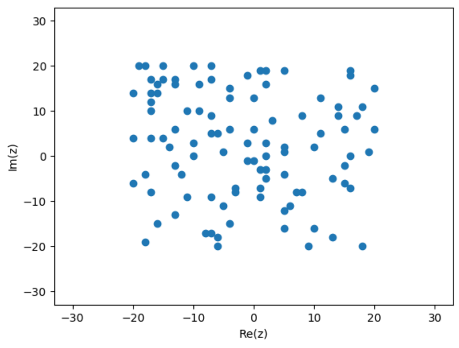


Figure 11: ε-pseudospectre computed with our function for B

The second matrix B with the same parameters takes a lot of time but is at the same time more precise. The time of computation is about `152.277965sec` with `1.39 M` allocations, `28.564 GiB` and `3.06% gc time`.

Parallel version :

Just as the grid-svd algorithm, the componentwise grid algorithm is also really easily parallelised and we had just to adapt the Algorithm 3.

Algorithm 9: `calcul2`

```
Data: $x$, $y$
$M = |(\texttt{diag}(x[k] + \mathbf{i}y[j]) - A)^{-1}|$
$N = ME$
$\texttt{max}(\texttt{eigvals}(N))$
```

Algorithm 10: `repartition2`

```
Data: $x$, $y$
$\texttt{map}(\texttt{calcul2}, x, y)$
```

Algorithm 11: `grid_parallel`

```
Data: $A \in M_n(\mathbb{C})$, $\varepsilon \in \mathbb{R}$, $nb_{points} \in \mathbb{N}$, $nb_{procs} \in \mathbb{N}$
Result: $p : plot$
Initialization :
Initialization :
$eig = \texttt{eigvals}(A)$
$circles = \texttt{cerclesG}(A, \ \varepsilon)$
$contour = \texttt{build\_contour}(circles)$
$x = \texttt{linspace}(contour[1], \ contour[1] + contour[3], \ nb_{points})$
$y = \texttt{linspace}(contour[2], \ contour[2] + contour[4], \ nb_{points})$
$E = (1)_{n \times n}$
$X, Y = \texttt{meshgrid}(x, \ y)$
$X\_ = [\,]$
$Y\_ = [\,]$
Initialization of sub-arrays :
for $i = 1$ to $nb_{procs} - 1$ do
    $X\_[i] = X[(i-1) \times N + 1 : N \times i, :]$
    $Y\_[i] = Y[(i-1) \times N + 1 : N \times i, :]$
$X\_[nb_{procs}] = X[(nb_{procs} - 1) \times N + 1 : end, :]$
$Y\_[nb_{procs}] = Y[(nb_{procs} - 1) \times N + 1 : end, :]$
$Z = \texttt{pmap}((a_1, \ a_2) \rightarrow \texttt{repartition2}(a_1, \ a_2), X\_, \ Y\_)$
$p = \texttt{contour}(x, \ y, \ \texttt{vcat}(Z...), \ levels = \varepsilon)$
$p = \texttt{scatter}(\mathrm{Re}(eigs), \ \mathrm{Im}(eigs))$
return $p$
```

Comparison :

We did the same tests as for the other two algorithms with the same initial conditions.

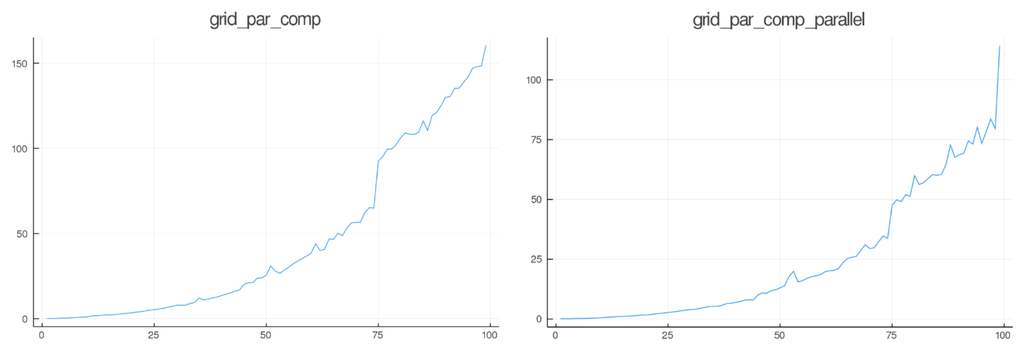


Figure 12: Speed comparison between sequential and parallel componentwise GRID algorithm.

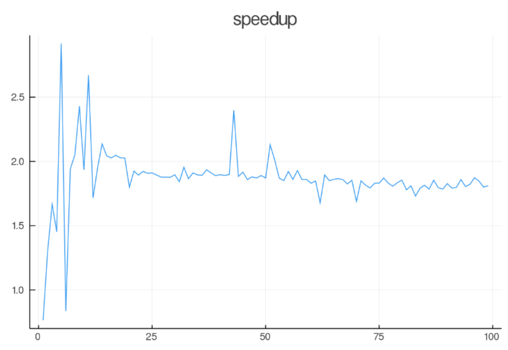


Figure 13: Speedup of our parallel algorithm.

First we can see that this algorithm takes way more time than the simplier grid to run and even for the parallel algorithm the time taken to compute the pseudospectra is more than 100 seconds. There is also a clear increase in computation time at around $n = 75$ and the reason is not clear. This time again we can observe that for all dimensions the speedup is about 2. For dimensions smaller than 25 the ration is even closer to $2,5$ some kind of accelerations due to Julia could be the reason again. Even though we can see a great improvement in speed we can also see that it looks like it is decreasing when the dimension of the matrix gets bigger.

Conclusion

In this project we described two relatively efficient ways to compute the pseudospectra of matrices. However, in many circonstances it is not needed to know the entire pseudospectra, but only certain scalar quantities derived from them, such as the $\varepsilon-$ pseudospectral abcissa and $\varepsilon-$ pseudospectral radius. To go further we could find ways to improve our algorithms to compute those quantities faster or implement the Burke, Lewis, Overton and Mendy 'criss-cross' algorithms as described in [2]

References

[1] Brühl, M. A curve tracing algorithm for computing the pseudospectrum. Bit Numer Math 36, 441–454 (1996).

[2] Lloyd N. Trefethen and Mark Embree. Spectra and pseudospectra : The Behavior of Nonnormal Matrices and Operators. Princeton University Press, Princeton, NJ, 2005

[3] A.N Malyshev and M Sadkane. 2004. Componentwise pseudospectrum of a matrix. Linear Algebra Its Appl. 378, (February 2004), 283–288.

[4] Pseudospectra Gateway. Retrieved March 16, 2020 from https://www.cs.ox.ac.uk/pseudospectra/intro.html