TP 11 - Images (suite)

Ce TP complète et achève le précédent, donc je vous invite à reprendre le script du TP précédent (ainsi que les images déjà téléchargées) et de compléter celui-ci. On suppose donc importées les bibliothèques `numpy` (avec le préfixe `np`), `matplotlib.pyplot` (avec le préfixe `mpl`) et la fonction `Image` du module `PIL`.

Les images dont il est question ici sont à télécharger dans `https://cahier-de-prepa.fr/pcsi-bdb/docs?rep=114`

1 Redimensionnement d'une image

Une opération très courante consiste à agrandir ou réduire la taille d'une image. Une image vectorielle pouvant être dessinée de n'importe quelle taille sans perte de qualité, nous ne nous intéresserons qu'aux images matricielles pour lesquelles différents algorithmes sont envisageables.

1.1 Plus proche voisin

Supposons une image initiale de n lignes de p pixels qu'on souhaite redimensionner selon un facteur f, de sorte que l'image finale comptera $n\,f$ lignes de $p\,f$ pixels (en pratique, $n\,f$ et $p\,f$ n'étant pas forcément des entiers, on en prendra l'entier le plus proche.) D'un pixel à la ligne i et colonne j de l'image redimensionnée, ses coordonnées dans l'image initiale seraient $j\,/\,f$ et $i\,/\,f$ (notez que les lignes sont numérotées de haut en bas, la première tout en haut étant d'indice 0 et la dernière d'indice $n-1$)

On arrondit $j\,/\,f$ et $i\,/\,f$ aux entiers x et y les plus proches (l'entier inférieur si la partie fractionnaire est au plus 0.5 et l'entier supérieur sinon, ce que réalise la fonction `round`) et pour ce pixel de coordonnées (j,i) de l'image résultante, on prend la couleur du pixel aux coordonnées (x,y) de l'image initiale.

Question 1. Recopier et compléter la fonction suivante qui réalise ce travail (ici, on se contente d'afficher l'image obtenue) :

```python
def redimensionne(image, rapport):
    n, p = image.shape # on considère ici des images en noir et blanc, donc une seule luminance
    n2 = round(n*rapport)
    p2 = round(n*rapport)
    output = np.zeros((n2, p2), dtype = np.uint8)
    # votre code ici...

    mpl.show(output)
```

Remarque. Dans le cas d'une réduction d'image, l'image qui résulte de ce traitement élémentaire est marquée par une perte d'information (certains pixels de l'image initiale ne sont pas du tout considérés) et souvent une image un peu bruitée. Dans le cas d'un agrandissement, on observe une pixelisation (qui n'est pas si différente de ce qu'on obtiendrait en observant à la loupe l'image initiale, où les pixels peu visibles à l'oeil nu apparaîtraient alors)

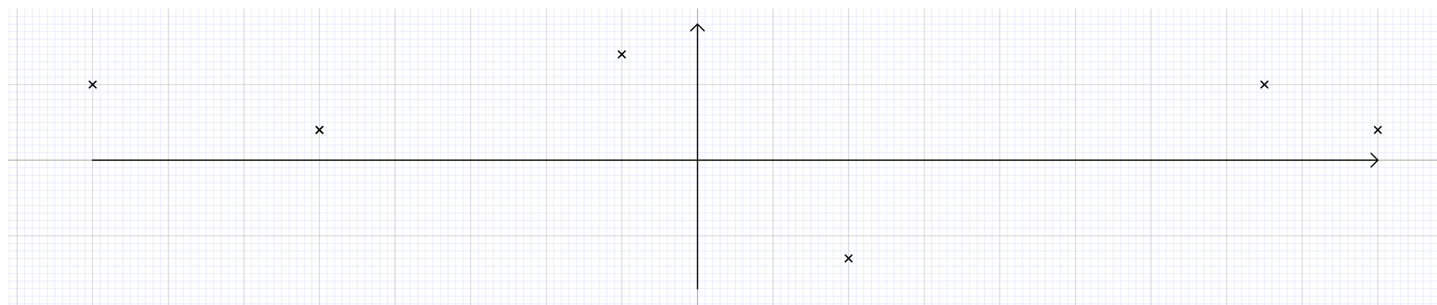
Question 2. Modifier la fonction précédente pour que, au lieu d'afficher l'image obtenue, on renvoie le tableau `output`.

Tester alors le traitement suivant :

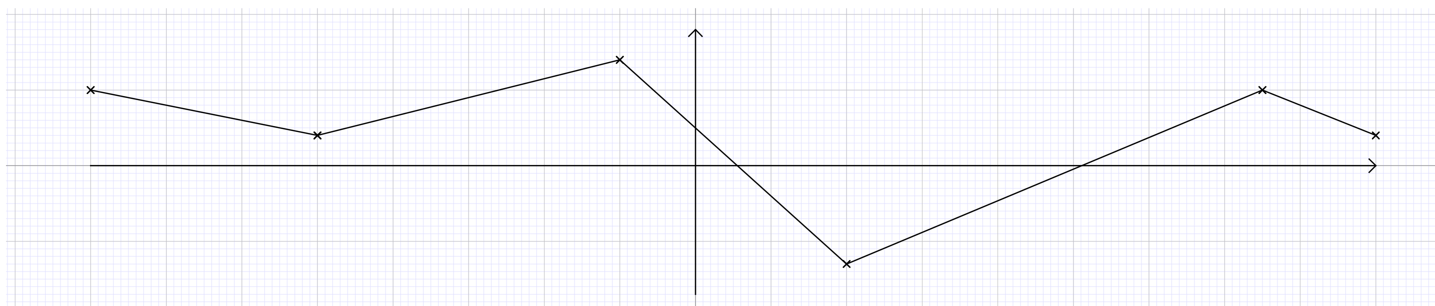
```python
mpl.imshow(redimensionne(redimensionne(chatGris,0.25),4))
```

1.2 Interpolation linéaire

Imaginons qu'on ne connaisse du graphe d'une fonction que quelques points particuliers (des mesures expérimentales, par exemple), et qu'on souhaite reconstruire ladite fonction :



La méthode de loin la plus simple consiste à réaliser une interpolation linéaire, et donc à considérer qu'entre deux abscisses où la fonction est connue, elle est affine (et on construit ainsi une fonction affine par morceaux) pour obtenir :

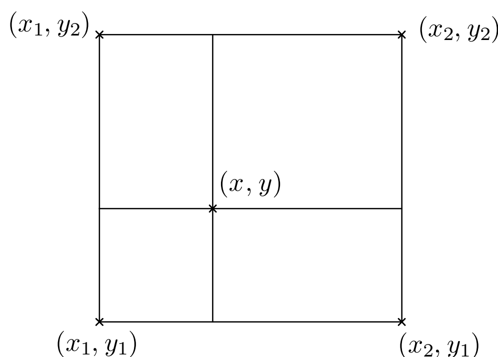


Question 3. On suppose connus deux points A et B du graphe de f, de coordonnées (x_1, y_1) et (x_2, y_2) (et aucun autre point connu ne figure entre les abscisses x_1 et x_2). Etant donné $x \in [x_1, x_2]$, déterminer l'ordonnée y du point d'abscisse x sur le segment $[A, B]$. On posera $\lambda = \frac{x - x_1}{x_2 - x_1}$.

(Il est judicieux de remarquer que $[A, B]$ est formé des points de coordonnées $(\lambda x_2 + (1 - \lambda) x_1, \lambda y_2 + (1 - \lambda) y_1)$ où λ parcourt $[0, 1]$)

1.2.1 Application au redimensionnement d'images : algorithme bilinéaire

L'idée de l'algorithme proposé ici est, afin de déterminer la couleur d'un pixel de l'image redimensionnée, de ne pas se contenter d'aller chercher la couleur du pixel le plus proche, mais d'aller chercher les quatre points dont la couleur est connue qui se situent autour de celui dont on cherche à déterminer la couleur :



Vous l'avez compris, le redimensionnement de notre image conduit à devoir déterminer quelle couleur attribuer au point de coordonnées (x, y) connaissant celles des points de coordonnées $(x_1, y_1), (x_2, y_1), (x_2, y_2)$ et (x_1, y_2). Au lieu de choisir comme dans l'algorithme précédent la couleur du point connu le plus proche (de coordonnées (x_1, y_1) selon le schéma précédent) on va désormais tenir compte des quatre couleurs connues, mais en faisant une moyenne pondérée des 4 couleurs connues.

On va en quelque sorte effectuer deux interpolations linéaires pour tenir compte de la position de x dans le segment $[x_1, x_2]$ et de celle de y dans le segment $[y_1, y_2]$ (ce qui donne son nom à l'algorithme…)

Question 4. Avec les notations précédentes, déterminer deux réels λ et μ tels que $x = \lambda x_2 + (1 - \lambda) x_1$ et $y = \mu y_2 + (1 - \mu) y_1$

Avec les valeurs trouvées de λ et de μ, si la luminance du point de coordonnées (x_i, y_i) est donnée par $L(x_i, y_i)$, on choisira pour luminance au point de coordonnées (x, y) :

$$\lambda (\mu L(x_2, y_2) + (1 - \mu) L(x_2, y_1)) + (1 - \lambda) (\mu L(x_1, y_2) + (1 - \mu) L(x_1, y_1))$$

ce qui peut aussi s'écrire :

$$\mu (\lambda L(x_2, y_2) + (1 - \lambda) L(x_1, y_2))) + (1 - \mu) (\lambda L(x_2, y_1) + (1 - \lambda) L(x_1, y_1))$$

Question 5. Expliquer la formule précédente !

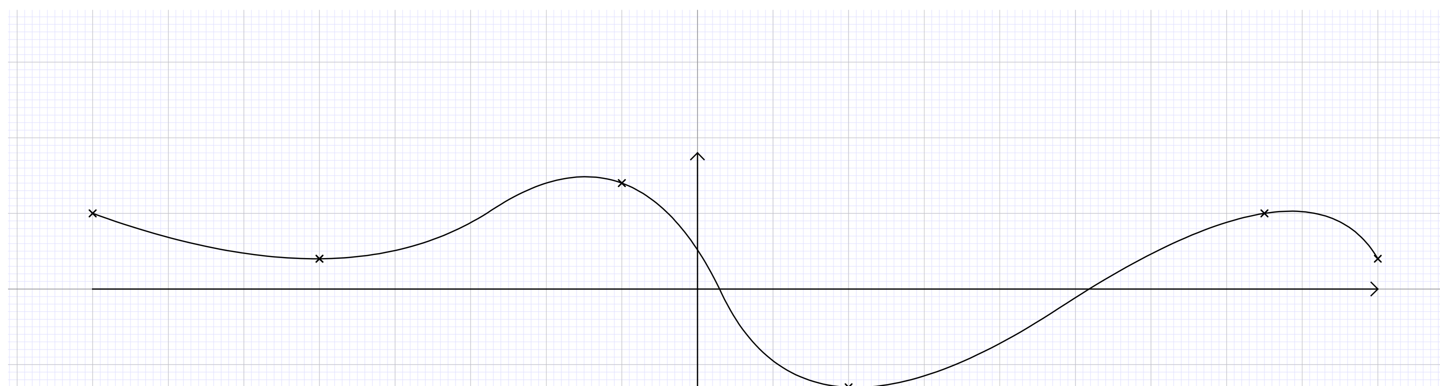
L'objectif n'est pas ici d'écrire une fonction qui réalise ce traitement. Rien de bien difficile au fond, mais les fautes de frappe sont très fréquentes et souvent on s'y reprend à plusieurs fois. Vous trouverez en annexe la fonction que j'ai écrite. On se contentera ici de comparer les différents traitements sur quelques images de test.

1.3 Algorithme bicubique

Si on revient au problème de l'interpolation d'une fonction à partir de quelques valeurs particulières, on peut souhaiter obtenir une fonction avec une plus grande régularité que la seule continuité. L'étape suivante cherche à obtenir une fonction f de classe $\mathcal{C}^1$ (dérivable et de dérivée continue), ce qui est en pratique obtenu dès lors que les restrictions à chaque intervalle $[x_i, x_{i+1}]$ de la fonction f reconstruite sont de classe $\mathcal{C}^1$ et que de plus, si $1 < i < n$ (pour n points initiaux), alors les dérivées de f en x_i à gauche et en x_i à droite sont égales. En pratique, on ne pourra plus considérer pour décider de la valeur de $f(x)$ pour x entre x_i et x_{i+1} que les seules valeurs y_i et y_{i+1} mais on devra tenir compte des points d'abscisses x_{i-1} et x_{i+2}.

Sur chaque segment $[x_i, x_{i+1}]$ la fonction f sera de la forme $t \mapsto a_i\, t^3 + b_i\, t^2 + c_i\, t + d_i$ (on peut parler de cubique) et les coefficients a_i, b_i, c_i, d_i sont choisis de telle sorte que $f(x_i) = y_i$, $f(x_{i+1}) = y_{i+1}$ et que f soit de classe $\mathcal{C}^1$.

Plusieurs méthodes sont envisageables, mais il est hors de portée de s'y attaquer aujourd'hui, d'autant que la mise en œuvre en python conduirait à des fonctions trop lentes pour être exploitables. En pratique, pour déterminer par un tel algorithme la couleur d'un pixel interpolé, il nous faudrait exploiter les couleurs des 16 pixels les plus proches du pixel à déterminer. Un point qui devrait sauter aux yeux sur l'exemple présenté ci-dessous est que les valeurs de la fonction f peuvent dépasser supérieurement ou inférieurement les valeurs initiales des points d'interpolation, et on va explorer quelques images transformées par ces différents algorithmes.



1.4 Quelques exemples

J'ai créé deux images tests : une croix (ou un plus...) sous la forme d'une image de 128×128 pixels en noir et blanc, codé sur 8 bits. Le fond clair (luminance 240) et la croix foncée (luminance 16), et un cercle, de même dimensions.

J'ai redimensionné ces deux images en des images de 512×512 pixels de trois manières :

- par l'algorithme faisant appel au plus proche voisin (vous devriez obtenir la même chose avec votre code)
- par l'algorithme bilinéaire présenté ci-dessus (voir l'annexe pour le code utilisé)
- à l'aide de l'application Aperçu sur mon ordinateur (MacOS) dont je ne connais pas l'algorithme utilisé (mais il ne fait guère de doute qu'une interpolation bicubique est employée)

Question 6. Etant donné un tableau numpy A à deux dimensions, écrire une fonction `def minEtMax(A):` qui explore toutes les valeurs de A et renvoie le couple $(\min, \max)$ formé de la plus petite et la plus grande valeur de A. Il va sans dire qu'on ne fera pas appel aux fonctions `min`, `max`, `np.min` ou `np.max`.

Question 7. Importer les fichiers `croix512bilineaire.png` et les fichiers `croix512bicubique.png` (à l'aide de `Image.open(...)`, vous ferez attention au fait que ces images sont en noir et blanc et ne comportent qu'une seule couche de luminance) et, à l'aide de la fonction précédente, indiquer quelles sont les plus petites et plus grandes luminances obtenues. Que constate-t-on ?

(Je vous invite aussi à utiliser l'explorateur de variables intégré à Spyder pour parcourir les luminances de ces deux images)

Remarque 1. J'ai réalisé l'expérience suivante : j'ai réduit au quart et agrandi 4 fois l'image du chat (en couleurs) par

- l'algorithme du plus proche voisin
- l'algorithme bilinéaire

- l'application Aperçu d'Apple (sans nul doute un algorithme bicubique)

Vous trouverez le résultat de cette expérience dans les trois fichiers `chatPlusProcheVoisin.png`, `chatBilineaire.png` et `chatBicubique.png`. Je pense que les images parlent d'elles-même !

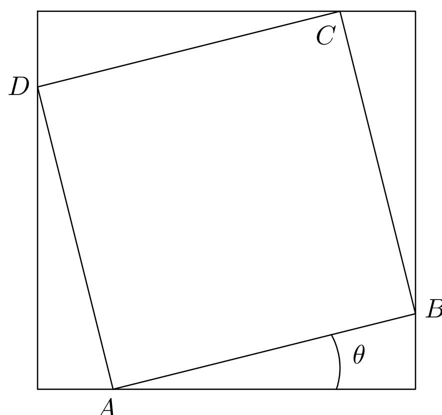
2 Rotation

Quelques rappels mathématiques : pour faire tourner un vecteur de coordonnées (x, y) selon un angle θ (le plan euclidien étant muni d'un repère orthonormé (O, e_1, e_2)) on peut :

- utiliser les affixes complexes : l'affixe du vecteur initial est $x + \mathrm{i}\, y$ et celui du vecteur obtenu après rotation est $e^{\mathrm{i}\theta}(x + \mathrm{i}\, y)$ ou encore $\cos\theta\, x - \sin\theta\, y + \mathrm{i}\,(\sin\theta\, x + \cos\theta\, y)$ (faites le calcul si nécessaire !)

- utiliser un calcul matriciel : les coordonnées du vecteur transformé s'obtiennent par le produit matriciel $\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \cos\theta\, x - \sin\theta\, y \\ \sin\theta\, x + \cos\theta\, y \end{pmatrix}$ (on pose souvent $R_\theta = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$ qu'on appelle matrice de rotation d'angle θ : on y reviendra en mathématiques l'an prochain)

On va supposer dans ce qui suit, pour simplifier, faire tourner une image d'un angle θ admettant une mesure strictement comprise entre 0 et 90 degrés, ce qui conduit à une image s'inscrivant dans un cadre agrandi (et ne remplissant pas complètement celui-ci) :



2.1 Détermination du cadre

On suppose que l'image initiale est formée de n lignes de p pixels, et on cherche à faire tourner celle-ci d'un angle $\theta\big(\in\big[0, \frac{\pi}{2}\big]\big)$

Question 8. Déterminer la largeur du cadre (la différence entre les abscisses des points B et D) et la hauteur de celui-ci (la différence entre les ordonnées de C et A)

Question 9. (*) Dans le cadre agrandi comptant n_2 lignes de p_2 pixels (bien sûr, les entiers n_2 et p_2 auront été obtenus en arrondissant les nombres calculés à la question précédente), d'un pixel situé à la ligne i et colonne j, calculer à quelle ligne et quelle colonne de l'image initiale ce pixel correspond (on peut bien sûr être en dehors ce cette image, auquel cas, on laissera ou bien noir, ou bien blanc la couleur de ce pixel) : ici on calcule des coordonnées flottantes, que bien sûr il faudra tôt ou tard convertir en des entiers.

Attention : dans une matrice ou un tableau numpy, les lignes vont en croissant de haut en bas, ce qui est bien sûr le sens contraire de ce à quoi on est habitué avec les repères cartésiens.

Vous vous en doutez, il en va des algorithmes de rotation d'image comme des algorithmes de redimensionnement d'images : en première approche, on peut se contenter d'un algorithme élémentaire comme celui du plus proche voisin (pour déterminer la couleur du pixel de l'image finale, on cherche le pixel le plus proche du point dont on vient de calculer les coordonnées, et c'est la couleur de ce seul pixel qui détermine la couleur du pixel souhaité), mais on pourrait choisir là encore un algorithme bilinéaire, bicubique et tenir compte des 4 ou 16 plus proches pixels de la position calculée...

Question 10. (*) Ecrire une fonction d'entête `def rotation(image, angle):` qui prend en argument un tableau numpy `image` à deux dimensions (image en noir et blanc) et un angle donné par sa mesure en degrés entre 0 et 90 degrés, et qui calcule et affiche par l'algorithme du plus proche voisin l'image donnée en paramètre ayant subi la rotation souhaitée.

3 Annexe

Ma fonction de redimensionnement par l'algorithme bilinéaire est la suivante :

```
def redimensionne2(image, rapport):
    if len(image.shape)==2: # image en noir et blanc, une seule luminance
        n, p = image.shape
        n2 = round(n*rapport)
        p2 = round(p*rapport)
        output = np.zeros((n2,p2),dtype=np.uint8)
    else: # image en couleurs
        n, p, c = image.shape
        n2 = round(n*rapport)
        p2 = round(p*rapport)
        output = np.zeros((n2,p2,c),dtype=np.uint8)

    for i in range(n2):
        for j in range(p2):
            x = int(j/rapport)
            y = int(i/rapport)
            x1 = min(x+1,p-1) # éviter les erreurs de type IndexError...
            y1 = min(y+1, n-1)
            la = j/rapport-x # lambda et mu
            mu = i/rapport-y
            output[i,j]=la*(mu*image[y1,x1]+(1-mu)*image[y,x1])+\
                        (1-la)*(mu*image[y1,x]+(1-mu)*image[y,x])
    return output
    # ou mpl.show(output)
```