Bureau d'étude Ma313 : Variations autour de la reconnaissance des chiffres manuscrits

DUCHANOY Colin – FOURNIER Nicolas – LANCERY Hugo – SOUCOURRE Vincent

15 Décembre 2021

M. COUFFIGNAL – M. EL MAHBOUBY

Table des matières

Introduction

Dans tout ce BE nous utiliserons l'exemple vu en TP de la reconnaissance des chiffres manuscrits à partir de la base de données MNIST. Nous essaierons d'améliorer quelque peu le taux de reconnaissance.

Dans la partie 1, nous verrons une méthode pour résoudre ce problème avec la factorisation de Cholesky en modifiant légèrement le problème et en le faisant dépendre d'un paramètre. En optimisant ce paramètre nous pourrons alors augmenter quelque peu le taux de réussite dans le cas de la reconnaissance d'un chiffre.

Dans la partie 2, nous verrons une approche « géométrique » sur la reconnaissance des chiffres manuscrits en utilisant l'analyse procustéenne.

Enfin dans la troisième partie nous couplerons les deux approches précédentes afin d'augmenter le taux de réussite.

I. Inverser le non-inversible

1.

Supposons donné un ensemble fini E partitionné en deux sous-ensembles disjoints de points de $\mathbb{R}^n$:

$$E_1 = \{u_i \in \mathbb{R}^n | \ i \in [1,p]\} \ et \ E_2 = \{v_j \in \mathbb{R}^n | \ j \in [1,q]\}$$

Ou p et q sont des entiers non nuls. Ces données forment ce qu'on appelle les données d'entrainement et on cherche f: $\mathbb{R}^n \to \mathbb{R}^n$ de la forme :

$$f(x) = \omega^T x + b$$

Avec $\omega \in \mathbb{R}^n$et $b \in \mathbb{R}^n$ telle que :

$$f(u_i) = 1 \quad \forall\, i \in [1,p] \quad et \quad f(v_j) = -1 \quad \forall\, j \in [1,q]$$

Nous allons rechercher f qui minimise la quantité suivante :

$$\sum_{i=1}^{p} (f(u_i) - 1)^2 + \sum_{j=1}^{q} \left(f(v_j) + 1\right)^2$$

On cherche donc à réécrire ce problème sous forme matricielle.

$$min \sum_{i=1}^{p} (f(u_i) - 1)^2 + \sum_{j=1}^{q} \left(f(v_j) + 1\right)^2$$

$$\Leftrightarrow \ min \left\| \begin{pmatrix} f(u_1) - 1 \\ \vdots \\ f(u_p) - 1 \\ f(v_1) + 1 \\ \vdots \\ f(v_q) + 1 \end{pmatrix} \right\|_2^2 = min \left\| \begin{pmatrix} \omega^T u_1 + b - 1 \\ \vdots \\ \omega^T u_p + b - 1 \\ \omega^T v_1 + b + 1 \\ \vdots \\ \omega^T v_q + b + 1 \end{pmatrix} \right\|_2^2$$

$$\Leftrightarrow \min \left\| \begin{pmatrix} {u_1}^T & 1 \\ \vdots & \vdots \\ \vdots & \vdots \\ {u_p}^T & 1 \\ {v_1}^T & 1 \\ \vdots & \vdots \\ \vdots & \vdots \\ {v_q}^T & 1 \end{pmatrix} \begin{pmatrix} \omega \\ b \end{pmatrix} - \begin{pmatrix} -1 \\ \vdots \\ \vdots \\ -1 \\ 1 \\ \vdots \\ \vdots \\ 1 \end{pmatrix} \right\|_2^2 = \min_{x \in \mathbb{R}^{n+1}} \|Ax - y\|_2^2$$

Avec :

$$A = \begin{pmatrix} {u_1}^T & 1 \\ \vdots & \vdots \\ \vdots & \vdots \\ {u_p}^T & 1 \\ {v_1}^T & 1 \\ \vdots & \vdots \\ \vdots & \vdots \\ {v_q}^T & 1 \end{pmatrix}, x = \begin{pmatrix} \omega \\ b \end{pmatrix} \text{ et } y = \begin{pmatrix} -1 \\ \vdots \\ \vdots \\ -1 \\ 1 \\ \vdots \\ \vdots \\ 1 \end{pmatrix} = \begin{pmatrix} -1_p \\ 1_q \end{pmatrix}$$

2.

On note $\phi(x) = \|Ax - y\|_2^2$

On veut justifier que les points critiques de ϕ vérifient l'équation normale $A^T Ax = A^T y$

On cherche donc $\phi'(x) = 0$

$$\Leftrightarrow 2A^T Ax - 2A^T y = 0$$
$$\Leftrightarrow A^T Ax = A^T y$$

3.

On calcule les valeurs propres approchées de $A^T A$ en utilisant l'algorithme d'itération QR. On utilise pour cela le cas de l'apprentissage de la reconnaissance du chiffre 0. On construit tout d'abord les matrices A et y puis pour calculer les valeurs propres de $A^T A$ on réalise une décomposition QR de cette matrice en utilisant la fonction python *np.linalg.qr($A^T A$)*. Cette décomposition renvoie une matrice B comportant sur sa diagonale les valeurs propres de $A^T A$. Il nous reste juste à récupérer sous forme d'un vecteur colonne ces valeurs en utilisant la fonction python *np.diag(B)*.

On remarque que la matrice $A^T A$ possède des valeurs propres nulles. Par définition elle est donc non inversible.

On cherche maintenant une approximation du rang de la matrice A. Les matrices A et $A^T A$ ont le même noyau, d'après le théorème du rang elles ont donc le même rang.

Il reste donc à calculer le rang de $A^T A$. Pour cela il suffit de compter le nombre de valeurs propres différentes et non nulles. Lorsque nous regardons le vecteur contenant les valeurs propres de $A^T A$ nous constatons que certaines des valeurs sont très faibles (10^{-5}). On les remplace par des zéros afin qu'elles ne soient pas comptées comme des valeurs propres. En suivant cette méthode on trouve un rang de 713.

4.

Avec la méthode SVD, on obtient les valeurs propres de $A^T A$ (*np.linalg.svd($A^T A$)* sur python). Le nombre de valeurs propres correspond ainsi approximativement au range de la matrice A. Ce rang est de 715. On constate qu'il est supérieur au rang obtenu avec la méthode QR. SVD converge moins rapidement vers les valeurs propres que QR pour un epsilon donné.

5.

A partir de maintenant, nous allons suivre les étapes du TP4 tout en changeant la manière de procéder. Dans le TP4, nous avions utilisé la technique du pseudo-inverse. Nous allons cette fois-ci utiliser la décomposition de Cholesky en rendant la matrice $A^T A$ inversible. Pour cela, on pose la matrice suivante :

$$A_\epsilon = A^T A + \epsilon I_{785}$$

Où ϵ est un réel strictement positif et I_{785} est la matrice identité de taille 785 x 785. La matrice A_ϵ est de même taille.

(a)

On commence tout d'abord par effectuer une analyse de cette matrice. En premier lieu, on calcule les valeurs propres approchées de la matrice A_ϵ en fonction de ϵ. On utilise la commande *np.linalg.eigvals(A_ϵ)* sur python pour obtenir ces valeurs propres et on constate qu'il s'agit d'un vecteur colonne (785,) correspondant aux valeurs de $A^T A$ obtenues précédemment multipliées par ϵ. De plus, là où certaines des valeurs propres de $A^T A$ étaient nulles, celles de A_ϵ sont maintenant égales à ϵ.

Nous pouvons alors montrer que cette matrice A_ϵ est symétrique définie positive. On observe par le calcul python que $A_\epsilon = {A_\epsilon}^T$. Elle est donc symétrique. De plus, nous avons pu voir qu'aucune de ses valeurs propres n'étaient nulles. Elle est donc également définie positive et donc inversible.

(b)

On va maintenant écrire la fonction *resChol(nombredetection, ϵ)* qui dépend d'un entier *nombredetection* $\in [\![0, 9]\!]$ correspondant au nombre à détecter et *epsilon* au ϵ de la question précédente. Cette fonction va nous permettre d'obtenir de nombreuses informations importantes pour analyser la détection des chiffres manuscrits.

i.

L'un des premiers éléments importants que nous donne cette fonction est la solution de l'équation exprimée de la manière suivante :

$$sol_\epsilon = {A_\epsilon}^{-1} A^T y$$

Pour obtenir cette solution, nous commençons par effectuer la décomposition de Cholesky (en utilisant la fonction *factoCholesky* sur python que nous avons déjà codé lors d'un précédent TP) de la matrice A_ϵ. On obtient alors une matrice triangulaire inférieure L de taille 785 x785.

A partir de cette matrice, nous la résolvons (grâce à la fonction *resolvtrianginf*) en posant la matrice $B = A^T y$, matrice nécessaire pour la résolution de la matrice triangulaire inférieure.

Pour finir, nous résolvons (grâce à la fonction *resolvtriangsup*) cette nouvelle matrice avec la transposée de la matrice L afin d'obtenir sol_ϵ, vecteur solution de taille (785,) pour chaque chiffre manuscrit. Ces vecteurs seront utiles pour la détermination du taux de reconnaissance globale sur l'ensemble des valeurs « test ».

ii.

Il est nécessaire de vérifier l'efficacité de cette fonction. Pour cela, en plus de ces vecteurs solutions, la fonction nous permet également de récupérer le taux de reconnaissance du chiffre *nombredetection* défini par :

$$T_r := \frac{N_{vp} + N_{vn}}{10000}$$

Ainsi que la matrice confusion M_{conf} associé sur les 10000 valeurs test :

$$M_{conf} := \begin{pmatrix} N_{vp} & N_{fn} \\ N_{fp} & N_{vn} \end{pmatrix}$$

Où N_{vp} le nombre de vrais positifs, N_{fn} le nombre de faux négatifs, N_{fp} le nombre de faux positifs et N_{vn} le nombre de vrais négatifs. Les nombres de vrais positifs et négatifs correspondent au nombre de fois où le programme a eu juste sur la détection du chiffre et les nombres faux, le nombre de fois où il s'est trompé sur la prédiction.

Si on prend l'exemple avec le chiffre 0 et un inconnu x à détecter :

- N_{vp} : notre programme pense qu'il s'agit d'un 0 et x est bien un 0.
- N_{fp} : notre programme pense qu'il s'agit d'un 0 et x n'est pas un 0.
- N_{fn} : notre programme pense qu'il ne s'agit pas d'un 0 et x est bien un 0.
- N_{vn} : notre programme pense qu'il ne s'agit pas d'un 0 et x n'est pas un 0.

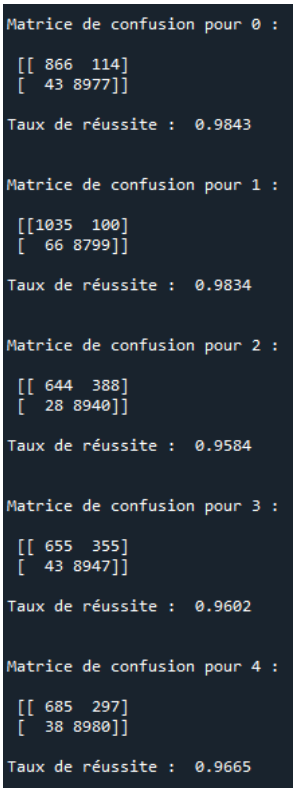
La matrice de confusion compare les données réelles pour une variable cible à celles prédites par un modèle. Les prédictions justes et fausses sont révélées et réparties par classe, ce qui permet de les comparer avec des valeurs définies. Dans notre code, la fonction va venir comparer le vecteur solution obtenue précédemment avec les données et valeurs test.

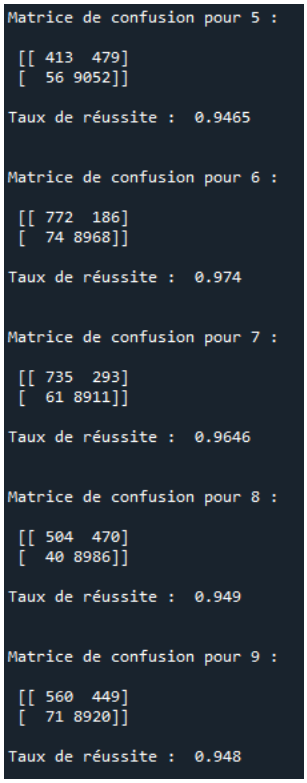
La matrice de confusion est en quelque sorte un résumé des résultats de prédiction de notre problème. Plus elle s'approche d'une matrice diagonale, plus le système est considéré comme performant et efficace.

(c)

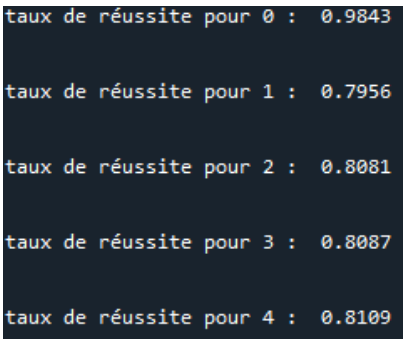
On calcule maintenant le taux de réussite ainsi que la matrice de confusion associée pour chaque chiffre *nombredetection* $\in [0,9]$pour $\epsilon = 1$
Pour cela il suffit de récupérer *txreussite* et *Mc* (défini précédemment) en sortie de notre fonction *resChol(0, ϵ)* qui correspondent respectivement aux taux de réussite ainsi qu'à la matrice de confusion.

Nous n'implémentons pas un affichage de ces valeurs directement dans la fonction *resChol(nombredetection, ϵ).* Afin de ne pas surcharger la console en texte lors des étapes suivantes où la fonction *resChol(nombredetection, ϵ)* sera utilisé plusieurs fois à la suite.





On observe des taux de réussite assez élevés et des matrices de confusion dont les coefficients diagonaux sont les plus élevés, ce qui signifie que notre programme est assez efficace. En comparant avec les taux de réussite obtenus lors du TP4 avec la méthode du pseudo-inverse suivants :

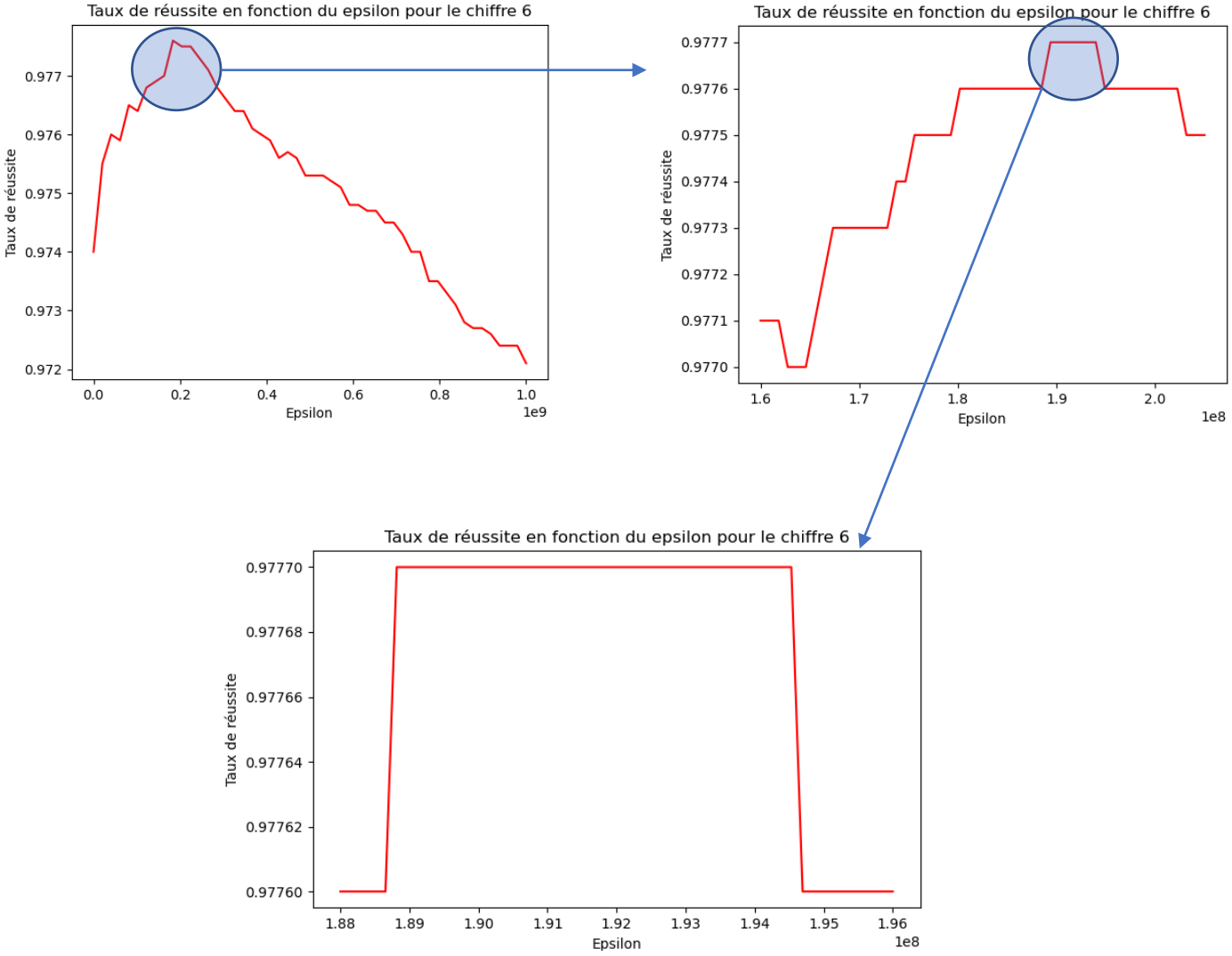


On constate que les taux avec la nouvelle méthode sont bien plus élevés. Ainsi, la nouvelle méthode réduit grandement le taux d'erreur de détection du chiffre correct.

(d)

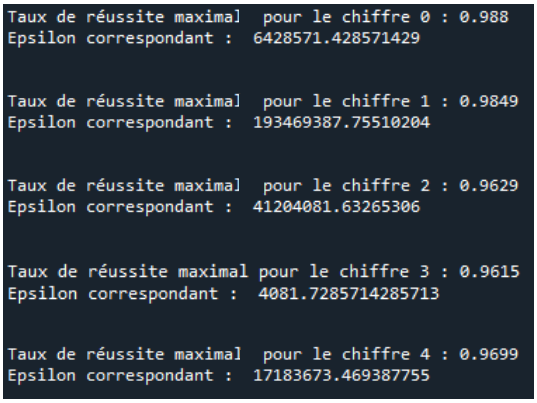
On va à présent faire varier ϵ sur l'intervalle $[10^{-10}, 10^{9}]$ et $\epsilon \neq 0$, puis tracer le taux de réussite en fonction de ϵ en utilisant la fonction *resChol(0, ϵ)*. Pour cela on découpe sous python cet intervalle en 50 parties (*np.linespace( )*) et on affiche le graphique.

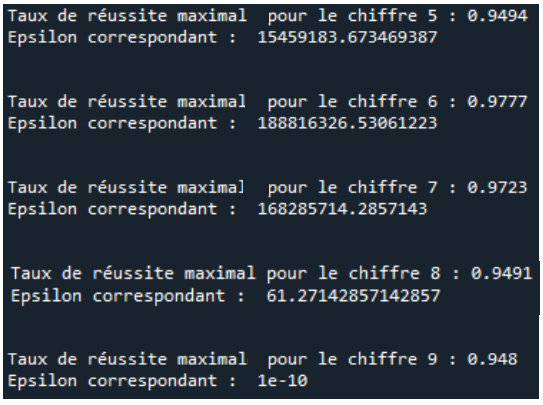
On détermine empiriquement un nouvel intervalle de ϵ qu'on redécoupe en 50 parties afin d'augmenter la précision sur le ϵ. On procède de la façon suivante. Une fois que le graphique apparait, on sélectionne le pic de la courbe correspondant au ϵ qui maximise le taux de réussite et on zoom sur ce pic. On trouve ainsi un nouvel intervalle en abscisse pour ϵ. On relance le programme pour ce nouvel intervalle et on vérifie qu'il est cohérent.



(e)

Il convient maintenant de trouver le ϵ_i qui maximise la détection pour chacun des chiffres $i \in [0,9]$. Pour cela on reprend le code de la partie précédente et de manière empirique on trouve pour chaque chiffre i l'intervalle de ϵ sur lequel se trouve le meilleur taux de réussite. Pour certains chiffres on réalise cette opération de zoom plusieurs fois afin d'affiner le résultat. On obtient alors les taux maximals pour chaque chiffre ainsi que la valeur de l'epsilon correspondant :





On observe que pour tous les chiffres, les taux de réussite sont au-dessus de 94%, ce qui signifie que notre programme reconnait dans une grande partie des cas les chiffres. La majorité de ces taux de réussite sont obtenus avec des epsilons très élevés, de l'ordre de $10^6 - 10^7$ sauf pour les chiffres 3, 8 et 9, pour lesquels l'ordre de grandeur est de 10^3 voire moins.

On forme ainsi 10 vecteurs colonnes $\epsilon_i$contenant les valeurs des ϵ précis pour chaque chiffre i (à l'aide de la fonction python *np.linespace( )*). On regroupe ensuite ces 10 vecteurs dans une même matrice.

On applique maintenant la fonction *resChol(i, ϵ)* à chacun des éléments de chaque ϵ_i. On a donc pour chaque ϵ_i une liste contenant le taux de réussites pour chacun des ϵ. Une fois tous les éléments de ϵ_i parcouru le programme extrait la valeur maximale du taux de réussite puis la valeur de ϵ correspondant à ce taux de réussite. Ensuite on utilise à nouveau la fonction resChol(i, ϵ) pour extraire le vecteur colonne solution « soleps » contenant … Enfin on répète ce programme pour chacun des chiffres $i \in [0,9]$.

6.

A partir des étapes que nous venons de détailler nous sommes maintenant capables de rédiger un programme de détection des chiffres. De la même manière que précédemment on regroupe ensuite les 10 vecteurs colonnes solution « soleps » dans une même matrice de taille (785,10). On crée tout ensuite les fonctions de détection globale *fglobale(x,matrix_soleps)* et *reponse(x,matrix_soleps)* …

On implémente ensuite un exemple test. En faisant varier le i on sélectionne une nouvelle ligne de la base de données mnist_test.csv et donc un nouveau chiffre manuscrit. On teste ensuite ce chiffre à l'aide des fonctions précédentes et on affiche l'image correspondante. Cette étape permet de s'assurer que les fonctions définies précédemment fonctionnent et sont cohérentes avec le taux de réussite.

Une fois cette vérification effectuée on peut tester pour chacune des données test, soit sur les 10 000 données, afin d'obtenir un taux de reconnaissance globale.

Le programme donne un taux de reconnaissance globale sur les 10 000 valeurs de 0.8652 soit 86,52 % qui est très légèrement plus élevé en comparaison à 86,03 % la valeur trouvée en TP en utilisant le pseudo-inverse. Il est donc possible d'améliorer le taux de reconnaissance en utilisant une méthode de décomposition différente. Cependant dans le cas présent la différence est presque négligeable (0,49 %).

Pour cela nous allons tester une nouvelle méthode afin de voir s'il n'est pas possible d'encore améliorer le taux de reconnaissance.

7.

Voir l'interface tkinter sur python

8.

(a)

On cherche maintenant à montrer que remplacer A^TA par $A^TA + \epsilon I_n$ $\phi(x) = \|Ax - y\|_2^2$ revient à rechercher les points critiques de $\phi(x) = \|Ax - y\|_2^2 + \epsilon\|x\|_2^2$.

On sait que les points critiques de ϕ vérifient l'équation : $A^TAx = A^Tb$

On remplace A^TA par $A^TA + \epsilon I_n$ et on obtient : $(A^TA + \epsilon I_n)x = A^Ty$

$$(A^TA + \epsilon I_n)x = A^Ty \Leftrightarrow (A^TA + \epsilon I_n)x - A^Ty = 0$$
$$\Leftrightarrow A^TAx + \epsilon I_n x - A^Ty = 0 \Leftrightarrow 2A^TAx + 2\epsilon I_n x - 2A^Ty = 0$$

$$\Leftrightarrow \|Ax\|_2^2 + \|\epsilon I_n x\|_2^2 - \|y\|_2^2 \Leftrightarrow \|Ax - y\|_2^2 + \epsilon\|x\|_2^2$$

Donc on trouve bien que remplacer A^TA par $A^TA + \epsilon I_n$ revient à rechercher les points critiques de $\phi(x) = \|Ax - y\|_2^2 + \epsilon\|x\|_2^2$

(b)

Minimiser $\|Ax - y\|_2^2 + \epsilon\|x\|_2^2$ revient à avoir $\phi(x) = 0$
Ce qui donne :

$$2A^TAx + 2\epsilon I_n x - 2A^Ty = 0$$
$$\Leftrightarrow A^TAx + \epsilon x - A^Ty = 0$$
$$\Leftrightarrow A^TAx + \epsilon x - A^Ty = 0$$
$$\Leftrightarrow A^TAx + \epsilon x = A^Ty$$

ϵ Influe donc *sur x*

II. L'approche procustéenne

1.

On crée une matrice *u* contenant toutes les lignes liées aux images du chiffre 0. Ensuite on parcourt la nouvelle matrice *u* par colonne et on effectue la moyenne de chaque colonne grâce à la fonction *np.mean*. On dispose toutes les valeurs obtenus grâce à notre boucle dans une liste vide *Um*. On obtient une liste de 784 éléments qu'on va *np.reshape* sous forme d'une matrice 28x28 afin de pouvoir l'afficher sous forme d'image.

Ainsi grâce à la fonction *cv2.imshow* nous pouvons afficher l'image du 0 moyen.

```
cv2.imshow('chiffre', np.uint8(image))  # Affichage de l'image avec cv2
cv2.waitKey(0)
cv2.destroyAllWindows()
cv2.waitKey(1)
```

2.

On va maintenant écrire une fonction *chiffremoy(train_data)*. Au lieu d'appliquer notre fonction pour une seule valeur (0 précédemment) on réalise une boucle qui va faire varier le *nombredetection* de 0 à 9 afin d'obtenir pour chaque nombre le vecteur du chiffre moyen.

On récupère premièrement la colonne 0 du fichier *scv* importé pour chercher avec *np.where* l'indice de la ligne contenant le chiffre i.

```
valeur = train_data[:, 0]
indiceu = np.where(valeur == nbdetection)
u = train_data[:, 1:][indiceu]
```

On ajoute alors toutes ces lignes dans u qui nous sert de stockage intermédiaire pour ensuite importer la moyenne de toutes ces lignes dans Umtot.

```
for j in range(c):
            Umtot.append(np.mean(u[:, j]))
    Umtot = np.array(Umtot)
    Umtot = Umtot.reshape((10, c))
```

On affiche alors les chiffres moyens présents dans la matrice *Umtot à* l'aide de *Tkinter* et de *PIL*.

Pour cela on initialise Tkinter et on créer des « box » qui nous servirons à afficher nos images.

```
plt.ioff()
        fen = tk.Tk()
        fen.title('Affichage des chiffres moyens')
        Framem = tk.Frame(fen, borderwidth=2, relief="groove")
        Framem.pack(side="left", padx=5, pady=5)
        Frame0 = tk.Frame(Framem, borderwidth=2, relief="groove")
        Frame0.pack(side="left", padx=5, pady=5)

                                ( . . . )

        Frame9 = tk.Frame(Framem, borderwidth=2, relief="groove")
        Frame9.pack(side="left", padx=5, pady=5)
```

Puis on utilise une boucle qui récupère, redimensionne et renvoie une image avec PIL des lignes de la matrice *Umtot*.

```
for i in range(10):
            nligne_default = i
            ligne_chiffre = Umtot[i]
            ligne_chiffre = np.array(ligne_chiffre).reshape((28, 28))
            locals()["Image_PIL{}".format(i)] =
ImageTk.PhotoImage(image=Image.fromarray(ligne_chiffre))
```

Pour terminer, il suffit de faire une boucle pour créer 10 labels chacun destiné à accueillir une image et à être placé dans une « box ».

```
for j in range(10):
            locals()["label{}".format(j)] = tk.Label(master=locals()[
                "Frame{}".format(j)],
image=locals()["Image_PIL{}".format(j)])
            locals()["label{}".format(j)].pack()
```

2.BONUS

Essayons maintenant d'augmenter la précision de cette moyenne en augmentant la data base. Prenons en exemple le 6 et le 9. Avec une double rotation de 90° on peut assimiler les données du fichier *train_data.scv* du 6 pour le 9 et vice versa. Etudions alors l'impact sur le taux d'erreur final.

```
i,j=9,9
ligne_chiffre = Umtot[i]
ligne_chiffre=np.array(ligne_chiffre).reshape((28,28))
ligne_chiffre=np.rot90(np.rot90(ligne_chiffre))
ligne_chiffre2=ligne_chiffre.reshape((1,784))
ligne_chiffre2=(Umtot[6]+ligne_chiffre2)/2
```

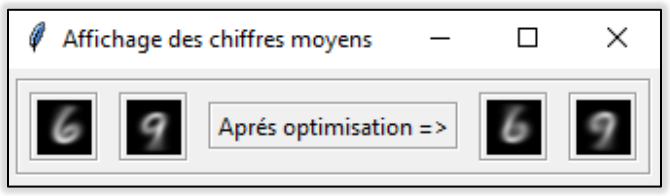
On récupère dans la variable *ligne_chiffre* le vecteur *Umtot* contenant l'image du chiffre i moyen que l'on souhaite étudier (ici 9) puis qu'on *reshape* sous forme d'une matrice 28x28.

On effectue grâce à la commande *np.rot90* une rotation de la matrice *ligne_chiffre* puis on injecte cette nouvelle matrice dans une matrice ligne_chiffre2 de taille 1x784 qui sera l'image de notre 9 en rotation (assimilable à un 6).

On effectue une moyenne avec le *Umtot[6]* obtenu grâce à *chiffre_moy* afin d'obtenir un nouveau vecteur permettant d'affichant la nouvelle image du 6 moyen.

On *reshape* sous une matrice 28x28 afin de pouvoir l'obtenir sous forme d'image.

On notera que l'on peut pousser cette idée avec le 1 qui peut être compris dans le 9,6 et 7.



Pour l'exécution de la fonction *taux_reconnaissance* pour 1000 lignes on obtient :

```
100%|██████████| 1000/1000 [06:49<00:00,  2.44it/s]

le taux d'erreur global pour le fichier test est de 20.9%

On a 209 mauvaises détections pour 1000 valeurs
```

Cependant après l'exécution de la fonction *taux_reconnaissance* pour 1000 lignes avec « l'optimisation » on obtient :

```
100%|██████████| 1000/1000 [06:43<00:00,  2.48it/s]

le taux d'erreur global pour le fichier test est de 21.7%

On a 217 mauvaises détections pour 1000 valeurs
```

21.7% > 20.9% Pas très concluant !

3.

On réalise la fonction *Procuste(A, B)* prenant 2 matrices de même taille. Ainsi *np.shape(A)=np.shape(B)*

On cherche à obtenir les matrices $A_G = A - a_G u$ et $B_G = B - b_G u$. Avec $a_G = \frac{1}{n}\sum_{j=1}^{n} a_j$ et $b_G = \frac{1}{n}\sum_{j=1}^{n} b_j$ Avec a_j et b_j les vecteurs lignes de A et B.

```
m, n = np.shape(B)
    aG = np.zeros((m, 1))
    for i in range(0, n):
        aG = aG+(1./n)*A[:, i][np.newaxis].T

    bG = np.zeros((m, 1))
    for i in range(n):
        bG = bG+(1./n)*B[:, i][np.newaxis].T
```

Grâce à une boucle parcourant les vecteurs lignes de A et de B on obtient ainsi A_G et B_G.
On introduit une nouvelle variable P=$A_G {B_G}^T$ et on effectue la fonction *np.linalg.svd(P)* afin d'obtenir $U_G \Sigma_G V_G^T$ = *np.linalg.svd(P).*

```
P = np.dot(Ag, Bg.T)
Ug, Sg, Vg = np.linalg.svd(P)
```

On peut donc obtenir la valeur $X=V_G U_G^T$ en faisant *np.dot* (V_G, U_G^T). On obtient aussi $\lambda = \frac{trace(\Sigma_G)}{\|A_G\|_F^2}$

```
l = np.trace(np.diag(Sg))/np.linalg.norm(Ag, 'fro')**2
```

Grâce à cela on peut obtenir $t = b_G$ - $\lambda X a_G$ et on peut obtenir l'erreur de transformation $\|B - (\lambda XA + tu)\|_F^2$ avec la commande suivante :

```
t = bG-l*np.dot(X, aG)
erreur_transformation = np.linalg.norm(B-l*X@A+t@u)
```

La fonction Procuste nous renvoie donc le Tuple $(\lambda, X, t, \|B - \phi(A)\|_F^2)$

4.

La méthode *Procuste* nous a permis de déterminer l'erreur de transformation en entrant 2 matrices même taille. Nous cherchons donc à appliquer notre fonction au problème de la reconnaissance des chiffres manuscrits.

Nous allons maintenant définir une fonction *Comparaison* qui va nous permettre de trouver le résultat de la prédiction sur le chiffre manuscrit *x* qui est une ligne du fichier *mnist_train.csv*.

On récupère dans une matrice A, les lignes correspondant à l'image du chiffre moyen obtenu grâce à la fonction *chiffremoy*. Afin de le faire pour tous les chiffres, on parcourt la matrice obtenue

avec *chiffremoy* ligne par ligne. Dans le même temps on récupère *x=train_data(i,1 :).reshape(784,1)* qu'on transpose et qu'on stocke dans une variable B=x .T puis on applique la fonction Procuste(A,B) .

Ainsi on peut donc récupérer le tuple obtenu grâce à la fonction Procuste mais surtout l'erreur de transformation. On injecte ainsi chaque erreur de transformation dans une variable *veccomp* de taille 10, qui prend pour chaque indice l'erreur de transformation entre le chiffre défini par le *x* et le chiffre moyen défini par le A

Exemple :

Si la ligne 3 du fichier *mnist_train.csv* correspond à un 3 alors l'erreur de transformation entre le 3 défini par *chiffremoy* sera faible. Ainsi *veccomp[3]* prendra cette valeur de l'erreur de transformation. *veccomp[4]*= erreur de transformation entre 3 défini par le fichier et 4 défini par *chiffremoy* etc.

Une fois la boucle ayant parcouru toutes les lignes de la matrice des nombres moyens obtenus grâce à *chiffremoy*, on isole la valeur minimale de *veccomp* qui correspond à l'endroit où l'erreur de transformation est la plus faible. Chaque indice correspondant à un nombre, on peut donc définir la prédiction de l'algorithme en fonction la position dans *veccomp* ou l'erreur est minimale.

```
veccomp = np.ones(10)
    for i2 in range(10):
        A = CM[i2, :].reshape((1, 784))
        B = x.T
        l, X, t, erreur_transformation = Procuste(A, B)
        veccomp[i2] = erreur_transformation
    resultat = np.where(veccomp == np.min(veccomp))[0]
    return veccomp, resultatveccomp = np.ones(10)
```

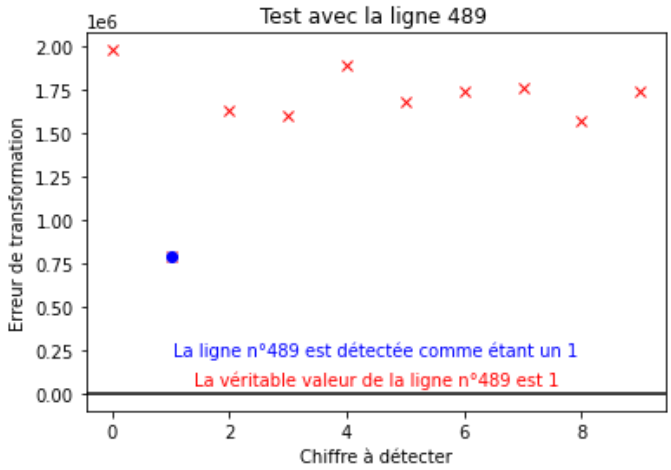
Ce qui nous donne si on exécute la fonction pour i=25

```
i=25
x=test_data[i, 1:].reshape((784, 1))
print(Comparaison(x, CM=chiffremoy(train_data))[0])
```

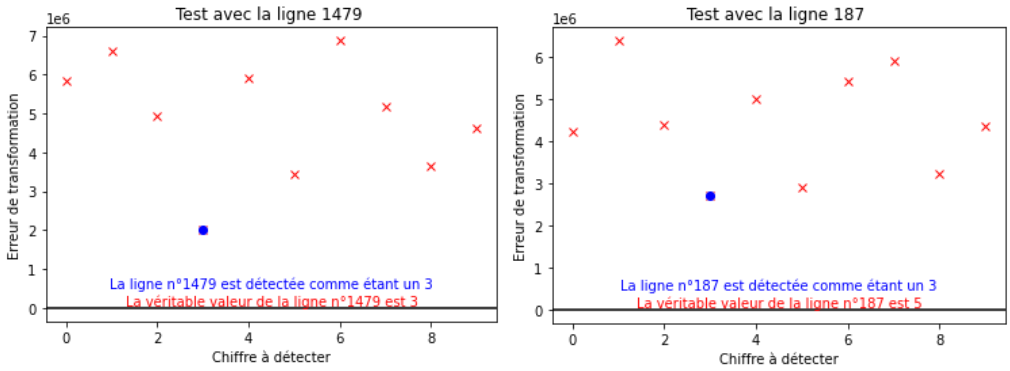
✓ 0.4s

```
[ 2366627.04738633 18072818.91367777  8148188.46541405 11616877.04122337
 10515214.83664291  6933348.00737358  8101272.81251491 12462610.17863443
  9662242.79653537 12256119.91902755]
```

Voici un code que nous avons réalisé mais qui n'était pas demandé pour pouvoir visualiser l'erreur de transformation pour une ligne en fonction de chiffres :



On remarque que le 3 et le 5 sont souvent proches, en effet :

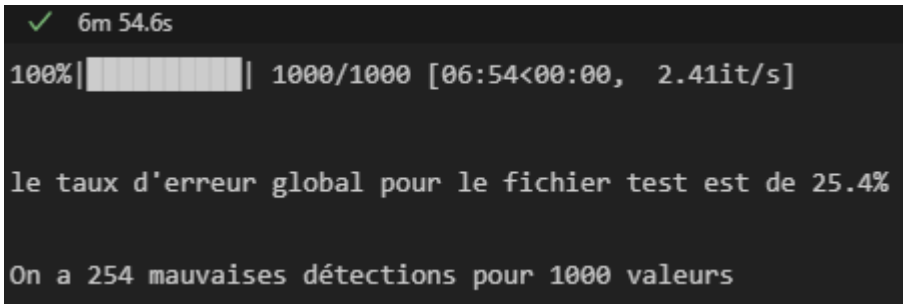


Il est possible de dire que le 3 ressemble bien moins a un 6 que le 5, si l'on compare alors l'écart avec un chiffre qui ne leur ressemble pas à tous les deux on peut faire une condition qui nous permet de corriger cette erreur.

Pour cela on a codé :

```
# Tentative de correction bonus de l'erreur 3/5
    if resultat[0] == 3:
        if veccomp[6] < veccomp[7]:
            resultat[0] = 5
```

Malheureusement l'on obtient :



25.4% > 20.9%
Même si l'idée semblait bonne, la précision n'est pas améliorée elle est même diminuée.

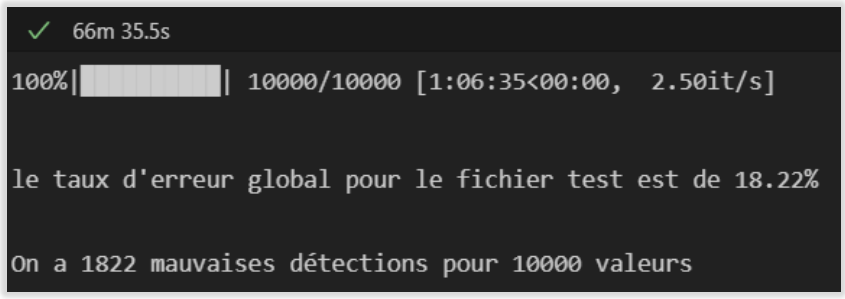
5.

Maintenant que toutes les fonctions sont implémentées, nous définissons une fonction *taux_2_reconnaissance*. On effectue une boucle allant de 0 à l avec l le nombre de lignes du fichier test_data que l'on veut traiter. Grace à cela on peut donc parcourir toutes les lignes du fichier test_data si on définit *l=np.shape(test_data)*.

On fait intervenir la fonction comparaison prenant en entrée x (une ligne du fichier test_data) et CM (la matrice obtenue avec *chiffremoy)*. Ainsi la fonction comparaison (x,CM) va nous renvoyer pour chaque ligne de *test_data*, *veccomp* et *resultat* qu'on va stocker respectivement dans 2 variables *a* et *b*. Ainsi b va nous renvoyer l'endroit de *veccomp* ou l'erreur est la plus faible et donc le chiffre prédit.

On va donc comparer la valeur de *b* avec la première composante de chaque ligne de *test_data* (correspondant au véritable chiffre). Ainsi si la valeur de b est différente du véritable chiffre, on définit un compteur *f0* qui augmentera de 1 à chaque erreur de prédiction.

Ainsi on obtient le taux de reconnaissance en mettant *f0* en pourcentage et donc le taux de réussite pour l'analyse procustéenne.



On constate que le taux de reconnaissance globale n'est pas amélioré en comparant aux valeurs des méthodes précédentes. Nous allons donc tenter de mélanger les deux nouvelles méthodes (QR et Procusteenne) que nous venons d'utiliser afin de tenter d'améliorer ce taux de reconnaissance.

III. Le mélange

1.

Après avoir testé les méthodes de Cholesky et Procustéenne, nous pouvons tenter de regrouper ces deux méthodes pour optimiser encore plus notre taux de réussite de détection des chiffres. En combinant ces deux algorithmes, nous pouvons espérer obtenir un algorithme encore plus précis. Pour cela nous allons utiliser des éléments les fonctions principales des programmes précédents :

- *Fglobale()* de la partie 1
- La matrice de vecteurs solutions *matrix_soleps* de la partie 1
- *Comparaison()* et *chiffremoy()* de la partie 2

On pose une donnée test de la forme : *x= train_data[i,1 :].reshape((784,1))* sur laquelle on va venir tester la combinaison des fonctions. Pour cela, on définit une fonction *reconnaissance()* qui prend en paramètres la donnée *x* (qui correspond donc à une ligne de la base de donnée train_data), les vecteurs solution dans la matrice *matrix_soleps* de la partie 1 et un coefficient noté *coeff*, réel strictement positif qui va nous permettre par la suite, à la manière de espilon dans la partie 1n d'optimiser le taux de réussite globale de cet algorithme combiné.

La fonction *reconnaissance()* renvoie comme pour les parties précédentes un vecteur *resultat* de taille 10 définie de la façon suivante :

```
v=coeff*(1/comparaison(test_data[i,1:],CM)[0])
rep=fglobale(test_data[i,1:].reshape((784,1)),SOL).reshape(np.shape(v))
resultat=v+rep
```

En réalité, comme pour les parties précédentes, chaque composante du vecteur en sortie correspond à un degré de correspondance à chaque chiffre. Ainsi, il nous suffit de créer une deuxième fonction comme pour la partie 1 notée *reponsev2()* qui récupère la composante la plus élevée du vecteur et qui renvoie son indice dans le vecteur, ce qui correspond au chiffre prédit par l'algorithme.

2.

Concernant le taux de réussite du programme combiné, c'est pour cette élément que le coefficient entre en jeu. Comme pour les autres parties, on utilise la table de données test_data ainsi que les labels présents sur la première colonne de la table (on vérifie par ailleurs toutes les données donc 10000 au total). Ainsi, en initialisant un compteur et en l'augmentant de 1 à chaque fois que la sortie de *reponsev2()* correspond au label de la ligne que l'on traite, on peut obtenir le taux de réussite de l'algorithme grâce à la formule suivante :

$$Taux_{reussite} = \frac{compteur}{10000}$$

Pour trouver la valeur maximale du taux de réussite, nous devons alors tester l'algorithme en faisant varier le coefficient *coeff* sur une grande plage de valeur.

Nous ne sommes pas parvenus à trouver un coefficient optimal malgré différente pistes. La principale d'entre-elles consistait à utiliser la même méthode que pour epsilon à la partie 1 et en réitérant plusieurs fois. Nous imaginions diviser les coefficients à l'aide d'un *np.linspace* puis de tester pour 1000 valeurs la valeur du coefficient donnant le meilleur taux de réussite. Puis réitérer cette opération en considérant un intervalle plus réduit autour de ce nouveau coefficient afin d'arriver après plusieurs itérations à un coefficient optimale le plus précis possible.

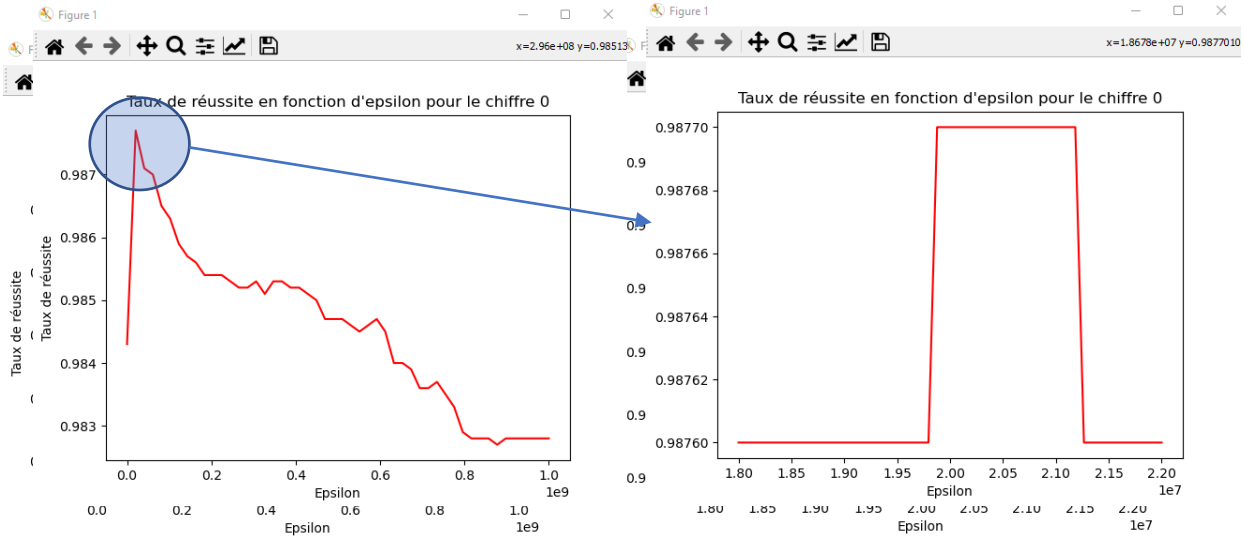
N'arrivant pas à trouver cette valeur de coefficient optimale nous avons donc échangé avec des camarades d'autres groupes et avons pu utiliser un coefficient approchant le coefficient optimal qu'ils avaient trouvé.

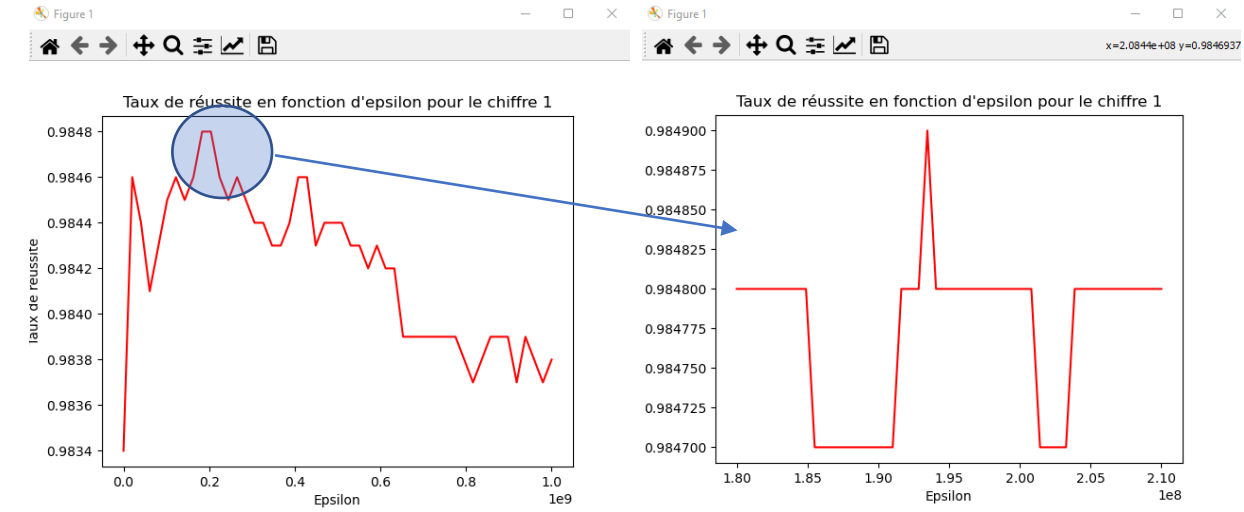
En utilisant ce coefficient de 6000 nous avons finalement obtenu un taux de réussite global de 86,48%. Pour rappel, avec la méthode de Cholesky nous avons obtenu un taux global de 86,52% et avec la méthode procustéenne, un taux de 81,78%.

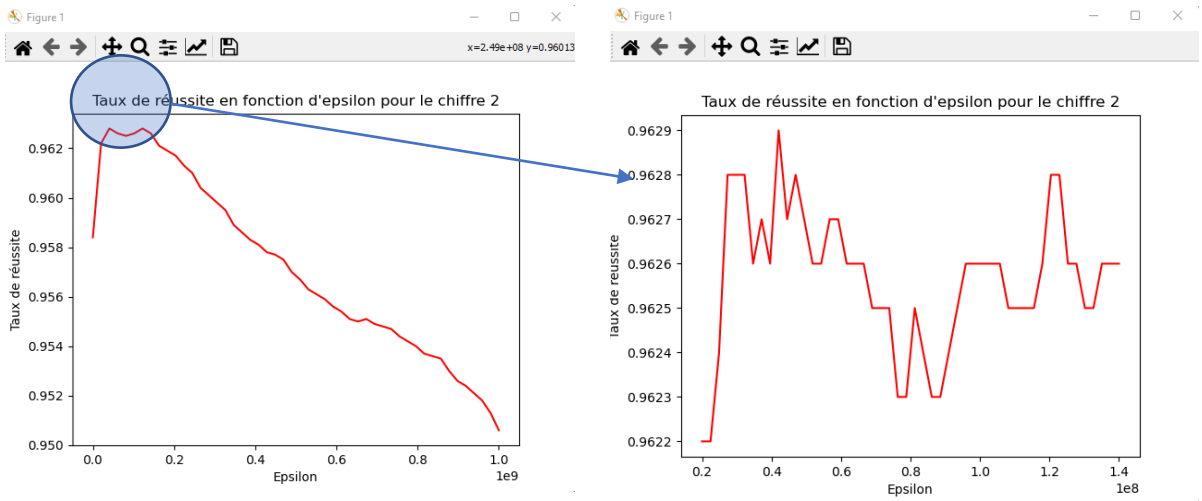
On peut donc en conclure que la combinaison des deux méthodes ne permet pas d'optimiser davantage l'algorithme, qui est déjà très efficace avec la méthode de Cholesky.

Annexe

- Graphiques représentant le taux de réussite en fonction d’epsilon pour les autres chiffres :







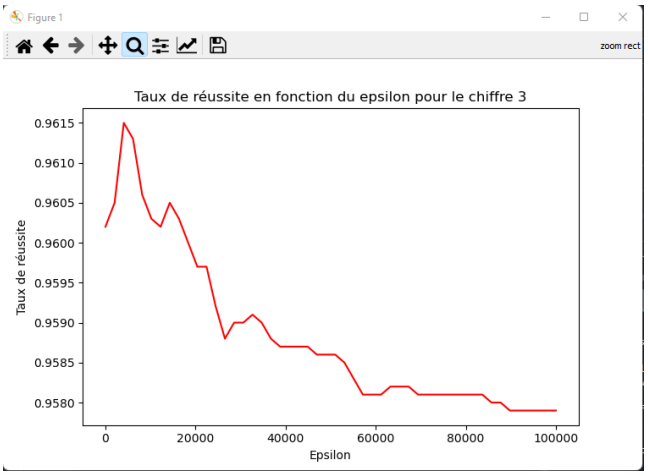


Figure 1
zoom rect
Taux de réussite en fonction du epsilon pour le chiffre 3
Taux de réussite
0.9615
0.9610
0.9605
0.9600
0.9595
0.9590
0.9585
0.9580
0
20000
40000
60000
80000
100000
Epsilon

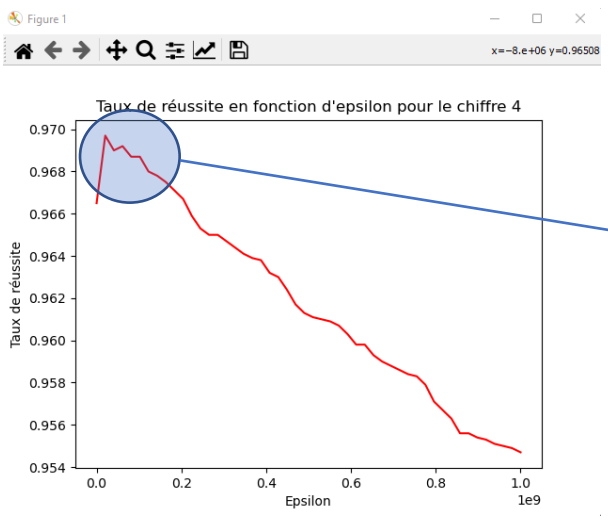


Figure 1
x=−8.e+06 y=0.96508
Taux de réussite en fonction d'epsilon pour le chiffre 4
Taux de réussite
0.970
0.968
0.966
0.964
0.962
0.960
0.958
0.956
0.954
0.0
0.2
0.4
0.6
0.8
1.0
1e9
Epsilon


Figure 1
Taux de réussite en fonction d'epsilon pour le chiffre 4
0.969900
0.969875
0.969850
0.969825
0.969800
0.969775
0.969750
0.969725
0.969700
1.55
1.60
1.65
1.70
1.75
1.80
1.85
1e7
Epsilon

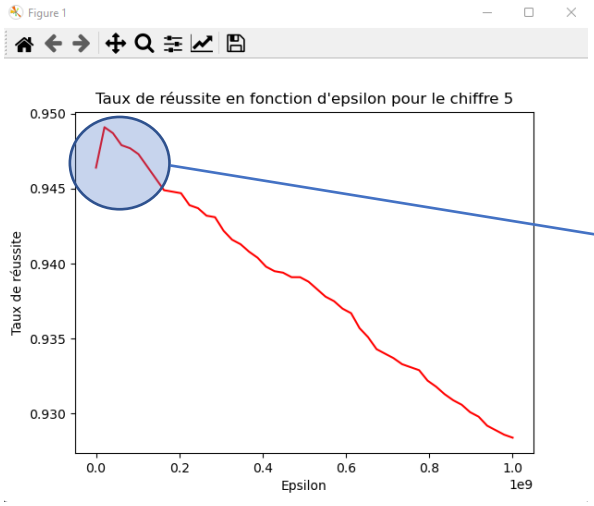


Figure 1
Taux de réussite en fonction d'epsilon pour le chiffre 5
Taux de réussite
0.950
0.945
0.940
0.935
0.930
0.0
0.2
0.4
0.6
0.8
1.0
1e9
Epsilon

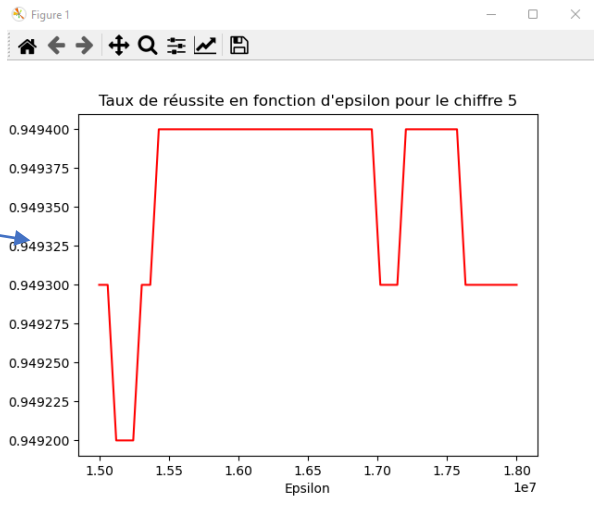


Figure 1
Taux de réussite en fonction d'epsilon pour le chiffre 5
0.949400
0.949375
0.949350
0.949325
0.949300
0.949275
0.949250
0.949225
0.949200
1.50
1.55
1.60
1.65
1.70
1.75
1.80
1e7
Epsilon

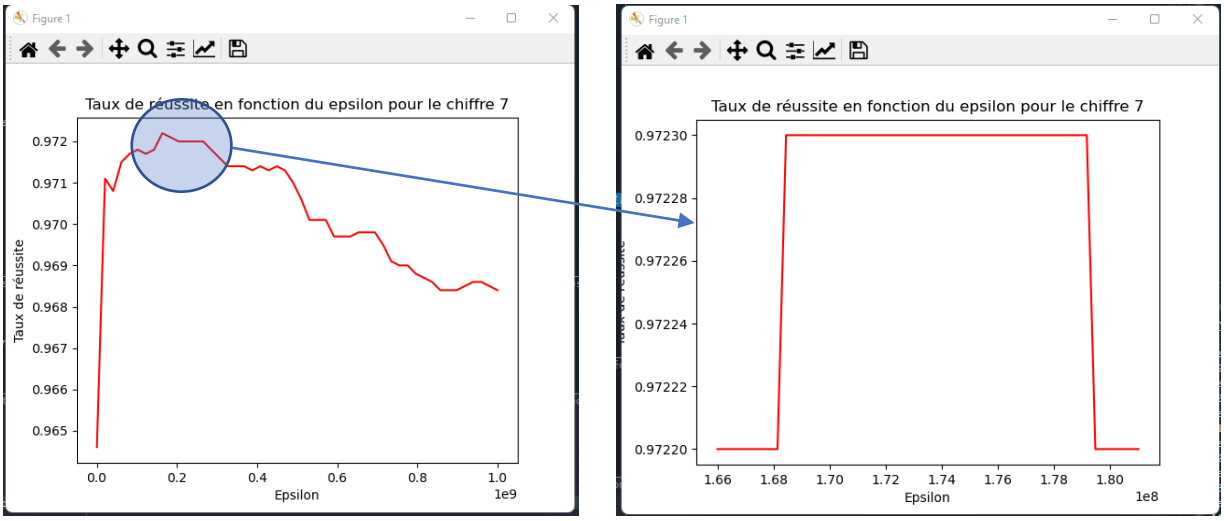


Figure 1
Taux de réussite en fonction du epsilon pour le chiffre 7
Taux de réussite
Epsilon
1e9
Figure 1
Taux de réussite en fonction du epsilon pour le chiffre 7
Epsilon
1e8

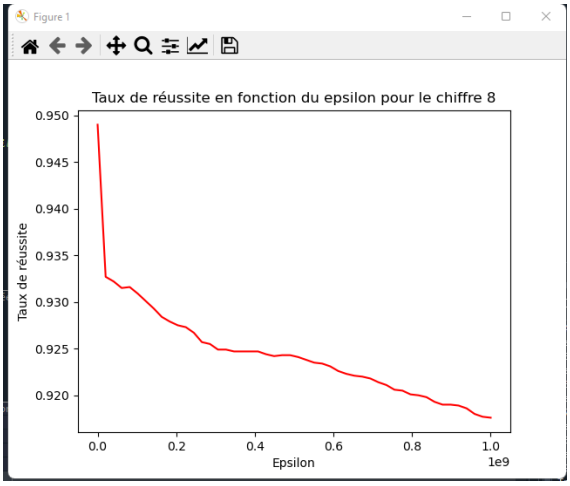


Figure 1
Taux de réussite en fonction du epsilon pour le chiffre 8
Taux de réussite
Epsilon
1e9

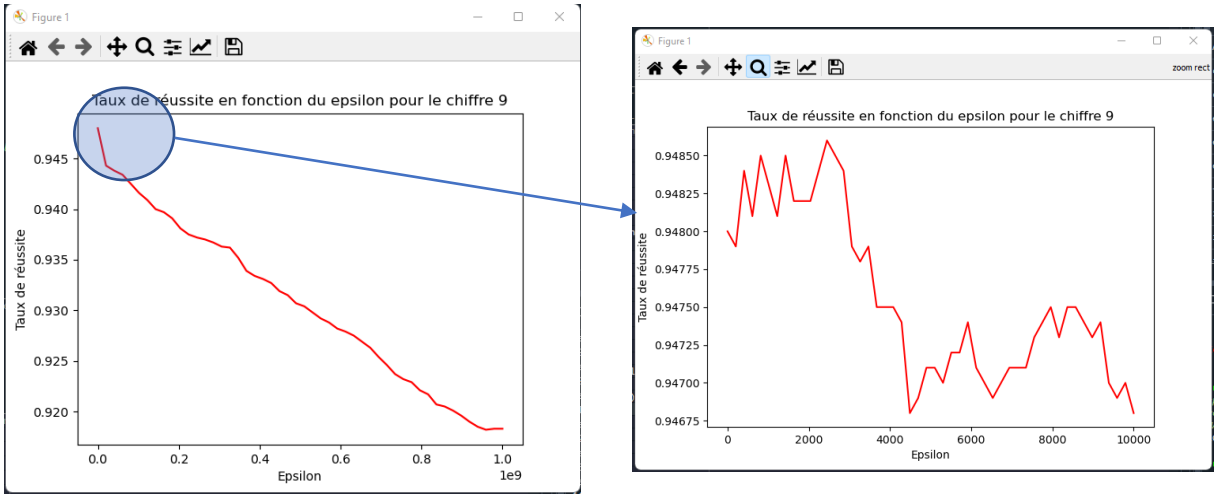


Figure 1
Taux de réussite en fonction du epsilon pour le chiffre 9
Taux de réussite
Epsilon
1e9
Figure 1
Taux de réussite en fonction du epsilon pour le chiffre 9
Taux de réussite
Epsilon