Sujet TP3

Nous travaillerons encore dans le repertoire **TP_ANANUM/** avec l'environnement virtuel **.venv** activé:

```
source .venv/bin/activate
python test_config.py
```

- Téléchargez et **décompressez** dans ce dossier l'archive tp3.zip présente sur moodle.

Objectifs du TP:

- Implémentez la méthode du gradient conjugué pour résoudre un système linéaire $A\mathbf{x} = \mathbf{b}$,
- Appliquer cet algorithme au problème de reconstruction d'images où la matrice A n'est pas construite explicitement.

Partie 1

Dans cette première partie vous implémenterez la méthode du gradient conjugué pour résoudre le problème linéaire suivant:

$$A\mathbf{x} = \mathbf{b},$$

où A est une matrice symétrique définie positive aléatoire et $\mathbf{b}$ est un vecteur aléatoire. Nous rappelons que la méthode du gradient conjugué est définie par les relations de recurrences suivantes:

```
Choose x^0
Set r^0 = b-A*x^0
Set p^0 = r^0
For k = 0 to N-1 {
    if (||r^k|| < eps) {
        break
    }
    a       = ||r^k||^2 / ||p^k||^2_A
    x^{k+1} = x + a * p^k
    r^{k+1} = r - a* A * p^k
    b       = ||r^{k+1}||^2 / ||r^{k}||^2
    p^{k+1} = r^{k+1} + b * p^k
}
```

Dans le fichier *main_tp3.py* complétez la fonction suivante:

```
"""
Function to solve a linear system Ax=b with conjugate gradient method
Input:
A : is a square matrix of dimension N x N, must be symmetric positive definite
b : is a vector of dimension N
Output:
x : approximation of the solution
err : list of the residuals norms computed at each step of the iteration
"""
def conjugate_gradient(A, b, eps = 1e-5):
    N   = b.shape[0]
    err = np.ones(shape=(N,))

    # Modify those instructions here
    x = np.zeros((N,))
    r = np.ones((N,))
    p = np.ones((N,))

    for k in range(N):
        err[k] = np.sqrt(np.inner(r,r))

        if (err[k] < eps):
            break

        # Complete instructions here

    return x, err[0:k]
```

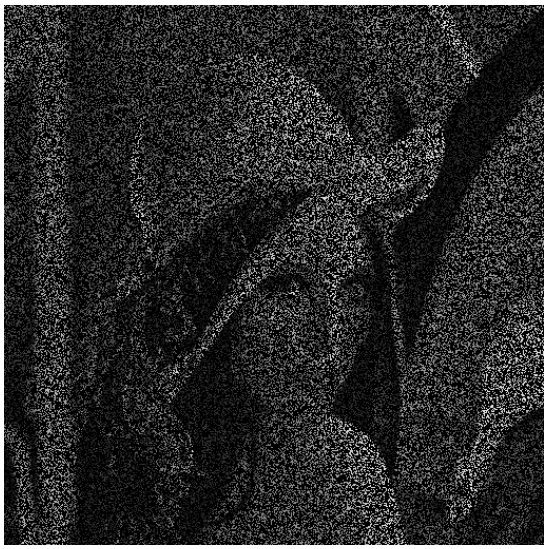
Exécutez votre code et observez la courbe de convergence des résidus. En particulier remarquez le nombre d'itérations nécessaires pour obtenir la précision ϵ choisie.

Commentaires:

Bonus: Optimisez votre code pour évitez la répétition des calculs identiques.

Partie 2

Cette partie du TP est emprunté à Gabriel Peyré: Numerical Tours. Nous considérons le problème d'inpainting, qui correspond à l'interpolation de données manquantes dans une image. Un ami vous envoi une photo de Lena, malheureusement dans la transmission, 70% des pixels de l'image se sont perdus

Image originale	Image reçue
	

Votre objectif est de retrouver une approximation de l'information manquante.

Remarque: La méthode développée ici ainsi que ses variantes peuvent servir à la définitions de standard de compression pour des fichiers images, par exemple le format jpg fonctionne sur ce principe (dans l'espace de Fourier).

Modélisation Mathématique

Les images sont modélisées par des vecteurs dans un espace de grande dimension, si l'image fait 512×512 pixels alors l'espace est de dimension $N = 262144$:

$$\mathbf{x} \in \mathbb{R}^N, \quad x_i \in [0,1].$$

Les composantes x_i du vecteur $\mathbf{x}$ indiquent le niveau de gris du pixel i: $x_i = 0$ correspond à du noir et $x_i = 1$ à du blanc.

Les données manquantes sont modélisées par un masque binaire $M \in \{0,1\}^N$, où $M_i = 0$ si le pixel i est manquant et $M_i = 1$ sinon. La dégradation d'une image $\mathbf{x}$ est alors donnée par l'application linéaire

$$\Phi(\mathbf{x}) = M \otimes \mathbf{x},$$

où $\otimes$ est la multiplication composante par composante de deux vecteurs: $\Phi(\mathbf{x})_i = M_i * x_i$ pour tout $1 \leq i \leq N$.

Méthodologie de reconstruction d'une image à partir d'observations dégradées $\mathbf{y}$:

> Nous souhaitons obtenir une image $\mathbf{x} \in \mathbb{R}^N$ telle que:
>
> 1. $\mathbf{x}$ coïncide le plus possible avec les observations, càd $\Phi(\mathbf{x}) \approx \mathbf{y}$;

> 2. $\mathbf{x}$ est une image *lisse*, on mesure le caractère lisse d'une image avec la norme du gradient $\|\nabla \mathbf{x}\|_2$;

Cela nous amène à considérer le problème de minimisation suivant:

$$\text{Trouver } \mathbf{x} \text{ réalisant le } \min_{\mathbf{x}\in\mathbb{R}^N} \|\mathbf{y} - \Phi(\mathbf{x})\|^2 + \lambda \|\nabla \mathbf{x}\|^2.$$

Ce problème possède une solution unique si $\ker(\Phi) \cap \ker(\nabla) = \{0\}$. Cette condition est vérifiée dans notre application car $\ker(\nabla)$ est l'ensemble des images constantes. La solution est alors obtenue en résolvant le système linéaire suivant:

$$A\mathbf{x} = \mathbf{b}, \quad \text{où} \quad \begin{cases} A & = \Phi^*\Phi - \lambda\Delta = \Phi - \lambda\Delta \\ \mathbf{b} & = \Phi^*(\mathbf{y}) = \mathbf{y} \end{cases}.$$

La valeur du paramètre λ doit être petite.

```
l = 0.01
```

L'opérateur laplacien Δ est défini comme suit

$$\Delta = -\nabla^* \circ \nabla$$

et dans le fichier *utils.py*, nous donnons un implémentation simple des opérateurs gradient et divergence. Cette implémentation est basée sur une discrétisation différences finies du premier ordre avec des conditions aux bords périodiques.

Ce laplacien est un opérateur linéaire symétrique défini négatif donc l'opérateur A défini ci-dessus satisfait aux hypothèses de la méthode du gradient conjugué. Cependant obtenir l'expression matriciel de Δ est inenvisageable : pour une image de taille 512×512, le laplacien serait une matrice de taille 262144×262144 (≈ 512 Go de données). Même si on peut conjecturer que cette matrice aurait une structure creuse, (et donc peu de coefficients à stocker: ≈ 6 M), son calcul explicite serait néanmoins bien douloureux.

C'est pourquoi nous allons utiliser une particularité très intéressante de la méthode du gradient conjugué. en effet, pour cette méthode, inutile de connaître les coefficients de la matrice (contrairement aux méthodes de Jacobi et Gauss-Seidel par exemple). Il suffit de savoir calculer l'action de l'application linéaire sur un vecteur:

$$A\mathbf{x}$$

Implémentation

L'implémentation se déroule dans le fichier *main_tp3_img.py*. Si vous souhaitez observez l'action du gradient et du laplacien sur les images vous pouvez décommentez les lignes suivantes :

```
"""
g = grad(I)
fig, ax = plt.subplots(1,2)
plt.title("Gradient of image")
ax[0].imshow(g[:,:,0], cmap='gray')
ax[1].imshow(g[:,:,1], cmap='gray')
ax[0].title.set_text('$d/dx$')
ax[1].title.set_text('$d/dy$')
ax[0].axis('off')
ax[1].axis('off')
```

```
plt.show()
"""
...
"""
L = laplacian(I)
plt.axis('off')
plt.imshow(L, cmap='gray')
plt.title("Laplacian of image: $\Delta (x)$")
plt.show()
"""
```

Implémentez la fonction suivante pour calculer $A\mathbf{x}$ avec $A = \Phi - \lambda\Delta$; vous avez à votre disposition une fonction *mask(x)* et une fonction *laplacian(x)*.

```
def linear_operator(x):
    x   = x.reshape(N,N)
    # Complete here
    res = np.zeros((N,N))
    return res.reshape(-1)
```

Complétez la fonction *conjugate_gradient(A,b)* : reprenez le code précédent mais remplacez le produit de A avec un vecteur x par l'appel $A(x)$.

```
def conjugate_gradient(A, b, eps = 1e-5):
    N   = b.shape[0]
    err = np.ones(shape=(N,))

    # Modify those instructions here
    x = np.zeros((N,))
    r = np.ones((N,))
    p = np.ones((N,))

    for k in range(N):
        err[k] = np.sqrt(np.inner(r,r))

        if (err[k] < eps):
            break

        # Complete instructions here

    return x, err[0:k]
```

Commentez la convergence de l'algorithme et la qualité visuelle de la reconstruction.

Commentaires:

Bonus: Quelles autres applications du même type en imagerie imaginez vous ?