# Redimensionnement intelligent d'image

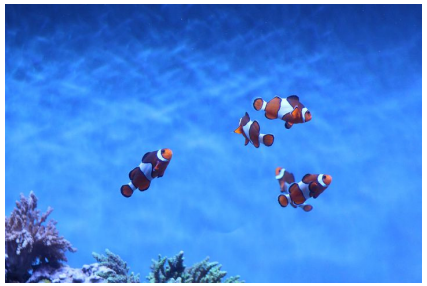
## 1 Préliminaires

### 1.1 Objectifs

Le premier but de cette séance de travaux pratique est d'écrire un algorithme permettant de redimensionner horizontalement des images de façon intelligente. Les images dont on dispose ont un aspect « 3:2 » (leur largeur est 1.5 fois plus grande que leur hauteur), et on souhaite obtenir des images carrées.

Comme nous le verrons, les algorithmes usuels, basés sur une homothétie, vont déformer les objets dans l'image. En particuler, des objets rond deviendront elliptiques, des objets carrés deviendront rectangulaires.

Il est cependant possible de procéder à des redimensionnements « intelligents » (dit « content-aware », attentifs au contenu) qui procèdent différemment, en comprimant davantage les zones peu intéressantes pour préserver la forme des zones les plus intéressantes, comme sur l'exemple ci-dessous (on remarquera en particulier que l'espace entre les amphiprioninae [1] a été réduit, il ne s'agit pas d'un simple recadrage), et c'est ce que nous allons essayer de mettre en œuvre aujourd'hui.



### 1.2 Récupération des données

Pour télécharger et installer les données que l'on va utiliser durant cette séance de TP, lancer l'application « cygwin terminal » dont le raccourci se trouve sur le bureau, et entrer soigneusement la commande suivante [2] (que l'on validera avec la touche entrée) :

```
cd /cygdrive/d; curl cdn.sci-phy.org/psi/tp2-carving.tgz | tar xvz
```

La commande devrait avoir créé un dossier `TP2-Carving` dans la racine du disque `D:` (ou `F:`). Vous pouvez ensuite refermer le terminal et lancer l'environnement Pyzo.

1. Vous avez toujours voulu connaître le nom scientifique du poisson-clown.
2. Si le disque de « données » sur la machine est `F:`, on écrira « `/cygdrive/f` ».

En cas de difficultés, ou si vous voulez travailler sur votre propre machine, vous pouvez télécharger le fichier `http://cdn.sci-phy.org/psi/TP2-Carving.tgz` par la méthode de votre choix, de décompresser le fichier ainsi obtenu (par exemple avec l'utilitaire `7zip`) et de placer le répertoire à un endroit aisément accessible.

### 1.3 Chargement des données

Le répertoire ainsi installé contient différentes images (en couleur, de petite taille [3]), nommées `bulle.png`, `horloge.png`, `lac.png`... Pour charger une de ces images, on peut utiliser le morceau de code suivant (on remplacera éventuellement `D:` par `F:`, en fonction de la configuration de la machine) :

```
from imageio import imread
img = [[[c/256.0 for c in pxl] for pxl in line]
            for line in imread("D:/TP2-Carving/bulle.png")]
```

Le résultat est une liste [4] correspondant aux différentes lignes de l'image, chaque ligne étant représentée par une liste des pixels de la ligne, chaque pixel étant lui-même représenté par une liste de trois entiers entre 0.0 et 1.0, correspondant à l'intensité dans chacune des trois couleurs (rouge, vert et bleu).

Pour accéder à la couleur rouge du pixel situé sur la ligne `42` et la colonne `78`, on écrira par exemple `img[42][78][0]`. Pour la couleur verte du pixel situé ligne `37`, colonne `80`, on écrira `img[37][80][1]`. Rappelons également que `len(img)` permettra d'obtenir le nombre de lignes dans l'image, et `len(img[0])` son nombre de colonnes.

Dans la suite du sujet, par commodité, on notera H la hauteur de l'image sur laquelle on travaille (son nombre de lignes) et L sa largeur (son nombre de colonnes).

Pour afficher une image couleur nommée `img`, on peut utiliser la commande suivante :

```
import matplotlib.pyplot as plt
plt.imshow(img)
plt.show()
```

**1.** S'assurer que l'on parvient à afficher correctement quelques-unes des images du répertoire téléchargé.

3. Par commodité car les algorithmes étudiés ici sont relativement gourmands, mais on peut travailler avec de plus grandes images sans problème particulier.
4. En général, on manipule plutôt les images comme des tableaux `numpy`, mais on utilisera des listes de listes de listes ici pour s'adapter au programme officiel, ce qui simplifiera par ailleurs quelques aspects de l'algorithme étudié.

# 2 Premières fonctions

## 2.1 Conversion en niveau de gris

La plupart des algorithmes que l'on va écrire sur les images nécessitent d'avoir une représentation de l'image en niveau de gris. Dans le cas d'une image en niveau de gris, chaque pixel est simplement décrit par la donnée d'un unique flottant, représentant une *intensité* lumineuse. Une image en niveau de gris sera ici représentée comme une liste de listes de flottants entre 0 et 1 correspondant à ces intensités, 0 correspondant à du noir et 1 à du blanc.

Pour convertir un pixel dont les composantes de couleur sont `[r, v, b]`, on utilisera l'intensité suivante [5] :

$$\texttt{i} = 0{,}3 \times \texttt{r} + 0{,}6 \times \texttt{v} + 0{,}1 \times \texttt{b}$$

**2.** Justifier que cette intensité `i` est bien comprise entre 0.0 et 1?0 quelle que soit la couleur du pixel.

**3.** Proposer une fonction `convertir_gris(img)` prenant en argument une image (qu'elle ne modifiera pas) et retournant une image en niveau de gris (une liste de `H` listes de `L` flottants).

Pour afficher une image appelée `img_g` en niveaux de gris, on peut utiliser la commande suivante :

```
plt.imshow(img_g, cmap="gray", vmin=0.0, vmax=1.0)
plt.show()
```

**4.** Afficher une image du répertoire convertie en niveau de gris à l'aide de la fonction précédente.

## 2.2 Détermination des zones contenant des détails

Dans les images, certaines zones sont plus intéressantes que d'autres. On considérera que les contours des objets et les zones fortement texturées sont plus intéressantes que les zones unies. Pour déterminer les points intéressants d'une image, on peut donc s'intéresser à la norme du gradient de l'intensité lumineuse en chaque point de l'image.

Cette norme, pour un point `(i, j)` d'une image `img`, sera calculé de la façon suivante :

$$n = \sqrt{\left(\texttt{img[i][j]} - \texttt{img[i-1][j]}\right)^2 + \left(\texttt{img[i][j]} - \texttt{img[i][j-1]}\right)^2}$$

Bien évidemment, cette définition pose un problème pour les pixels de la première ligne et de la première colonne. Si `img[i-1][j]` se trouve hors de l'image, on utilisera `img[i+1][j]`. De même pour `img[i][j-1]` qui sera remplacé par `img[i][j+1]`. Le lecteur avisé remarquera qu'il est inutile de faire des tests sur les valeurs de `i` et `j` et qu'il suffit d'utiliser `img[abs(i-1)][j]` et `img[i][abs(j-1)]` dans la formule, puisque les seuls cas problématiques concernent `i=0` et `j=0`!

**5.** Proposer une fonction `zones_interet(img_g)` prenant en argument une image en niveaux de gris (qu'elle ne modifiera pas) et retournant une liste de `H` listes de `L` flottants, correspondant à la norme du gradient en tout point de l'image, calculé avec la fonction précédente.

Le résultat, une liste de listes de flottants qu'on appelera par exemple `img_interet`, ressemble par sa structure à une image en niveau de gris, même si les valeurs ne sont pas nécessairement comprises entre 0 et 1. On peut cependant l'afficher en écrivant simplement (les endroits où le gradient est le plus élevé, c'est-à-dire les zones probablement les plus intéressantes, apparaissent en vert) :

```
plt.imshow(img_interet)
plt.show()
```

**6.** Afficher les zones intéressantes de quelques images du répertoire, calculées à partir de la fonction précédente (l'image doit être convertie en noir et blanc avant le calcul des zones d'intérêt).

# 3 Réduction d'une image

## 3.1 Principe et suppression de points

Pour réduire la taille horizontale d'une image d'un pixel, une solution simple consiste à retirer un pixel de chacune des lignes [6]. Le tout est de bien choisir ces points.

On fournit une fonction `retire(img, cols)` qui prend en argument une image (couleur ou noir et blanc) de hauteur `H`, et une liste de `H` entiers entre 0 et `L-1` (inclus), et qui, pour chaque ligne `i`, supprime le pixel d'abscisse `cols[i]`. Attention, cette fonction **modifie** l'image passée en argument.

```
def retire(img, cols):
    for line, j in zip(img, cols):
        line.pop(j)
```

5. Les coefficients différents pour le rouge, le vert et le bleu ont été choisis pour correspondre à la réponse différente de l'oeil aux différentes couleurs, mais d'autres choix peuvent être faits.

6. Rappelons que l'on essaie de ne *pas* effectuer une homothétie, mais de supprimer les points de l'image peu intéressants!

## 3.2 Solutions naïves

Une première solution pour redimensionner l'image est de la recadrer. Pour réduire d'un tiers la largeur de l'image, on peut simplement retirer les `L//6` colonnes de gauche et les `L//6` colonnes de droite.

**7.** Proposer une fonction `recadre(img)` qui modifie l'image passée en argument (elle ne retourne rien) et la recadre. On utilisera la fonction `retire` autant de fois que nécessaire, en construisant des listes `cols` adaptées.

**8.** Tester la fonction `recadre(img)` sur des images du répertoire téléchargé.

Pour faciliter la comparaison entre images originales et images retravaillées, le programme dans son ensemble pourra avoir cette structure, afin d'afficher côte à côte l'image originale et l'image retravaillée :

```
from imageio import imread
from copy import deepcopy
import matplotlib.pyplot as plt

# (on placera ici toutes les fonctions qu'il est demandé d'écrire)
def recadre(img):
    ...

img = [[[c/256.0 for c in pxl] for pxl in line]
            for line in imread("D:/TP2-Carving/bulle.png")]
orig = deepcopy(img) # Pour garder une copie de l'image originale

recadre(img) # <- à changer selon la question

plt.figure(figsize=(8,3))
plt.subplot(121)
plt.imshow(orig)
plt.subplot(122)
plt.imshow(img)
plt.show()
```

Une autre solution consiste à répartir les colonnes à supprimer sur toute l'image (cela reviendra quasiment à obtenir une homothétie).

**9.** Proposer une fonction `redimensionne(img)` qui modifie l'image passée en argument (elle ne retourne rien) et supprime une colonne sur trois dans l'image, les colonnes étant régulièrement espacées. Là encore, on utilisera la fonction `retire` autant de fois que nécessaire, en construisant des listes `cols` adaptées.

**10.** Tester la fonction `redimensionne(img)` sur des images du répertoire téléchargé.

## 3.3 Suppression des points les moins intéressants

Nous allons à présent utiliser les mesures d'intérêt de chaque pixel obtenus avec la fonction `zones_interet`. Pour réduire la taille horizontale d'un pixel, une première solution consiste, pour chaque ligne, à retirer le pixel le moins intéressant.

**11.** Proposer une fonction `imin(lst)` prenant en argument une liste de flottants et retournant la position du plus petit flottant de la liste (en cas d'égalité, on pourra choisir librement entre les différents plus petits flottants identifiés).

**12.** En déduire une fonction `cols_1(img_interet)` qui, à partir d'une liste de listes contenant les normes des gradients, construit une liste de taille `H` d'index de colonne (entre `0` et `L-1`) identifiant les pixels les moins intéressants de chaque ligne.

| | | | | | | |
|---|---|---|---|---|---|---|
| 0.6 | 0.4 | 0.2 | 0.5 | 0.3 | 0.2 | 0.1 |
| 0.7 | 0.3 | 0.2 | 0.3 | 0.4 | 0.3 | 0.5 |
| 0.7 | 0.3 | 0.4 | 0.2 | 0.3 | 0.2 | 0.6 |
| 0.6 | 0.3 | 0.5 | 0.3 | 0.1 | 0.6 | 0.5 |
| 0.2 | 0.4 | 0.5 | 0.3 | 0.4 | 0.7 | 0.5 |

`img_interet`

$\longmapsto$ `[6, 2, 3, 4, 0]`

**13.** Grâce à la fonction précédente, proposer une fonction `transforme_1(img)` prenant en argument une image et ne retournant rien, mais modifiant l'image en retirant les `L//3` pixels les moins intéressants de chaque ligne. On effectuera `L//3` opérations de retrait d'un pixel sur chacune des lignes. Par simplicité, on recalculera la carte des zones d'intérêt `img_interet` après chaque retrait[7].

**14.** Tester la fonction sur quelques images du répertoire, et constater que les déformations dans l'image obtenue rendent le résultat peu satisfaisant.

## 3.4 Suppression des colonnes les moins intéressantes

Pour obtenir un meilleur résultat, on peut vouloir supprimer des colonnes entières de points, plutôt que des points situés dans des colonnes différentes sur chaque ligne.

| | | | | | | |
|---|---|---|---|---|---|---|
| 0.6 | 0.4 | 0.2 | 0.5 | 0.3 | 0.2 | 0.1 |
| 0.7 | 0.3 | 0.2 | 0.3 | 0.4 | 0.3 | 0.5 |
| 0.7 | 0.3 | 0.4 | 0.2 | 0.3 | 0.2 | 0.6 |
| 0.6 | 0.3 | 0.5 | 0.3 | 0.1 | 0.6 | 0.5 |
| 0.2 | 0.4 | 0.5 | 0.3 | 0.4 | 0.7 | 0.5 |

`img_interet`

$\longmapsto$ `[4, 4, 4, 4, 4]`

7. L'image en noir et blanc peut, au choix, être recalculée à chaque étape, ou être tenue à jour grâce à la fonction `retire`.

**15.** Proposer une fonction `transforme_2(img)` qui, `L//3` fois, détermine la colonne la moins intéressante (identifiée comme celle dont la somme des normes des gradients de ses pixels est la plus faible) et retire les points de ladite colonne.

**16.** Tester la fonction sur quelques images du répertoire, et vérifier que si les résultats sont généralement satisfaisants, on voit apparaître des artefacts dans les images produites.

### 3.5 Algorithme de « *seam carving* »

Nous en arrivons à l'algorithme de *seam carving* permettant d'effectuer le suppression de façon un peu plus intelligente, en choisissant plus soigneusement les pixels à éliminer à chaque itération : plutôt qu'une ligne verticale de points à supprimer, on cherche à éliminer un ensemble de `H` pixels (un par ligne) voisins mais pas nécessairement dans la même colonne (deux pixels sur deux lignes consécutives pouvant appartenir à des colonnes voisines, se touchant donc « *en diagonale* »). La somme des gradients des points de ce chemin doit être la plus faible possible, telle que sur l'exemple ci-dessous :

| | | | | | | |
|---|---|---|---|---|---|---|
| 0.6 | 0.4 | 0.2 | 0.5 | 0.3 | 0.2 | 0.1 |
| 0.7 | 0.3 | 0.2 | 0.3 | 0.4 | 0.3 | 0.5 |
| 0.7 | 0.3 | 0.4 | 0.2 | 0.3 | 0.2 | 0.6 |
| 0.6 | 0.3 | 0.5 | 0.3 | 0.1 | 0.6 | 0.5 |
| 0.2 | 0.4 | 0.5 | 0.3 | 0.4 | 0.7 | 0.5 |

`img_interet`

$\longmapsto$ `[2, 2, 3, 4, 3]`

Il y a trop de chemins possibles, menant du haut de l'image au bas de l'image et allant de proche en proche, pour une approche naïve. Nous allons donc utiliser une solution basée sur la programmation dynamique. À partir de `img_interet`, contenant les normes des gradients, on va construire, ligne par ligne, une liste de listes `interets_cummules` de la façon suivante :

- la première ligne est une copie de la première ligne de `img_interet` ;
- pour les lignes suivantes, la valeur située ligne `i` colonne `j` de `interets_cummules` correspond à la valeur ligne `i` colonne `j` de `img_interet` à laquelle on ajoute la plus petite valeur parmi celles rangées dans la ligne `i-1` et les colonnes `j-1` à `j+1` (si elles existent) de `interets_cumules`.

| | | | | | | |
|---|---|---|---|---|---|---|
| 0.6 | 0.4 | 0.2 | 0.5 | 0.3 | 0.2 | 0.1 |
| 0.7 | 0.3 | 0.2 | 0.3 | 0.4 | 0.3 | 0.5 |
| 0.7 | 0.3 | 0.4 | 0.2 | 0.3 | 0.2 | 0.6 |
| 0.6 | 0.3 | 0.5 | 0.3 | 0.1 | 0.6 | 0.5 |
| 0.2 | 0.4 | 0.5 | 0.3 | 0.4 | 0.7 | 0.5 |

`img_interet`

| | | | | | | |
|---|---|---|---|---|---|---|
| 0.6 | 0.4 | 0.2 | 0.5 | 0.3 | 0.2 | 0.1 |
| 1.1 | 0.5 | 0.4 | 0.5 | 0.6 | 0.5 | 0.6 |
| 1.2 | 0.7 | 0.8 | 0.6 | 0.8 | 0.7 | 1.1 |
| 1.3 | 1.0 | 1.1 | 0.9 | 0.7 | 1.3 | 1.2 |
| 1.2 | 1.4 | 1.4 | 1.0 | 1.1 | 1.4 | 1.7 |

`interets_cumules`

**17.** Proposer une fonction `cumule(img_interet)` qui prend en argument une liste de listes de flottants contenant les normes des gradients et retournant une liste de listes de flottants contenant les sommes cumulées telles que décrites précédemment.

**18.** Justifier que le chemin continu (en 8-connexité) entre un pixel de la première ligne et un pixel de la dernière ligne et dont la somme des normes des gradients est minimal parmi tous les chemins de ce type se termine nécessairement sur un pixel correspondant, sur la dernière lignes de `interet_cumule`, à une valeur minimale.

**19.** Afficher (avec `plt.imshow`) le résultat obtenu pour une image d'exemple, et vérifier que les zones les moins intéressantes sont bien apparentes.

**20.** Comment reconstruire le chemin optimal à partir de `interet_cumule` ? Proposer une fonction `opt_cols(interet_cumule)` retournant une liste de `H` entiers représentant les index de colonne de chaque pixel dans le chemin optimal (comme précédemment, un index de colonne par ligne, chaque ligne étant considérée de haut en bas).

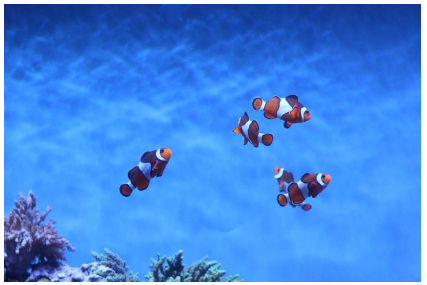
**21.** Proposer une fonction `carve(img)` qui prend en argument une image, et la modifie en retirant, `L//3` fois de suite, le chemin le moins intéressant.

**22.** Tester la fonction précédente sur quelques-unes des images proposées, et discuter du résultat obtenu. Pourquoi la fonction est-elle relativement lente ? Quelle est sa complexité ?

## 4 renforcement d'une image

### 4.1 Principe

L'algorithme de *seam carving* permet également de « renforcer » une image sans en changer les proportions. Pour ce faire, on l'utilise dans les deux directions, en retirant $\lfloor \alpha H \rfloor$ lignes et $\lfloor \alpha L \rfloor$ colonnes, $\alpha$ étant un paramètre réel choisi entre 0 et 1. On peut ensuite éventuellement effectuer une homothétie pour rétablir la taille initiale de l'image. Le résultat sera par exemple similaire à l'exemple ci-dessous, où $\alpha = \frac{1}{3}$ :



### 4.2 Implémentation

**23.** Plutôt que de repartir de zéro, nous allons exploiter ce qui a été fait précédemment. Proposer une fonction `transpose(img)` qui prend en argument une liste de listes cor-

respondant à une image et retourne une nouvelle liste de listes correspondant à l'image transposée (où les lignes sont devenues des colonnes et vice-versa).

**24.** Utiliser les fonctions `transpose` et `carve` pour obtenir une fonction `renforce(img)` prenant en argument une liste de listes représentant une image et retournant une liste de listes correspondant à l'image renforcée avec $\alpha = \frac{1}{3}$. Pour simplifier l'implémentation, on tolère ici que l'image passée en argument soit modifiée.