

Programmation impérative, projet 2024

Date et principe

Cette page peut être mise à jour, avec informations complémentaires, précisions, questions bonus, etc. Pensez à y revenir souvent.

Projet à rendre pour le **06/01/2025 à 23h59**, aucun retard ne sera toléré.

Lire tout le sujet.

Un rendu de projet comprend :

- Un rapport typographié précisant vos choix, les problèmes techniques qui se posent et les solutions trouvées ; il présente en introduction le contexte et le sujet du projet, et il précise en conclusion les limites de votre programme. Le rapport sera de préférence composé avec L^AT_EX. Le soin apporté à la grammaire et à l'orthographe est largement pris en compte.
- Un code abondamment commenté ; la première partie des commentaires comportera systématiquement les lignes :
 1. @requires décrivant les préconditions : c'est-à-dire conditions sur les paramètres pour une bonne utilisation (pas de typage ici),
 2. @assigns listant les zones de mémoire modifiées,
 3. @ensures décrivant la propriété vraie à la sortie de la fonction lorsque les préconditions sont respectées, le cas échéant avec mention des comportements en cas de succès et en cas d'échec, En outre chaque boucle while doit contenir un commentaire précisant la raison de sa terminaison (le cas échéant). De même en cas d'appels récursifs. On pourra préciser des informations additionnelles si des techniques particulières méritent d'être mentionnées.

Le code doit enfin compiler sans erreur (évidemment) et sans warning lorsque l'option -Wall est utilisée. **Un code qui ne compile pas se verra attribuer la note de 0.**

- Un manuel d'utilisation de votre exécutable, même minimal, est toujours bienvenu.

Avez-vous lu tout le sujet ?

Protocole de dépôt

Vous devez rendre

- votre rapport (en pdf) et
- vos fichiers de code

rassemblés dans une archive tar gziippée identifiée comme `votre_prénom_votre_nom.tgz`. La commande devrait ressembler à : `tar zcvf randolph_carter.tgz rapport.pdf fichiers.c autres_trucs_éventuels.c...`

N'OUBLIEZ surtout PAS de mettre le nom identifiant l'archive (donc nouveau) en PREMIER.

Lisez le man ! et testez le contenu de votre archive (une commande comme par exemple : `tar tvf randolph_carter.tgz` doit lister les fichiers et donner leur taille).

- Une archive qui ne contient pas les fichiers demandés ne sera pas excusable.
- **Une archive qui n'est pas au format demandé (archive tar gziippée avec suffixe .tgz) ne sera pas corrigée** donc c'est 0/20.

Toute tentative de fraude (plagiat, etc.) sera sanctionnée. Si plusieurs projets ont des sources trop similaires (y compris sur une partie du code uniquement), tous leurs auteurs se verront attribuer la note 0/20. En particulier, il faudra prendre soin de ne pas publier son travail sur un dépôt public (en tout cas pas avant la date de fin de rendu). On évitera également de demander (ou de donner) des conseils trop précis à ses camarades (y compris des promotions précédentes), ces conseils ayant pu être donnés à plusieurs personnes. Les rendus seront comparés deux à deux.

De même, l'usage d'intelligence artificielle générative pour produire le code et/ou le rapport est strictement interdite.

Procédure de dépôt

Vous devez enregistrer votre archive tgz dans le dépôt dédié au cours PRIM11 (prim11-projet-2024) en vous connectant à exam.ensiie.fr. Ce dépôt sera ouvert jusqu'au 6 janvier 2025 inclus.

Contexte

Le but de ce projet est d'implémenter différents traitements sur des images en niveau de gris ou des images en couleur comme lors du TP 6.

Pour réaliser ces traitement vous allez écrire un programme permettant de :

- Lire un ou plusieurs fichiers images (au format [Portable Pixmap](#) et plus particulièrement le format PPM binaire pour les images couleurs (vu en cours) et PGM binaire pour les images en niveau de gris (du TP6)).
- Puis de réaliser différents traitements sur les images lues.
- Et enfin écrire les images résultantes dans de nouveaux fichiers.
 - Par exemple si vous lisez l'image `images/Lenna_color.ppm` pour lui appliquer un traitement qui consistera à inverser les niveaux des pixels de l'image vous pourrez sauvegarder le résultat dans un fichier `images/Lenna_color_inv.ppm`.

Structures de données et modules

- Une image est constituée d'une matrice de pixels (Picture Elements).
 - Pour une image en niveaux de gris chaque pixel est représenté par une valeur (généralement codée sur 1 octet, donc à valeurs dans l'intervalle [0…255])
 - Pour une image en couleur chaque pixel est représenté par un triplet d'octets (un pour chaque composante : *rouge*, *vert* et *bleu*).
 - Même si une image est censée être une “matrice” de pixels, on utilise la plupart du temps un tableau monodimensionnel pour stocker en mémoire les valeurs de ces pixels. On ne réalisera ainsi qu'une seule allocation / dés-allocation pour créer / détruire les pixels en mémoire.

Il serait utile de créer un module `pictures.[h|c]` contenant les structures de données dont vous aurez besoin ainsi que le fonctions travaillant directement sur des images:

Vous pourrez donc dans ce module déclarer :

- Un type `byte` par exemple pour contenir les valeurs codées par un octet. On pourra aussi définir une constante `MAX_BYTE` 255 pour contenir la valeur max des octets.
- Une structure `picture` similaire à celle utilisée dans le TP 6 qui contiendra
 - Une hauteur, une largeur
 - Le nombre de canaux de l'image
 - 1 canal pour les images en niveau de gris.
 - 3 canaux pour les images en couleur correspondant aux valeurs pour le rouge, le vert et le bleu.
 - Un pointeur vers les données pixels de l'image utilisant un tableau unidimensionnel d'octets.

- Pour accéder à la composante $k \in [0 \dots c - 1]$ d'un pixel situé à la ligne $i \in [0 \dots h - 1]$ et à la colonne $j \in [0 \dots w - 1]$ d'une image couleur p on pourra donc utiliser $\dots \text{data}[((i * w + j) * c) + k]$.
- Vous pourrez isoler les opérations liées à la **lecture** ou à l'**écriture** d'une composante de pixel (i, j, c) dans une image dans un sous module `pixels.[h/c]` dans lequel pour pourrez aussi avantageusement définir des constantes symboliques (en utilisant un `enum` par exemple) `RED`, `GREEN` et `BLUE` pour accéder directement à ces composantes dans les pixels.
- On vous fournit un module `filename.[h/c]` qui vous permettra de
 - Séparer un chemin vers un fichier image `<dirname>/<name>.ext` en ses composantes `<dirname>`, `<name>` et `<ext>`
 - Composer un chemin vers un fichier image à partir de ces composantes.
 - Vous pourrez ainsi extraire le nom du fichier image traité par votre programme et le customiser avec les noms des opérations que vous appliquerez dessus avant

Lecture et écriture de fichiers images

La structure des fichiers binaires PGM et/ou PPM est toujours la même

```
P6
512 512
255
...
```

- La première ligne contient un “magic number” correspondant au type de fichier (Ici `P6` identifie un fichier PPM binaire contenant une image couleur (ce serait `P5` pour les fichiers PGM binaires contenant des images en niveaux de gris)).
- La seconde ligne contient les dimensions de l'image : `512 512` correspondent ici à la largeur et la hauteur de l'image respectivement.
- La troisième ligne contient la valeur maximale des pixels à lire dans ce qui suit ... : Ici `255`.
 - Cette valeur maximale peut varier entre 1 et 255.
 - Si cette valeur était 64 cela voudrait dire qu'un pixel composé des valeurs $(r = 64, v = 64, b = 64)$ correspondrait à la couleur blanche, alors que si le maximum est de 255 un pixel $(r = 64, v = 64, b = 64)$ correspond à du gris foncé.
 - Pour simplifier les choses nous ne prendrons pas en compte cette valeur maximale dans nos structures de données (ce qui revient implicitement à considérer une valeur maximale de 255 dans nos images). Néanmoins, dans ce cas il faudra la prendre en compte lors de la lecture des fichiers et corriger les valeurs des composantes des pixels (par $\frac{255}{max}$) pour qu'elles soient comprises dans l'intervalle $[0 \dots 255]$ au lieu de la valeur maximale indiquée dans la 3ème ligne du fichier.
- **Après** la première ligne et **avant** la quatrième ligne il peut y avoir un nombre indéterminé de lignes commençant par le caractère `#` que l'on doit considérer comme des lignes de commentaires et donc ignorer.
- La quatrième ligne ... ou tout ce qui suit correspond aux données binaires des pixels. Dans le cas présenté ci-dessus il y a donc $512 \times 512 \times 3 = 786432$ octets à lire avant la fin du fichier.

Implémentez les fonctions pour :

- Lire des fichiers pour créer des images : `picture read_picture(char * filename);`
 - `[in] filename` le nom du fichier à lire
 - `[out]` l'image lue
 - Où `filename` pourra être :
 - `*.pgm` pour lire des images en niveau de gris.
 - `*.ppm` pour lire des images en couleur (vu en cours).
- Écrire des images dans des fichiers : `int write_picture(picture p, char * filename);`
 - `[in] p` l'image à écrire.
 - `[in] filename` le nom du fichier à écrire.
 - `[out]` valeur de retour nulle si l'écriture s'est bien passée et non nulle si un problème est survenu.

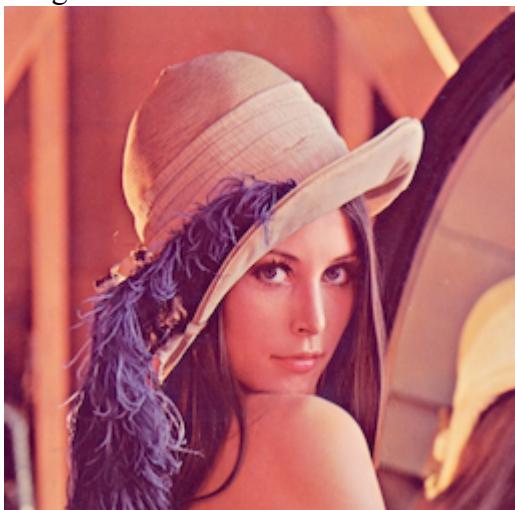
- Par exemple : Si le type de l'image ne correspond pas au type de fichier demandé (d'après l'extension du fichier contenue dans `filename`) on renverra un code d'erreur non nul.

Dans les deux cas, on fera attention à bien traiter les différents cas d'erreur (ouverture du fichier, lecture des données de l'en-tête, allocation mémoire, etc.).

Quelques images de test vous sont fournies :



- [Lenna_gray.pgm](#) : une version en niveaux de gris de la célèbre image “Lenna”.



- [Lenna_color.ppm](#) : Une version en couleurs



- [Lenna_BW.pgm](#) : Une version de noir et blanc

Gestion des images

Implémentez des fonctions pour créer, détruire, copier, interroger et convertir des images :

- **Créer une image** : `picture create_picture(unsigned int width, unsigned int height, unsigned int channels, byte max);` où
 - *[in]* *width* représente la largeur de l'image,
 - *[in]* *height* représente sa hauteur,
 - *[in]* *channels* le nombre de canaux et
 - *[in]* *max* la valeur max pour chaque valeur des pixels
 - *[out]* l'image initialisée
- **Nettoyer les données d'une image** : `void clean_picture(picture * p);`
 - *[in, out]* *picture* l'image à nettoyer
 - Les données de l'image ont été libérées et ses champs remis à 0 ou NULL.
- **Copie d'une image** : `picture copy_picture(picture p)`
 - *[in]* *p* l'image à copier
 - *[out]* une copie de l'image
- Obtention d'informations sur une image
 - Indication d'**image vide** (si un de ses champs est nul) : `int is_empty_picture(picture p);`
 - *[in]* *p* l'image à inspecter
 - *[out]* une valeur non nulle si *p* est vide, 0 sinon.
 - Indication d'**image en niveaux de gris** : `int is_gray_picture(picture p);`
 - *[in]* *p* l'image à inspecter
 - *[out]* une valeur non nulle si *p* est une image en niveaux de gris (ne possédant qu'un seul canal), 0 sinon.
 - Indication d'**image en couleurs** : `int is_color_picture(picture p);`
 - *[in]* *p* l'image à inspecter
 - *[out]* une valeur non nulle si *p* est une image en couleurs (possédant 3 canaux), 0 sinon.
 - **Affichage des infos** d'une image sur la console : `void info_picture(picture p);`
 - *[in]* *p* l'image dont on veut afficher les infos.
 - Affiche la chaîne : “(<width> x <height> x <channels>)” où <width>, <height> et <channels> sont à remplacer par leur valeurs respectives.
- **Conversion** d'un format à un autre :
 - Convertir une image en niveau de gris vers une image en couleur : `picture convert_to_color_picture(picture p);`
 - en répétant les valeurs de niveau de gris dans chaque canal R, V, B.
 - *[in]* *p* l'image à convertir en couleurs
 - *[out]* l'image couleur convertie en couleurs.
 - Si *p* était déjà en couleur on se contentera de faire une copie
 - Si *p* était une image en niveaux de gris on répétera la composante de niveau de gris dans chacune des composantes (rouge, vert, bleu) de l'image résultat.
 - Convertir une image en couleur vers une image en niveaux de gris : `picture convert_to_gray_picture(picture p);`
 - *[in]* *p* l'image à convertir en niveaux de gris
 - *[out]* l'image convertie en niveaux de gris
 - Si *p* était un image en couleur on a convertie les couleurs en niveau de gris en utilisant la règle suivante : $G = (0.299 \times R) + (0.587 \times V) + (0.114 \times B)$.
 - Si *p* était déjà une image en niveau de gris on se contentera d'en faire une copie.
- **Séparation ou mélange** des composantes d'une image
 - **Séparer les composantes** d'une image couleur en 3 images en niveau de gris contenant les valeurs pour le rouge, le vert et le bleu respectivement : `picture * split_picture(picture p);`
 - *[in]* *p* l'image couleur dont on veut séparer les composantes
 - *[out]* un tableau de 3 images en niveau de gris contenant les valeurs des canaux R, V et B.
 - Si *p* ne peut pas être décomposée on se contentera de renvoyer NULL.
 - Si *p* est une image en niveaux de gris on renverra un tableau ne contenant qu'un seul élément.
 - **Mélanger les composantes** à partir de 3 images en niveau de gris pour composer une image couleurs : `picture merge_picture(picture red, picture green, picture blue);`
 - *[in]* *red* l'image en niveau de gris à utiliser pour fabriquer la composante rouge de l'image résultat.
 - *[in]* *green* l'image en niveau de gris à utiliser pour fabriquer la composante verte de l'image résultat.

- *[in]* blue l'image en niveau de gris à utiliser pour fabriquer la composante bleue de l'image résultat.
- *[out]* l'image composée
 - Si l'image résultat ne peut pas être créée (si par exemple les trois images red, green et blue ne sont pas de même taille ou type) on se contentera de renvoyer une image vide.



o

RVB →



Red,



Green,



Blue,

Manipulation directe des valeurs des pixels

Ecrivez les fonctions pour :

- **Eclaircissement d'une image** : picture brighten_picture(picture p, double factor);
 - *[in]* p l'image à éclaircir
 - *[in]* factor le facteur à appliquer aux valeurs des pixels de l'image
 - *[out]* l'image éclaircie



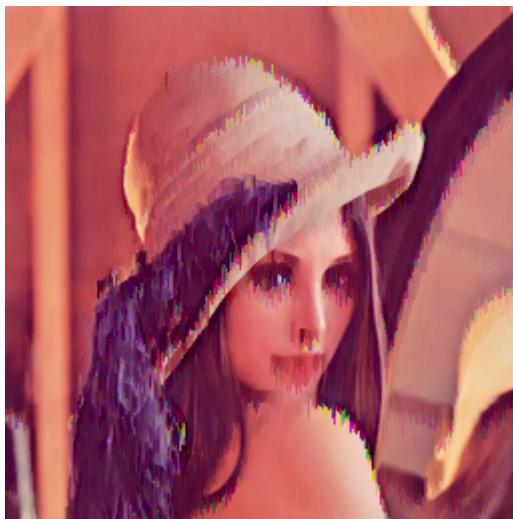
- o `picture melt_picture(picture p, int number);`, *factor* = 1.5 →



- **Fondre (vers le bas) des valeurs des pixels d'une image :** `picture melt_picture(picture p, int number);`
 - o On choisit N pixels au hasard dans l'image. Pour chacun de ces pixels, si le pixel situé juste au dessus est plus sombre que le pixel choisi alors le pixel courant prend la valeur du pixel du dessus.
 - o *[in]* *p* l'image à faire fondre
 - o *[in]* *number* le nombre de pixels à choisir aléatoirement
 - o *[out]* l'image contenant les pixels qui ont fondu vers le bas

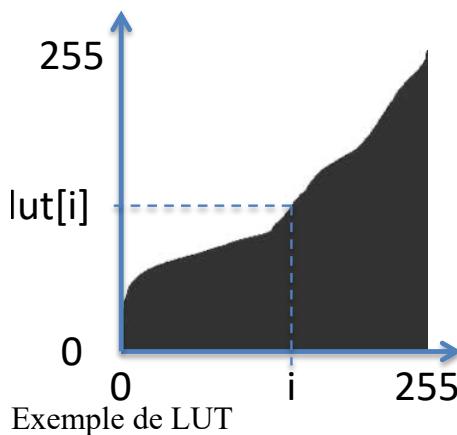


o $\text{number} = \text{width} \times \text{height} \times \text{channels} \times 5 \rightarrow$



Manipulation des valeurs des pixels en utilisant une LUT (Look Up Table)

Une LUT (pour Look Up Table, aussi appelée “fonction de transfert”) est une fonction qui à chaque niveau i d’une composante d’un pixel dans $[0 \dots 255]$ fait correspondre un autre niveau: $j = lut(i)$, avec la plupart du temps $i \in [0 \dots 255], j \in [0 \dots 255]$ mais pas obligatoirement.



Vous pourrez avantageusement créer un sous-module `lut.[h|c]` qui contiendra :

- La définition d'un type abstrait `lut` constitué :
 - d'une taille n indiquant le nombre de valeurs de la LUT.
 - La plupart du temps cette taille sera 256 pour avoir un index $i \in [0 \dots 255]$. Mais si l'on choisit une LUT avec une taille bien plus petite on pourra créer des fonctions de transfert possédant des marches d'escalier qui permettront de grouper plusieurs indices i pour une

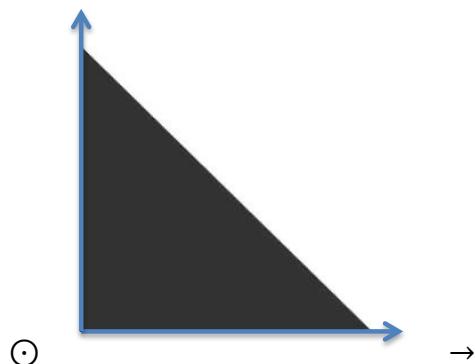
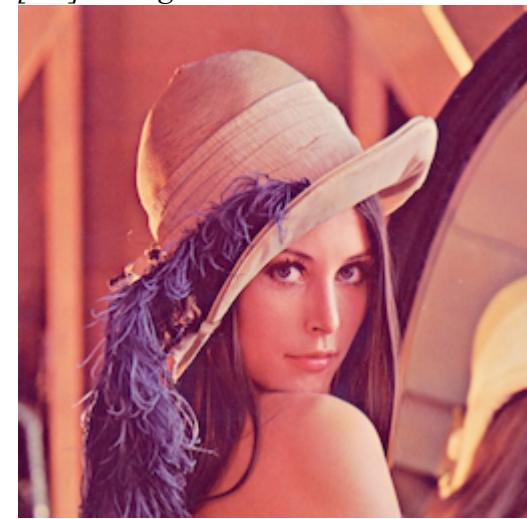
même valeur $lut(i)$: Voir le troisième exemple avec la fonction `set_levels_picture` ci-dessous.

- d'un tableau de n valeurs.
- De quoi gérer les LUTs :
 - Création
 - Nettoyage
 - Application d'une LUT à une image pour créer l'image modifiée.

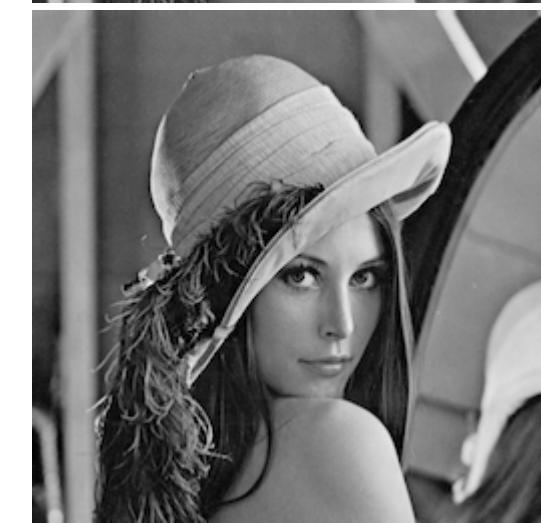
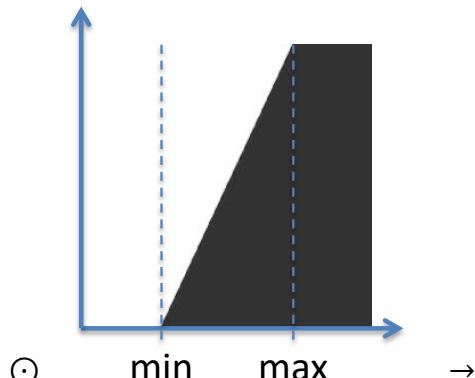
Ce module pourra alors être utilisé dans votre module principal `pictures.c`, mais ne devra pas être visible de votre programme principal `main.c`.

Les opérations à réaliser avec des LUTs sont les suivantes :

- **Inverser les valeurs** d'une image `picture inverse_picture(picture p);` de $[0 \dots max] \rightarrow [max \dots 0]$
 - *[in] l'image à inverser*
 - *[out] l'image inversée*



- **Normaliser les valeurs d'une image** : `picture normalize_dynamic_picture(picture p);`
 - Si les valeurs d'une image sont comprises entre \min et \max (à rechercher dans les valeurs des composantes des pixels), la normalisation des valeurs d'une image répartit celles-ci entre 0 et 255.
 - *[in, out] p l'image à normaliser*



- **Réduction du nombre de niveaux pour les pixels d'une image : picture**
`set_levels_picture(picture p, byte nb_levels);`
 - [in, out] p l'image dont on veut changer le nombre de niveaux



- La LUT utilisée ici est une une LUT identité $lut(i) = i$ de taille 8 à laquelle on a ajouté 1. Le fait d'utiliser une LUT de taille 8 (avec des valeurs $\in [0 \dots 7]$) permet de discréteriser les valeurs dans 8 groupes lorsque l'intervalle $[0 \dots 7]$ est étalé sur $[0 \dots 255]$. Il faudra pour ce faire que l'on puisse appliquer des LUTs de tailles inférieures à 256.

Opérations arithmétiques sur les images

- **Différence entre deux images**
 - *[in] la première image*
 - *[in] la seconde image*
 - *[out] une image contenant la différence (en valeur absolue) des deux images en entrée.*
- **Multiplication de deux images** : `picture mult_picture(picture p1, picture p2)`
 - *[in] p1 la première image*
 - *[in] p2 la seconde image*
 - *[out] une image contenant le produit des deux images en entrée.*

- Exemple :



p1

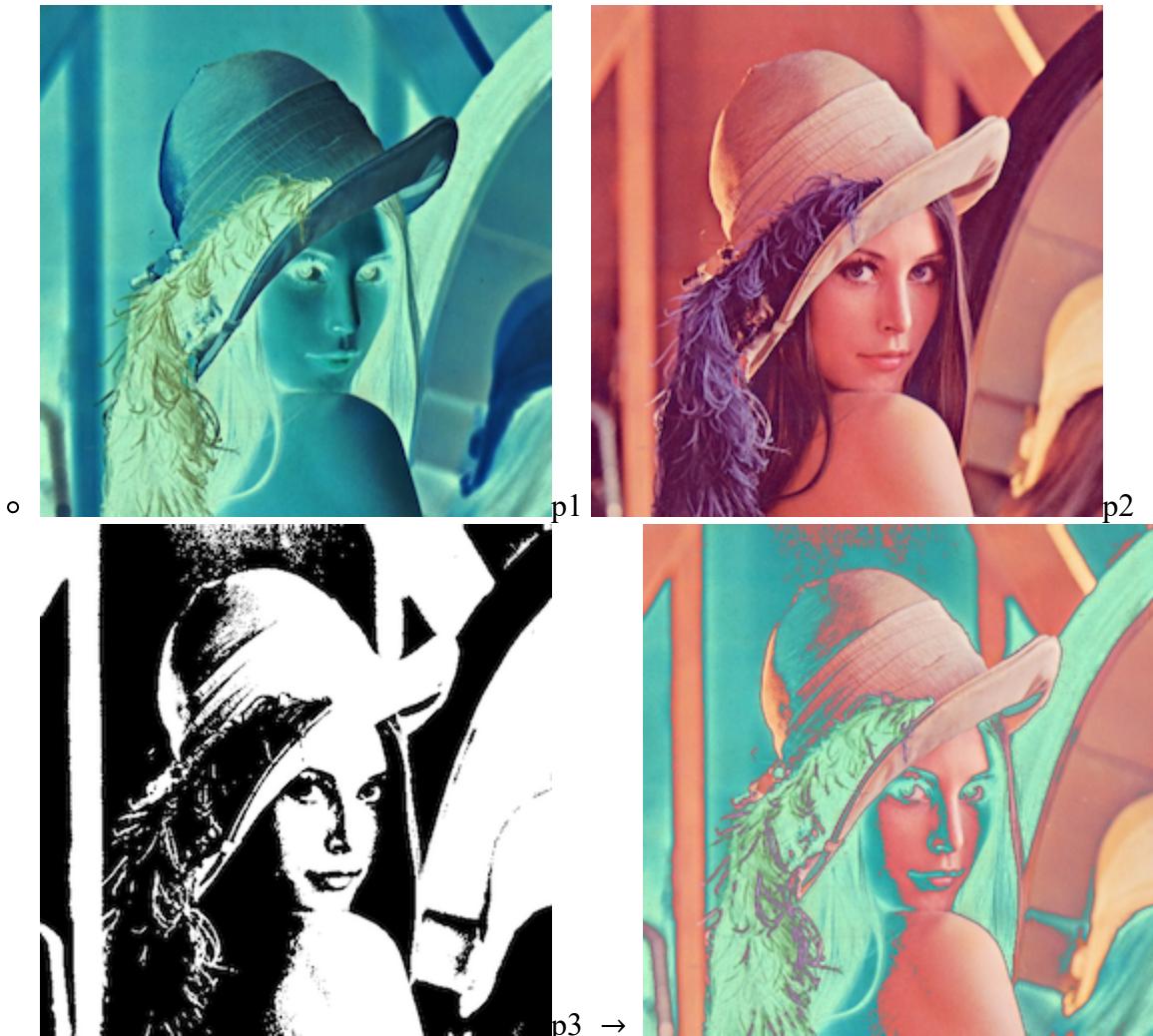


p2



→

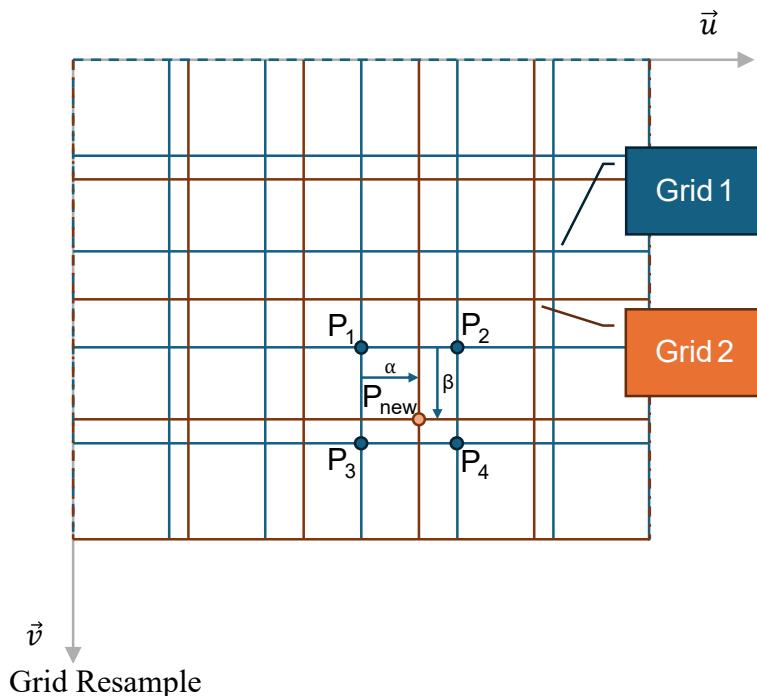
- **Mé lange de deux images suivant une 3ème image** : picture mix_picture(picture p1, picture p2, picture p3)



- $result(i, j, c) = (1 - \alpha) \cdot p1(i, j, c) + \alpha \cdot p2(i, j, c)$ avec $\alpha = \frac{p3(i, j, c)}{255}$

Re-échantillonnage d'images

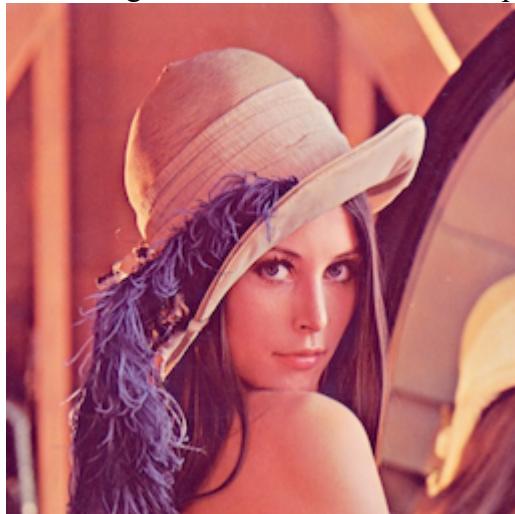
Le re-échantillonnage d'image consiste à changer la taille d'une image (largeur et/ou hauteur). L'idée principale consiste à calculer la position des pixels de l'image résultat sur la grille des pixels de l'image source. La figure suivante présente une grille 7×6 que l'on cherche à re-échantillonner par une grille 6×5 .



Ainsi, un pixel de l'image résultat se situera toujours entre 4 pixels de l'image source. On peut ainsi définir deux politiques pour calculer les valeurs des pixels de l'image résultat :

- **La politique de plus proche voisin** (nearest neighbor) : la valeur d'un pixel résultat sera celle du pixel de l'image source le plus proche.

- Dans le cas de la figure ci-dessus la valeur d'un pixel de l'image résultat $P_{new} = P_4$



- Exemple

$256 \times 256 \rightarrow$



348×348

- **Une politique d'interpolation bi-linéaire** : La valeur d'un pixel résultat est le résultat de l'interpolation bi-linéaire des valeurs des 4 pixels de l'image source entourant le pixel de l'image résultat.

- Dans le cas de la figure ci-dessus $\alpha \in [0 \dots 1]$ correspond au coefficient d'interpolation horizontal et $\beta \in [0 \dots 1]$ au coefficient d'interpolation vertical, on a alors la valeur d'un pixel de l'image résultat $P_{new} = (1 - \alpha)(1 - \beta)P_1 + \alpha(1 - \beta)P_2 + (1 - \alpha)\beta P_3 + \alpha\beta P_4$.



Vous pouvez consulter les pages suivantes pour de plus amples explications sur le re-échantillonnage :

- [Image Resampling Algorithms](#) qui présente les deux techniques mentionnées ci-dessus.
- [Image scaling](#) qui présente de nombreuses autres techniques de re-échantillonnage.

Programme Principal

Votre programme principal devra consister en la lecture d'une ou plusieurs images source depuis des fichiers (PGM ou PPM) fournis en arguments du programme. Exemple :

```
./projet Lenna_gray.pgm Lenna_color.ppm
```

Votre programme devra alors écrire les fichiers suivants :

- Lenna_gray_convert_color.ppm qui contiendra la conversion de Lenna_gray.pgm en une image couleur
- Lenna_color_convert_gray.pgm qui contiendra la conversion de Lenna_color.ppm en une image en niveaux de gris.
- Lenna_color_red.pgm qui contiendra la composante rouge de Lenna_color.ppm
- Lenna_color_green.pgm qui contiendra la composante verte de Lenna_color.ppm
- Lenna_color_blue.pgm qui contiendra la composante bleue de Lenna_color.ppm
- Lenna_[gray|color]_brighten.p[g|p]m qui contiendra l'image d'entrée éclaircie d'un facteur 1.5.
- Lenna_[gray|color]_melted.p[g|p]m qui contiendra l'image dont on aura fait "fondre" $width \times height \times channels \times 5$ pixels.

- Lenna_[gray|color]_inverse.p[g|p]m qui contiendra l'inversion de l'image source.
- Lenna_gray_dynamic.pgm qui contiendra la version avec une dynamique (différence entre les niveaux les plus sombre et les plus clair) maximisée (aussi appelée dynamique optimale) de Lenna_gray.pgm.
- Lenna_color_dynamic.ppm qui ne contiendra pas directement la dynamique optimale de Lenna_color.ppm car celle-ci présente déjà une dynamique quasiment optimale. Ce n'est pas le cas en revanche de chacune de ses composantes (rouge, vert, bleue) prise séparément. Appliquez une dynamique optimale sur chacune des composantes, puis re-composez une image couleur à partir de ces composantes optimisée que vous sauverez dans le fichier Lenna_color_dynamic.ppm.
- Lenna_[gray|color]_levels.p[g|p]m qui contiendra une version de l'image source limitée à 8 niveaux par composante.
- Lenna_[gray|color]_smaller_nearest.p[g|p]m qui contiendra l'image source rétrécie d'un facteur 1.36 en utilisant la politique du plus proche voisin.
- Lenna_[gray|color]_smaller_bilinear.p[g|p]m qui contiendra l'image source rétrécie d'un facteur 1.36 en utilisant la politique du plus proche voisin.
- Lenna_[gray|color]_larger_nearest.p[g|p]m qui contiendra l'image source agrandie d'un facteur 1.36 en utilisant la politique du plus proche voisin.
- Lenna_[gray|color]_larger_bilinear.p[g|p]m qui contiendra l'image source agrandie d'un facteur 1.36 en utilisant la politique d'interpolation bi-linéaire.
- Lenna_[gray|color]_difference.p[g|p]m qui contiendra la différence normalisée entre les deux types d'interpolations utilisées. Par exemple entre Lenna_[gray|color]_larger_nearest.p[g|p]m et Lenna_[gray|color]_larger_bilinear.p[g|p]m. Pour normaliser cette différence, il suffira de maximiser sa dynamique.
- Pour les opérations suivantes vous aurez besoin de charger une nouvelle image (qui contient si possible une majorité de noir et de blanc comme dans l'image Lenna_BW.pgm) qui servira de masque. Si cette image n'est pas de la bonne taille, vous pourrez la redimensionner à la taille de vos images d'entrées.
 - Lenna_[gray|color]_product.p[g|p]m qui contiendra le produit de l'image en entrée avec le masque.
 - Lenna_[gray|color]_mixture.p[g|p]m qui contiendra la mixture de l'image inversée calculée précédemment et de l'image en entrée en utilisant le masque.

Conseils

- Dans toutes les opérations sur les composantes des pixels, il faudra veiller aux erreurs de dépassement avec un type byte utilisant 1 octet et dont les valeurs doivent donc être dans l'intervalle [0…255]. Par exemple, dans l'expression byte b = 192 + 137; b vaudra $(192 + 137) \% 256 = 73$ et non pas 329 comme avec le type int.
- Vous veillerez dans votre programme à libérer les ressources qui ne sont plus utilisées.
- Utilisez les fonctions du module filename.[h|c] pour décomposer les noms de fichiers à lire et composer les noms des fichiers à écrire.

Questions Bonus

Si vous avez terminé les questions précédentes ainsi que le programme principal.

Utilisation d'une LUT pour l'éclaircissement

Vous remarquerez que vous auriez pu utiliser une LUT pour "éclaircir" l'image comme mentionné dans la [Manipulation directe des valeurs des pixels](#).

Concevez une LUT appliquant le même effet que l'éclaircissement d'image et appliquez la dans une fonction `picture brighten_picture_lut(picture p, double factor);`

Filtrage d'images

Le filtrage pour une image consiste à convoluer une image avec un "[noyau](#)" contenant des coefficients. La valeur d'un pixel de l'image résultat $y(k, l)$ est le résultat de la somme des pixels $x(k - k', l - l')$ de

l'image source dans un voisinage de taille $n \times n$ multipliés par les coefficients du noyau $f(k', l')$:

- $y(k, l) = \sum_{k'=0}^{k'=n-1} \sum_{l'=0}^{l'=n-1} x\left(k - \frac{n}{2} + k', l - \frac{n}{2} + l'\right) \cdot f(k', l')$ avec $k' \in [0 \dots N-1]$ & $l' \in [0 \dots M-1]$ où M et N sont respectivement les nombres de lignes et de colonnes de l'image x .
- où $x(k, l)$ représente le pixel à la ligne l et la colonne k .
 - Lorsque $k - \frac{n}{2} + k' < 0$ on pourra remplacer par 0
 - Lorsque $k - \frac{n}{2} + k' > N-1$ on pourra remplacer par $N-1$
 - Il en ira de même pour l' avec comme limite supérieure $M-1$.
- $f(k', l')$ représente la réponse impulsionnelle du filtre contenant n lignes et n colonnes.
 - En général $n \ll M$ et $n \ll N$.

On dénotera un noyau de filtrage k comme ceci :

$$\text{factor} \begin{pmatrix} k_{11} & \cdots & k_{1n} \\ \vdots & \ddots & \vdots \\ k_{n1} & \cdots & k_{nn} \end{pmatrix} + \text{offset}$$

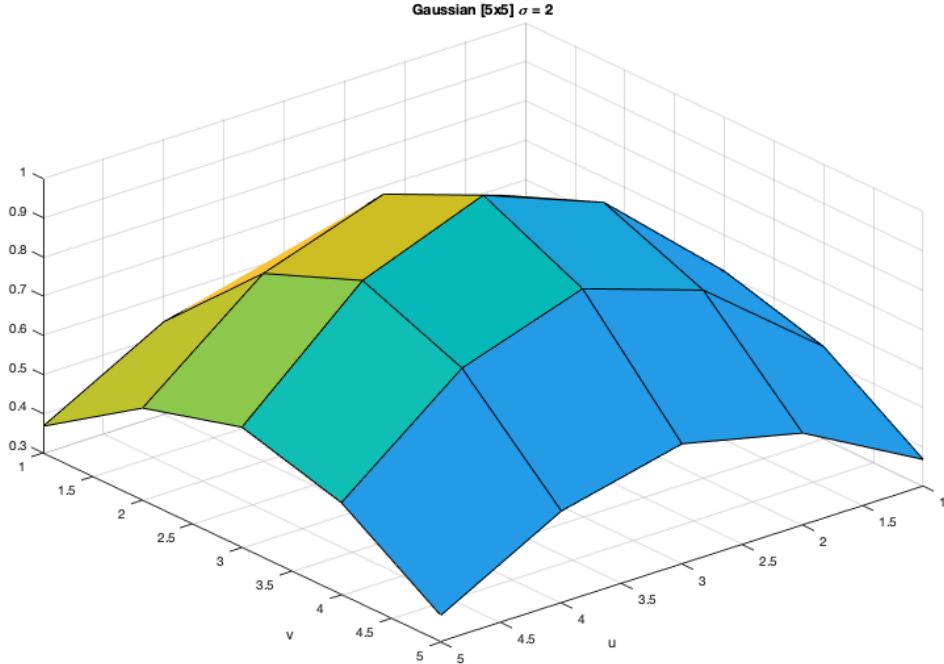
- Les k_{ij} représentent les coefficients du noyau.
- factor indique le facteur multiplicatif à appliquer aux coefficients k_{ij} pour normaliser leur somme afin de ne pas dépasser la valeur MAX_BYTE dans un pixel résultat.
- Et offset indique l'offset à ajouter à la valeur d'un pixel résultat pour que celui ci reste dans l'intervalle $[0 \dots 255]$

Exemples de noyaux avec $n = 3$:

$$\frac{1}{9} \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} + 0, \frac{1}{4} \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ 1 & 0 & 1 \end{pmatrix} + \frac{255}{2}, \frac{1}{4} \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} + \frac{255}{2}$$

Exemple de noyau avec $n = 5$ représentant une gaussienne d'écart type 2

$$\begin{aligned} G(u, v) &= Ae^{-\frac{(u-u_0)^2(v-v_0)^2}{2\sigma^2}} \\ n &= 5 \\ A &= 0.063 \\ \sigma &= 2.0 \\ u, v &\in [0 \dots n-1] \\ u_0, v_0 &= \lfloor \frac{n}{2} \rfloor \\ \\ 0.063 &\begin{pmatrix} 0.368 & 0.535 & 0.607 & 0.535 & 0.368 \\ 0.535 & 0.779 & 0.882 & 0.779 & 0.535 \\ 0.607 & 0.882 & 1.000 & 0.882 & 0.607 \\ 0.535 & 0.779 & 0.882 & 0.779 & 0.535 \\ 0.368 & 0.535 & 0.607 & 0.535 & 0.368 \end{pmatrix} + 0 \end{aligned}$$

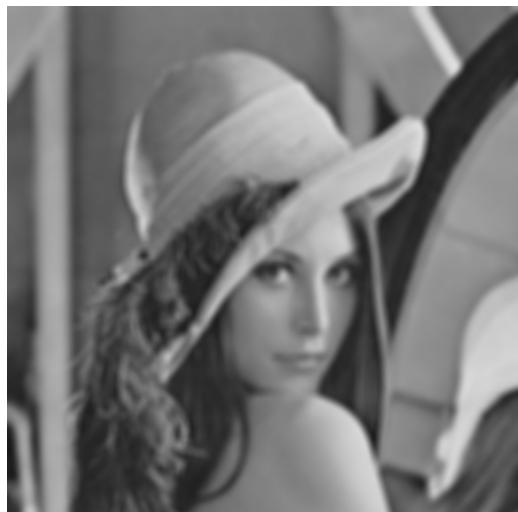


gaussienne 5x5, sigma = 2

Exemples d'images filtrées



• filtrée avec le noyau gaussien ci-dessus →

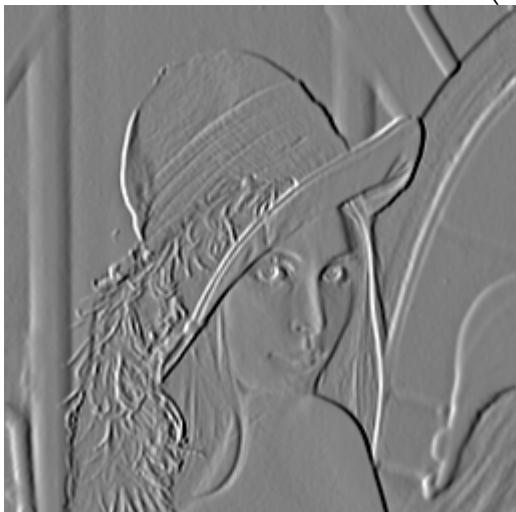


ce qui permet de flouter l'image.



- filtrée avec le noyau

$$\frac{1}{4} \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ 1 & 0 & 1 \end{pmatrix} + \frac{255}{2}$$



→ Ce qui permet d'estimer une (parmi d'autres) dérivée horizontale de l'image.

VSCODE

Les extensions suivantes pour Visual Studio Code vous seront utiles :

- Extension [C/C++](#)
 - [C/C++ Extension Pack](#)
- Extension pour gérer un projet basé sur un Makefile : [Makefile tools](#)
- Pour vous aider à visualiser les images PGM ou PPM, vous pourrez utiliser l'extension suivante : [PBM/PPM/PGM Viewer for Visual Studio Code](#).