TP 3 : Gradient conjugué pour un problème d'inpainting.

```
In [ ]: %pylab inline
```

1. Mise en place de la méthode du gradient conjugué

On veut mettre en place une méthode de gradient conjugué qui soit souple, et puisse résoudre dans le cas général la minimisation de $\frac{1}{2}\langle x, \mathcal{A}(x)\rangle + \langle b, x\rangle$, lorsque $\mathcal{A}$ correspond à la fonction $X \mapsto AX$, où X est la représentation de x dans une base orthonormale, et A est une matrice symétrique définie positive.

On veut que la méthode fonctionne dans (au moins) deux cadres, lorsque x est un vecteur, ou x une matrice à n_1 lignes et n_2 colonnes (ce dont on aura besoin dans l'application au problème d'inpainting).

On utilisera donc toujours dès qu'il faudra calculer un produit scalaire (ou une norme) la fonction suivante, qui a l'avantage de fonctionner à la fois sur des matrices et des vecteurs (de type array dans les deux cas) :

```
In [ ]: def prodscal(x,y):
            return sum(x*y)
```

(a) Programmer la méthode de gradient conjugué en donnant $\mathcal{A}$ comme argument sous la forme d'une fonction, et en utilisant seulement la fonction de produit scalaire ci-dessus dès qu'on a besoin de calculer des produits scalaires ou des normes. On pourra se donner un nombre maximal d'itérations, et prendre comme critère d'arrêt que la norme du gradient $\|\nabla f(x_k)\|$ est inférieure à $\varepsilon\|\nabla f(x_0)\|$, où ε est une tolérance relative donnée. Ceci permet d'avoir un critère qui soit cohérent même quand la dimension change grandement. La méthode devra renvoyer l'itérée x_k correspondant à la dernière itération, ainsi que la liste des $\|\nabla f(x_k)\|$, afin de pouvoir faire des tests de vitesse de convergence.

(b) Générer une matrice aléatoire N contenant plus de colonnes que de lignes. Définir A comme étant la matrice NN^T. Afficher le nombre de conditionnement de A (la plus grande valeur propre divisée par la plus petite) à l'aide de la fonction `cond`. Enfin définir une fonction $\mathcal{A}$ qui prend pour argument une seule variable x et qui renvoie $\mathcal{A}(x) = Ax$. Intégrer un compteur global dans cette fonction, qui permettra de vérifier son nombre d'appel.

Définir ensuite aléatoirement un vecteur b de la bonne taille, et vérifier qu'on peut appliquer la fonction $\mathcal{A}$ sur b.

(c) Tester la méthode du gradient conjugué sur $\mathcal{A}$ et b. Que peut-on prendre pour x_0, si l'on n'a aucune indication ?

Observer la convergence de la norme du gradient vers 0, à l'aide d'un graphique adapté. Expliquer pourquoi la méthode ne renvoie pas exactement 0 au bout de n itérations.

Vérifier que le nombre d'appel de la fonction (calculé en utilisant le compteur) correspond bien au nombre de passages dans la boucle (qui doit être la longueur de la liste des norme des gradients).

(d) Vérifier que la méthode fonctionne lorsqu'on prend b dans un espace de matrices (par exemple ayant autant de lignes que A, et au moins deux colonnes). Quelle est alors la dimension de l'espace dans lequel on travaille ? En combien d'itérations obtient-on une matrice de norme très petite ? Est-ce cohérent avec le résultat du cours ?

2. Outils de manipulation d'images, problème d'inpainting.

(a) Voici tout d'abord une fonction permettant de lire une image en niveau de gris sous la forme d'un tableau F. La fonction `shape` permet de donner les tailles dans chaque direction.

```
In [ ]: F=imread('mystere.png')
        pixelsv,pixelsh=shape(F)
        print(pixelsv,pixelsh)
```

On a donc dans notre cas un tableau de taille $n_1 \times n_2$, voyons à quoi ressemblent les premières lignes et colonnes.

```
In [ ]: F[:20,:5]
```

On voit qu'une image en niveau de gris est codée par des valeurs dans $[0,1]$, correspondant à la luminosité du pixel considéré (0 pour noir, 1 pour blanc).

Voici maintenant comment afficher une image sur une figure dont on peut varier au préalable la taille. La fonction `gray()` sert à ce que toutes les images affichées par la suite le soient en niveaux de gris. Le dernier point-virgule sert à ne pas afficher la sortie de `axis` (qui retourne sinon la valeur des bornes des axes).

```
In [ ]: gray()
        figure(figsize=(6,6))
        imshow(F)
        axis('off');
```

Le problème d'« inpainting » est le suivant : on se donne une image dont certains pixels n'ont pas été transmis, ce sont ceux pour lesquels la valeur vaut exactement 0, et on aimerait « interpoler » entre ces pixels de façon à remplir ces zones indéterminées.

On modélise l'image par une matrice $F = (f_{i,j}) \in M_{n_1,n_2}(\mathbb{R})$, où $f_{i,j} \in [0,1]$ est la valeur de la luminosité au pixel de la ligne i et colonne j, et n_1 (resp. n_2) est le nombre de lignes (resp. de colonnes) de pixels. Et on va noter $M = (m_{i,j})$ la matrice du masque, c'est à dire $m_{i,j} \in \{0,1\}$ avec $m_{i,j}$ qui vaut 1 si le pixel a été transmis et 0 si le pixel n'a pas été transmis (lorsque $f_{i,j} = 0$). Ce sont les données du problème.

Pour des raisons pratiques, on notera $\overline{M} = (\overline{m}_{i,j}) = (1 - m_{i,j})$ le contraire du masque, c'est-à-dire dont les entrées valent 1 si le pixel n'a pas été transmis et 0 sinon.

Le but est de retrouver une image complète, qui corresponde le plus possible à l'image originale. Pour cela on cherche donc une matrice $U = (u_{i,j})$ qui soit telle que $u_{i,j} = f_{i,j}$ si le pixel a été transmis et qui soit la plus « lisse » possible, elle va minimiser une sorte de norme L^2 d'une discrétisation du gradient. On considère donc le problème d'optimisation suivant :

$$\inf_{U \in C} \frac{1}{2} \sum_{i=1}^{n_1-1} \sum_{j=0}^{n_2-1} (u_{i,j} - u_{i-1,j})^2 + \frac{1}{2} \sum_{i=0}^{n_1-1} \sum_{j=1}^{n_2-1} (u_{i,j} - u_{i,j-1})^2,$$

où l'ensemble des contraintes est

$$C = \{U \in M_{n_1,n_2}(\mathbb{R}) | \forall (i,j) \text{ tels que } m_{i,j} = 1, \text{ on a } u_{i,j} = f_{i,j}\}.$$

On a fait commencer les différents indices à 0, pour être cohérent avec Python.

On a vu en TD que cela équivaut à minimiser $\frac{1}{2}\langle X, \mathcal{A}(X)\rangle + \langle B, X\rangle$, lorsqu'on pose $X = U - M \circledast F$, pour X appartenant au sous-espace

$$E = \{X \in M_{n_1,n_2}(\mathbb{R}) | \forall (i,j) \text{ tels que } m_{i,j} = 1, \text{ on a } x_{i,j} = 0\},$$

où on a noté $\circledast$ la multiplication élément par élément (c'est tout simplement l'opérateur $*$ en Python), et où $\langle \Delta, \Delta \rangle$ correspond au produit scalaire standard sur les matrices (qui correspond bien à la fonction codée dans la première partie).

La fonction $\mathcal{A}$ $(E \to E)$ est donnée par

$$\mathcal{A}(X) = (D_{n_1}^T D_{n_1} X + X D_{n_2}^T D_{n_2}) \circledast \overline{M},$$

où D_k est une matrice à $k-1$ lignes et k colonnes, ayant des -1 sur la diagonale principale et des 1 sur la diagonale située juste au-dessus (et des zéros ailleurs).

Enfin B est donné par

$$B = (D_{n_1}^T D_{n_1} (M \circledast F) + (M \circledast F) D_{n_2}^T D_{n_2}) \circledast \overline{M}.$$

(b) Dans le cas qui nous intéresse, la matrice F est donnée par le tableau `F` obtenu par lecture de l'image, et on considère que les pixels non-transmis sont ceux pour lesquels la valeur dans `F` vaut zéro. Que vaut alors $M \circledast F$? Calculer le tableau correspondant à la matrice $\overline{M}$ à partir de `F`, et calculer la dimension n du problème d'optimisation. Quel est la proportion de pixels transmis ?

(c) On cherche à obtenir une méthode efficace pour calculer (X).

— Calculer $D_k^T D_k$ pour des petites valeurs de k. Pour des tableaux à deux dimensions, la multiplication matricielle est donnée par la fonction `dot` : `dot(G,H)` renvoie la multiplication matricielle des tableaux `G` et `H`, à condition que `G` ait autant de lignes que `H` a de colonnes. La transposition est donnée par la fonction `transpose`.
— Y a-t-il un intérêt à stocker $D_{n_1}^T D_{n_1}$, et $D_{n_2}^T D_{n_2}$?
— Trouver une méthode pour calculer rapidement $D_k^T D_k X$ pour X une matrice de taille $k \times \ell$ en additionnant et soustrayant seulement des blocs de taille $k-1 \times \ell$, et sans utiliser de multiplication matricielle. On pourra commencer par voir comment faire seulement $D_k X$, puis ensuite comment calculer rapidement $D_k^T Y$.
— Vérifier le résultat en prenant un X aléatoire pour des petites valeurs de k et ℓ.
— Évaluer le temps de calcul des deux méthodes pour des k et ℓ plus grands. On pourra utiliser la commande `%%timeit` de IPython qui évalue le temps mis à exécuter les commandes d'une cellule (exemple ci-dessous).

```
In [ ]: %%timeit
        N=100
        G=randn(N,N)
        dot(transpose(G),G)
```

(d) Dans le cas du problème qui nous intéresse, programmer la fonction $X \mapsto \mathcal{A}(X)$ de manière efficace par rapport à ce qui a été dit dans la question **(e)**. Évaluer également la matrice B de la même manière.

(e) Résoudre le problème d'inpainting grâce à la méthode de gradient conjugué : comment trouver le U qui minimise la première fonction, lorsque l'on a trouvé un X qui minimise la fonction quadratique sur E. Afficher l'image obtenue correspondant à la matrice U. Combien d'itérations sont-elles suffisantes pour obtenir une bonne image ? Comparer avec la dimension de l'espace E.

3. Résolution par méthode de descente de gradient à pas optimal et comparaison.

(a) Programmer comme dans la partie précédente la méthode de descente de gradient à pas optimal dans le cas particulier de la minimisation d'une fonction quadratique $\frac{1}{2}\langle X, \mathcal{A}(X)\rangle + \langle B, X\rangle$.

(b) L'appliquer aux mêmes cas test que dans la partie précédente, et comparer les performances des méthodes.

4. Minimisation différente pour l'inpainting

On cherche à éviter certains problèmes qui sont liés au fait que la minimisation est quadratique (en particulier, les bords sont mal rendus). On modifie la fonction à optimiser, et on s'intéresse au problème suivant :

$$\inf_{U \in C} J(U), \qquad \text{avec } J(U) = \frac{1}{2} \sum_{i=1}^{n_1-1} \sum_{j=1}^{n_2-1} \sqrt{\delta + (u_{i,j} - u_{i-1,j-1})^2 + (u_{i,j-1} - u_{i-1,j})^2},$$

où δ est un paramètre de régularisation qu'on aimerait prendre assez petit. On pose de nouveau $X = U - M \circledast F$ et on cherche donc de nouveau à minimiser $f(X) = J(X + M \circledast F)$ pour $X \in E$.

(a) Pourquoi ne peut-on plus appliquer la méthode de gradient conjugué pour ce problèmes ? Quelle méthode pourrait-on prendre ?

(b) À l'aide du calcul de la dérivée partielle de f par rapport à $x_{i,j}$, programmer une méthode qui renvoie le gradient sans faire de multiplication matricielle.

(c) Programmer la méthode choisie, et comparer « visuellement » le rendu des images correspondant au minimum de la fonction, par rapport aux résultats obtenus dans les parties précédente.

Comparer également le nombre d'itérations nécessaires dans les différents cas.