DS MPSI : flottants et images

Solution. □

Les réponses doivent être écrites sur les feuilles quadrillées A4 fournies. Les pages doivent être rendues non agrafées, non cornées et numérotées avec le recto de la page 1 au-dessus.

1 Flottants

Soit $x \in \mathbb{R}^*$. Il existe un unique triplet $(s, e, m) \in \{0,1\} \times \mathbb{Z} \times [0,1[$ tel que

$$x = (-1)^s(1+m)2^e.$$

Le nombre s est appelé *bit de signe* ; e est l'*exposant* et m la *mantisse*. Le nombre $1+m$ est appelé le *significande complet*.

L'exposant e est l'unique entier tel que

$$|x| \times 2^{-e} \in [1,2[.$$

On fixe un format de représentation des nombres en virgule flottante, dans lequel chaque nombre est codé par un mot binaire composé d'un bit de signe, de E bits d'exposant et de M bits de mantisse. Pour ce format, on définit le *biais* b comme $b = 2^{E-1} - 1$. **On ne considère que les nombres pour lesquels** $e \in [\![-b+1, b]\!]$: on les appelle *nombres représentables* ou encore *nombres normalisables*.

L'encodage $\mathrm{enc}(x)$ de x est un mot binaire défini comme la concaténation de 3 mots w_s, w_e, w_m possédant respectivement 1, E et M bits :

- Le mot w_s, appelé *bit de signe* est simplement la valeur de s ;
- le mot w_e, appelé *champ exposant* est l'encodage binaire de l'*exposant décalé*, c'est à dire de la somme de e et du *biais*.
- Le mot w_m, appelé *champ mantisse*, est l'encodage de la mantisse. Le significande complet est de la forme $1+m$ avec $m \in [0,1[$; sa partie entière vaut toujours 1 (*bit implicite*). On n'encode donc que les M bits de la partie fractionnaire m en s'aidant de 3 bits supplémentaires.
  Pour déterminer ces bits, on itère le procédé suivant : multiplier m par 2, extraire la partie entière (qui fournit le bit suivant), puis conserver la partie fractionnaire. On réalise ce calcul $M+2$ fois ; les deux derniers bits étant appelés *bits de garde* (en anglais *guard*) et *d'arrondi* (en anglais *round*). Après avoir déterminé $M+2$ bits, un dernier bit (appelé *sticky bit*) est calculé selon la valeur de la fraction restante (notée m_r) :

  $$\text{sticky} = \begin{cases} 0 & \text{si } m_r = 0, \\ 1 & \text{sinon.} \end{cases}$$

  Ce sticky indique si des bits non nuls apparaissent au-delà du rang $M+2$.
  On applique alors la règle de l'arrondi au plus proche pair à l'aide du triplet (guard, round, sticky). Si ce triplet est différent de $(1,0,0)$, la mantisse w_m est obtenue en choisissant l'unique mot binaire de M bits le plus proche de m. En revanche, lorsque le triplet vaut $(1,0,0)$, on est en situation d'équidistance entre deux valeurs possibles ; on choisit alors celle dont le bit de poids faible est nul (règle du pair).
  Le réel x n'est jamais connu exactement : le procédé ci-dessus revient en pratique à travailler avec une approximation finie de m, la fraction restante m_r qui sert à calculer le bit sticky condense toute l'information des bits non calculés.

Pour un réel x représentable, on détermine donc son encodage binaire

$$\mathrm{enc}(x) = b_0 b_1 \dots b_{E+M}$$

à l'aide de l'algorithme précédent.

À partir de l'encodage binaire $b_0 b_1 \dots b_{E+M}$ d'un nombre représentable, on obtient le nombre :

$$\text{dec}(b_0 b_1 \dots b_{E+M}) = (-1)^{b_0} 2^{e'} (1 + m')$$

où l'exposant e' est obtenu en interprétant $b_1 \dots b_E$ en base 2 puis en retranchant le biais, et la mantisse m' vaut

$$m' = \sum_{i=1}^{M} b_{E+i} 2^{-i}.$$

Question 1.

Quels mots binaires de E bits ne sont pas des champs exposants d'un nombre normalisable ?

Solution. Les mots identiquement nuls et identiquement égaux à 1. En effet, ces deux valeurs du champ exposant sont réservées respectivement aux zéros et aux nombres dénormalisés, et aux infinis et aux `NaN`. □

Question 2.

En `Python`, $E = 11$ et $M = 52$. Décrire les mots binaires représentant zéro et $-\infty$ comme des flottants `Python`.

Solution. Pour zéro, 0 ou 1 suivi de 63 bits nuls.
Pour $-\infty$: $1 + E$ bits à 1 puis M bits à 0. □

On appelle *application d'arrondi* et on note f l'application qui a un nombre représentable associe le décodage de son encodage :

$$f = \text{dec} \circ \text{enc}$$

Cette fonction associe à tout x représentable la valeur flottante la plus proche selon la règle d'arrondi au plus proche pair. On appelle *ensemble des nombres normalisés* l'image par f de l'ensemble des nombres représentables. Un nombre normalisé est donc un point fixe de f, c'est à dire que $f(x) = x$.

Question 3. Nombre de flottants

Donner le cardinal de l'ensemble des nombres normalisés.

Solution. Le nombre de mots binaires correspondant aux nombre normalisés est

$$2 \times (2^E - 2) \times 2^M$$

On peut également pinailler en remarquant que zéro n'est pas à proprement parler normalisable : il faudrait enlever ses deux écritures :

$$2 \times (2^E - 2) \times 2^M - 2$$

□

Question 4.

Quelle est la valeur du *biais* si $E = 4, M = 8$?

Solution. $2^{E-1} - 1$ et $2^{4-1} - 1 = 7$ □

Question 5. Encodage

Calculer l'expression binaire de $f(\frac{1}{5})$ si $E = 4,\ M = 8$. On donnera bien la valeur du triplet (guard,round,sticky).

Solution. Puisque $\frac{1}{5} \times 2^3 \in [1; 2[$: $e = -3$ et $1 + m = \frac{3}{5} = 1.6$. Alors l'exposant décalé est 4 soit 0100_2. De plus $0.6 \times 2 = 1.2$, $0.2 \times 2 = 0.4$ $0.4 \times 2 = 0.8$ et $0.8 \times 2 = 1.6$. Puis c'est cyclique. La mantisse vaut

$$1001\,1001\,100\,1...$$

Alors (guard,roud,sticky) vaut (1,0,1) (puisqu'il y a des 1 après le dixième bit).

On applique l'arrondi au plus proche pair :

$$\begin{array}{r} +10011001 \\ 00000001 \\ \hline 10011010 \end{array}$$

Alors

$$\mathrm{enc}(f(\frac{1}{5})) = 0\,0100\,10011010$$

□

Question 6. Décodage

L'encodage d'un nombre normalisé x au format $E = 4,\ M = 8$ est le mot :

$$1\,0110\,10100000$$

Quelle est la valeur de x en base 10 exprimée sous forme de fraction irréductible ?

Solution.

$$(-1)^1 2^{2^2+2^1-7}(1 + \frac{1}{2} + \frac{1}{8}) = -(\frac{1}{2})\frac{8+4+1}{8} = -\frac{13}{16}$$

□

Question 7. Plus grand nombre représentable

Quelle est la valeur du plus grand nombre normalisé dans le format E, M ? Indiquer d'abord son expression binaire puis sa valeur en fonction de E, M sous la forme la plus condensée possible.

Application au format $E = 4, M = 8$.

Solution. Le biais vaut

$$b = 2^{E-1} - 1.$$

Le plus grand nombre normalisé est obtenu avec un bit de signe nul, le plus grand exposant décalé autorisé pour un nombre normalisé, et la plus grande mantisse possible. Son encodage est donc

$$0\ \underbrace{11\ldots10}_{E}\ \underbrace{11\ldots11}_{M}.$$

Son exposant décalé vaut $2^E - 2$, donc son exposant réel vaut

$$e = (2^E - 2) - b = (2^E - 2) - (2^{E-1} - 1) = 2^{E-1} - 1.$$

Sa mantisse vaut

$$m = \sum_{i=1}^{M} 2^{-i} = 1 - 2^{-M},$$

donc son significande complet vaut

$$1 + m = 2 - 2^{-M}.$$

Ainsi, le plus grand nombre normalisé vaut

$$2^{2^{E-1}-1}\left(2 - 2^{-M}\right) = 2^{2^{E-1}} - 2^{2^{E-1}-1-M}.$$

□

Solution. Au format $(4, 8)$, on obtient

$$2^{2^{4-1}-1}\left(2 - 2^{-8}\right) = 2^7\left(2 - \frac{1}{256}\right) = 128 \cdot \frac{511}{256} = \frac{511}{2}.$$

Donc le plus grand nombre normalisé vaut

$$\frac{511}{2} = 255.5.$$

□

Question 8. Encodage de 1

Le nombre 1 est exactement représentable. Donner son encodage binaire.

Solution.

$$1 = (-1)^0 2^0 (1 + 0)$$

Son exposant décalé vaut b c'est à dire $2^{E-1} - 1$ soit 0 suivi de $E - 1$ bits à 1. Sa mantisse est nulle

$$0\ \underbrace{01\ldots11}_{E}\ \underbrace{00\ldots00}_{M}.$$

□

L'ensemble des nombres normalisés au format E, M est fini : tout nombre normalisé y non égal au maximum possède donc un unique *successeur* noté $s(y)$: le minimum des nombres normalisés plus grands strictement que y.

Question 9. Décodage de $s(1)$

Donner l'encodage binaire de $s(1)$ puis son expression en base 10.

Solution. L'encodage de $s(1)$ est

$$0\ \underbrace{01\ldots11}_{E}\ \underbrace{00\ldots0001}_{M}.$$

En base 10 :

$$2^0(1 + \frac{1}{2^M})$$

□

Question 10. Epsilon machine

En déduire la valeur du *epsilon machine*, c'est à dire $s(1) - 1$. On note ε ce nombre.

Solution.

$$\frac{1}{2^M}$$

□

Question 11.

Quelle est la valeur du epsilon machine si $E = 4$ et $M = 8$? Est-il représentable comme nombre normalisé dans ce format ?

Solution. Le plus petit exposant admissible pour le format $4, 8$ est $1 - b = 1 - 7 = -6$.

Or

$$\varepsilon = (-1)^0 2^{-8}(1 + 0)$$

Donc son exposant est -8. Non représentable.

Remarque. A noter que en `Python`, le epsilon machine est 2^{-52}. Or le plus petit exposant autorisé est $1-(2^{11-1}-1) = -1022$. Donc en `Python`, le epsilon machine est représentable comme nombre normalisé.

□

Nous voulons maintenant donner un ordre de grandeur de l'erreur de représentation des réels par des flottants.

Question 12. Erreur absolue

Soit un réel non nul $x = (-1)^s(1 + m)2^e$ tel que $f(x) < N$ en notant N le plus grand nombre exactement représentable du format. Donner une majoration la plus fine possible de $|x - f(x)|$ par une fonction de ε et e.

Solution. Écrivons

$$x = (-1)^s(1 + m)2^e \qquad \text{et} \qquad f(x) = (-1)^s(1 + m')2^e,$$

La mantisse m' est obtenue en arrondissant m au plus proche sur M bits (règle du pair en cas d'égalité).

Remarque. Observons que si $m' \geq 1$, l'exposant véritable de $f(x)$ est $e + 1$. Ce qui fait que l'écriture de $f(x)$ telle qu'elle est donnée n'est peut-être pas l'écriture scientifique (mais cette remarque ne change rien aux calculs ci-dessous).

Ecrivons $1 + m$ ainsi :

$$\sum_{i=0}^{+\infty} b_i 2^{-i}$$

En posant $f(x) = (-1)^s 2^e(1 + m')$, on a

$$1 + m' = \sum_{i=0}^{M} b_i 2^{-i} + \begin{cases} \mathbf{0} & \text{si } b_{M+1} = 0 \\ \mathbf{0} & \text{si } (b_M, b_{M+1}, b_{M+2}, sticky) = (0, 1, 0, 0) \\ \mathbf{2^{-M}} & \text{si } (b_M, b_{M+1}, b_{M+2}, sticky) = (1, 1, 0, 0) \\ \mathbf{2^{-M}} & \text{sinon} \end{cases}$$

si $b_{M+1} = 0$ Alors $m > m'$ et

$$m - m' \leq 0 \times 2^{-M-1} + 2^{-M-2} \sum_{i=1}^{+\infty} b_i 2^{-i} \leq 2^{-M-1}(2) = 2^{-M-1}$$

si $(b_M, b_{M+1}, b_{M+2}, sticky) = (0, 1, 0, 0)$ alors

$$m = \sum_{i=1}^{M+1} b_i 2^{-i}$$

et donc, on a $m > m'$ et

$$m - m' = 2^{-M-1}$$

si $(b_M, b_{M+1}, b_{M+2}, sticky) = (1, 1, 0, 0)$ alors m a la valeur donnée au dessus et $m' = m + 2^{-M}$. Donc

$$0 \leq m' - m = 2^{-M} - 2^{-M-1} = 2^{-M-1}$$

Autres cas Dans tous les autres cas, la mantisse m' vaut

$$\sum_{i=0}^{M} b_i 2^{-i} + 2^{-M}$$

et on a

$$(\sum_{i=0}^{M} b_i 2^{-i}) + \ 2^{-M-1} < m \leq \sum_{i=1}^{M} b_i 2^{-i} + 2^{-M-1} \sum_{i=0}^{+\infty} 1 \leq (\sum_{i=1}^{M} b_i 2^{-i}) + 2^{-M-1} \times 2 = m'$$

Ainsi $m' > m$ et

$$0 \leq m' - m < 2^{-M-1} = \frac{\varepsilon}{2}$$

Ainsi :

$$|m - m'| \leq \frac{2^{-M}}{2} = \frac{\varepsilon}{2},$$

avec égalité uniquement en cas d'équidistance, c'est-à-dire lorsque le triplet (guard, round, sticky) vaut $(1, 0, 0)$.

On en déduit

$$|x - f(x)| = |(-1)^s(1+m)2^e - (-1)^s(1+m')2^e| = |m - m'|\, 2^e \leq \frac{\varepsilon}{2} 2^e.$$

□

Question 13. Erreur relative

Avec les mêmes conventions qu'à la question précédente majorer $\dfrac{|x - f(x)|}{|x|}$ par une constante.

Solution. D'après la question précédente,

$$|x - f(x)| \leq \frac{\varepsilon}{2} 2^e$$

On en déduit :

$$\frac{|x - f(x)|}{|x|} \leq \frac{\varepsilon 2^{e-1}}{(1+m)2^e} \leq \frac{\varepsilon}{2},$$

puisque $m \in [0, 1[$. □

Question 14. Cas pathologique

Question annulée

Question 15.

Le epsilon-machine peut être vu comme le plus petit nombre normalisé qui ajouté à 1 donne une quantité différente de 1. Connaissant cette propriété, donner un code pour retrouver la longueur du champ mantisse en `Python` si on a oublié de lire la documentation officielle.

Solution. Code

```
eps = 1
e = 0
while 1 + eps != 1:
    eps = eps / 2
    e+=1
e-1 # 52
```

□

2 Images

2.1 Images en couleur

Solution. Quand le sujet impose un type de retour différent de `None` dans la signature d'une fonction, par exemple : `negatif(img: np.ndarray) -> np.ndarray`,
alors les effets de bords (modification des paramètres d'entrée) sont à proscrire.

S'agissant des images, on construit une nouvelle image sans adresse commune avec celle passée en entrée. □

Dans cette partie, nous étudions les images en couleur. Notre image de référence est une photographie d'astronaute fournie par la bibliothèque `scikit-image` de `Python`.

Pour information, on commence par charger cette image sous forme de tableau (matrice de pixels), puis par la sauvegarder dans un fichier.

```
import matplotlib.image as mpimg
from skimage import data

# Chargement d'une image couleur
# img est un tableau NumPy de taille (hauteur, largeur, 3)
img = data.astronaut()

# Sauvegarde de l'image dans un fichier PNG
mpimg.imsave("astronaut.png", img)
```

Toujours à titre informatif, on peut ensuite afficher cette image à l'écran à l'aide de la bibliothèque `matplotlib` :

```
import matplotlib.pyplot as plt

# Affichage de l'image
plt.imshow(img)

# Suppression des axes pour
# un affichage plus lisible
plt.axis("off")

# Affichage à l'écran
plt.show()
```

FIGURE 1 – Affichage de l'image décrite par `img`

Les commandes précédentes sont données uniquement à titre indicatif, afin d'illustrer la manière dont une image peut être manipulée en `Python`. Dans la suite du problème, on ne demande ni d'afficher ni de sauvegarder les images : on travaille uniquement sur leur représentation matricielle.

Dans toute la suite, une image est donc vue comme un tableau à trois dimensions, où chaque pixel est décrit par trois composantes correspondant aux intensités des couleurs rouge, vert et bleu (RGB). Plus précisément, nous considérons que les images manipulées dans ce problème sont représentées par des objets de type `numpy.ndarray`, c'est-à-dire des tableaux (multidimensionnels) de nombres. Une image couleur est un tableau à trois dimensions : si `img` désigne une image, alors `img[i][j]` correspond au pixel situé à la ligne `i` et à la colonne `j`, et est un triplet de la forme `[R, G, B]`.

Les composantes `R`, `G` et `B` sont des entiers compris entre 0 et 255, représentant respectivement les intensités des couleurs rouge, vert et bleu. En pratique, ces entiers sont stockés avec un type particulier appelé « `uint8` », qui signifie « entier non signé codé sur 8 bits » : cela permet de représenter exactement les entiers de 0 à 255.

Ainsi, un pixel est un triplet d'entiers compris entre 0 et 255, et une image est un tableau de tels triplets.

Attention au type `uint8`. Les composantes des pixels étant codées sur 8 bits, les opérations arithmétiques sont effectuées modulo 256. Ainsi, un dépassement de capacité (« overflow ») peut se produire.

Par exemple :

$$250 + 15 = 265 \equiv 9 \pmod{256}.$$

Autrement dit, si l'on additionne deux intensités trop grandes, le résultat peut être ramené dans l'intervalle $[0, 255]$ de manière circulaire.

```
import numpy as np
np.uint8(250) + np.uint8(15) # produit 9 mais affiche un avertissement overflow
```

Si `img` est une image, alors `img.shape` est un triplet de la forme `(H, L, 3)`, où :

— `H` est le nombre de lignes (hauteur de l'image),

— `L` est le nombre de colonnes (largeur de l'image),

— `3` correspond aux trois composantes de couleur (RGB).

Ainsi, le premier indice correspond à la position verticale (ligne) et le second à la position horizontale (colonne). En particulier, `img[i, j]` désigne le pixel situé à la ligne `i` et à la colonne `j`.

Par conséquent, l'expression `img[0]` désigne la première ligne de l'image (et non la première colonne).

Dans ce devoir, on importe `numpy` une fois pour toutes avec le préfixe `np.`. On peut alors construire une image « à la main » en utilisant des listes :

```
import numpy as np

pixel1 = [0, 200, 255]   # bleu-vert assez vif
pixel2 = [0, 0, 255]     # bleu
pixel3 = [255, 255, 255] # blanc
pixel4 = [0, 0, 0]       # noir

# image 2x2 construite comme une liste de listes de pixels
liste = [[pixel1, pixel2],
         [pixel3, pixel4]]

# conversion en tableau numpy avec le bon type
image = np.array(liste, dtype=np.uint8)  # conversion en entiers sur 8 bits
```

On obtient ainsi une image de taille 2×2, où chaque pixel est décrit par un triplet (R, G, B). Les composantes des pixels (initialement des entiers `Python`) sont converties en entiers de type `uint8`, c'est-à-dire des entiers compris entre 0 et 255.

Question 16.

On souhaite transformer une image en son *négatif*.

1. Que devient un pixel de composantes (R, G, B) dans l'image négative ?
2. Écrire une fonction `negatif(img: np.ndarray) -> np.ndarray` qui à une image `img` (tableau de taille $H \times L \times 3$ à valeurs entières dans $[\![0, 255]\!]$ associe son négatif.

Solution. Le sujet pouvait certes porter à confusion. Certains en ont déduit que l'on pouvait modifier l'image passée en paramètre : le code obtenu était alors plus court. MAIS le type de retour attendu était `np.array`. Si on choisissait de faire des *effets de bord* (=modification du paramètre d'entrée), alors il n'y avait aucune nécessité de renvoyer `img`.

```
import numpy as np

def negatif(img):
    H, L, _ = img.shape  # hauteur, largeur, 3 canaux

    res = []  # future image sous forme de liste
    # l'image initiale n'est pas modifiée !
    for i in range(H):
        ligne = []
        for j in range(L):
            pixel = img[i][j]
            R = pixel[0]
```

```
            G = pixel[1]
            B = pixel[2]

            # calcul du négatif
            ligne.append([255 - R, 255 - G, 255 - B])

        res.append(ligne)

    # conversion en tableau numpy
    return np.array(res, dtype=np.uint8)
```

Attention : `255 - R` convertit `R` en entier `Python`. Il faut reconvertir en `np.uint8` avec `dtype=np.uint8`. □

La figure 2 donne un aperçu de l'image obtenue en passant au négatif.

```
negimg = negatif(img)
plt.imshow(negimg)
plt.axis("off")
plt.show()
```

FIGURE 2 – Le négatif de la photo de l'astronaute

Question 17.

On souhaite construire une image représentant un damier rouge et vert.

Soient n et k deux entiers naturels tels que $k \leq n$. On veut construire un damier vert et rouge de $2^k \times 2^k$ blocs, chaque bloc étant de taille $2^n \times 2^n$.

Dans ce qui suit, toutes les fonctions renvoient des listes de listes de 3 entiers sauf la dernière qui renvoie un tableau `numpy`.

1. Que valent les triplets RGB correspondant aux couleurs rouge et verte ?
2. Écrire une fonction

```
bloc(couleur: list[int], k: int) -> list[list[list[int]]]
```

qui renvoie une image carrée de taille $2^k \times 2^k$ dont tous les pixels sont de la couleur donnée (un triplet `[R,G,B]` d'entiers `Python`).

Code

```
bloc([0,0,0],2)  # bloc noir
```

Rendu

```
[[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]],
 [[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]],
 [[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]],
 [[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]]
```

3. Écrire une fonction

```
juxtapose_h(b1: list[list[list[int]]],
            b2: list[list[list[int]]]) -> list[list[list[int]]]
```

qui prend deux images de même hauteur et renvoie leur juxtaposition horizontale.

$$b_1 \mid b_2$$

4. Écrire une fonction

```
juxtapose_v(b1: list[list[list[int]]],
            b2: list[list[list[int]]]) -> list[list[list[int]]]
```

qui prend deux images de même largeur et renvoie leur juxtaposition verticale.

$$\frac{b_1}{b_2}$$

5. Écrire une fonction

```
assemble(b1: list[list[list[int]]],
         b2: list[list[list[int]]],
         b3: list[list[list[int]]],
         b4: list[list[list[int]]]) -> list[list[list[int]]]
```

qui prend 4 images de dimensions compatibles et renvoie l'image

$$\begin{array}{c|c} b_1 & b_2 \\ \hline b_3 & b_4 \end{array}$$

6. Écrire une fonction récursive

```
damier_rec(n: int, k: int,
           c1: list[int],
           c2: list[int]) -> list[list[list[int]]]
```

qui construit un damier de 2^k blocs par ligne, chaque bloc étant de taille $2^n \times 2^n$.
FFFF `c1` et `c2` sont des pixels colorés (par exemple un pixel rouge et un vert).
FFF question mal posée : préciser que le damier est carré, que les blocs de taille $2^n \times 2^n$ sont alternativement d'une couleur et de l'autre puis qu' il y a dans une ligne de blocs 2^k blocs d'une couleur et 2^k blocs d'une autre.

7. En déduire une fonction `damier(n, k)` construisant l'image demandée.

```
#2^2 * 2^2 blocs de 2^3 * 2^3 pixels
img_damier = damier(3, 2)
plt.imshow(img_damier)
plt.axis("off")
plt.show()
```

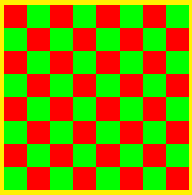

FIGURE 3 – Affichage du damier décrit par `img_damier`

Solution. Code

```
import numpy as np

def bloc(couleur: list[int], k: int) -> list[list[list[int]]]:
    taille = 2**k
    res = []
    for i in range(taille):
        ligne = []
        for j in range(taille):
            ligne.append(couleur[:])# pour ne pas avoir d'adresse commune
        res.append(ligne)
    return res

def juxtapose_h(b1: list[list[list[int]]],
                b2: list[list[list[int]]]) -> list[list[list[int]]]:
```

```python
    H = len(b1)
    res = []
    for i in range(H):
        res.append(b1[i] + b2[i])#aucun risque d'adresse commune
    return res

def juxtapose_v(b1: list[list[list[int]]],
                b2: list[list[list[int]]]) -> list[list[list[int]]]:#V1
    return b1 + b2 #avec partage d'adresses de lignes!

def copy(b):
    return [[[v for v in p] for p in ligne]  for ligne in b]

def juxtapose_v(b1: list[list[list[int]]],
                b2: list[list[list[int]]]) -> list[list[list[int]]]:#V2
    return copy(b1) + copy(b2) # sans partage d'adresse


def assemble(b1: list[list[list[int]]],
             b2: list[list[list[int]]],
             b3: list[list[list[int]]],
             b4: list[list[list[int]]]) -> list[list[list[int]]]:
    return juxtapose_v(juxtapose_h(b1, b2), juxtapose_h(b3, b4))

def damier_rec(n: int, k: int,
               c1: list[int],
               c2: list[int]) -> list[list[list[int]]]:
    if k == 0:
        b1, b2, b3, b4 = bloc(c1, n), bloc(c2, n), bloc(c2, n), bloc(c1, n)
    else:
        b1, b2, b3, b4 = [damier_rec(n, k - 1, c1, c2) for _ in range(4)]
    return assemble(b1, b2, b3, b4)

def damier(n: int, k: int) -> np.ndarray:
    rouge = [255, 0, 0]
    vert = [0, 255, 0]
    return np.array(damier_rec(n, k, rouge, vert), dtype=np.uint8)
```

□

2.2 Images en niveaux de gris

Nous souhaitons maintenant transformer une image couleur en image en niveaux de gris (ces derniers sont plus faciles à manipuler pour les actions que nous souhaitons effectuer ensuite). Nous partons donc d'un tableau multidimensionnel `numpy` d'entiers 8 bits et nous le transformons pour la commodité des calculs en liste de listes d'entiers `Python` (cette transformation est à écrire). Au moment de faire l'affichage de la liste obtenue, nous la retransformons en tableau `numpy` d'entiers 8 bits (cette transformation n'est pas à écrire : elle est donnée).

Question 18.

On souhaite transformer une image en *niveaux de gris*.

Une méthode simple consiste à remplacer un pixel de composantes (R, G, B) par la moyenne $(R + G + B)/3$. Cependant, cette méthode n'est pas très pertinente car l'il humain est plus sensible au vert qu'au rouge, et plus au rouge qu'au bleu.

On utilise donc la pondération suivante :

$$0{,}299R + 0{,}587G + 0{,}114B.$$

Le résultat s obtenu est un entier 8 bits. Il est ensuite converti en un entier `Python` grâce à `int(s)`.

Écrire une fonction `gris(img: np.ndarray) -> list[list[int]]` qui à une image `img` (tableau de taille $H \times L \times 3$ à valeurs entières dans $[\![0, 255]\!]$) associe une image en niveaux de gris, c'est à dire une liste d'entiers de taille $H \times L$.

Solution. Un pixel (R, G, B) devient g avec

$$g = \lfloor 0{,}299R + 0{,}587G + 0{,}114B \rfloor.$$

```python
def gris(img):
    H, L, _ = img.shape

    res = []

    for i in range(H):
        ligne = []
        for j in range(L):
            R, G, B = img[i][j]

            g = int(0.299 * R + 0.587 * G + 0.114 * B)

            ligne.append(g)  # un seul entier !

        res.append(ligne)

    return res  # plus de np.array ici
```

□

La figure 4 donne un aperçu de l'image obtenue en passant en niveau de gris.

```python
imggris = gris(img) # liste de listes
#conversion en tableau numpy :
imggris_np = np.array(imggris,dtype=np.uint8)
#on précise la colormap cmap="gray" :
# l'image est en niveau de gris
plt.imshow(imggris_np, cmap="gray")
plt.axis("off")
mpimg.imsave("astronaut_gris.png",imggris_np)
plt.show()
```

FIGURE 4 – La photo de l'astronaute en niveau de gris

On considère désormais une image en niveaux de gris, représentée par un tableau I de taille $H \times L$, où $I(i, j)$ désigne le niveau de gris du pixel situé à la ligne i et à la colonne j.

On appelle *convolution* le procédé consistant à transformer l'image en appliquant un *filtre* (ou *noyau*) donné par une matrice K de taille $(2k + 1) \times (2k + 1)$.

Pour chaque pixel (i, j), on remplace $I(i, j)$ par une combinaison linéaire des pixels voisins, pondérée par les coefficients du noyau. Plus précisément, la nouvelle image J est définie par :

$$J(i, j) = \sum_{u=-k}^{k} \sum_{v=-k}^{k} K(u, v)\, I(i + u, j + v).$$

Autrement dit, on centre le noyau sur le pixel (i, j), on multiplie chaque coefficient du noyau avec le pixel correspondant, puis on fait la somme des résultats obtenus.

Selon le choix du noyau K, la convolution permet de réaliser différents traitements sur l'image, comme le flou, l'accentuation ou la détection de contours.

La définition précédente pose un problème pour les pixels situés près du bord de l'image, car certains indices $i+u$ ou $j+v$ peuvent sortir du domaine de définition.

On choisit ici de prolonger l'image par des zéros : on suppose que tout pixel situé en dehors de l'image a une valeur nulle.

Ainsi, la formule de convolution reste valable pour tout pixel (i,j), en convenant que $I(i+u,j+v)=0$ dès que l'indice $(i+u,j+v)$ est en dehors de l'image.

Ce choix permet de conserver une image de même taille tout en simplifiant l'implémentation.

Question 19.

On souhaite écrire une fonction calculant la somme des produits des coefficients correspondants de deux matrices carrées de même taille.

Écrire une fonction `somme_produit(A: list[list[int]], B: list[list[float]]) -> float` qui prend en argument deux matrices carrées de même taille à coefficients entiers pour la première et flottants pour la seconde et renvoie la somme des produits de leurs coefficients correspondants.

Code

```
A=[[1,2],[-1,0]]
B=[[1.5,2.5],[0.5,1.]]
somme_produit(A,B)
```

Rendu

```
6.0
```

Solution. Code

```
def somme_produit(A, B):
    n = len(A)
    s = 0

    for i in range(n):
        for j in range(n):
            s += A[i][j] * B[i][j]

    return s
```

□

Question 20.

On souhaite extraire, autour d'un pixel donné, une matrice carrée de taille impaire en prolongeant l'image par des zéros.

Écrire une fonction `matrice_autour(img: list[list[int]], i: int, j: int, k: int) -> list[list[int]]` qui prend en argument une image en niveaux de gris `img`, deux entiers `i` et `j`, ainsi qu'un entier impair `k`, et renvoie la matrice carrée de taille `k` centrée sur le pixel de coordonnées `(i,j)`.

Si certains pixels nécessaires sont en dehors de l'image, on les remplace par des zéros.

Solution. Code

```
def matrice_autour(img, i, j, k):
    H = len(img)
    L = len(img[0])
    r = k // 2
    res = []

    for u in range(i - r, i + r + 1):
        ligne = []
```

```
        for v in range(j - r, j + r + 1):
            if 0 <= u < H and 0 <= v < L:
                ligne.append(img[u][v])
            else:
                ligne.append(0)
        res.append(ligne)

    return res
```

Code

```
A = [
    [10, 20, 30],
    [40, 50, 60],
    [70, 80, 90]
]
# On regarde autour du pixel
# central (1,1) avec k = 3
print(matrice_autour(A, 1, 1, 3))
# Pb des bords
print(matrice_autour(A, 0, 0, 3))
```

Rendu

```
[[10, 20, 30], [40, 50, 60], [70, 80, 90]]
[[0, 0, 0], [0, 10, 20], [0, 40, 50]]
```

□

Question 21.

Après convolution, certaines valeurs peuvent être en dehors de l'intervalle $[\![0, 255]\!]$.
Écrire une fonction `clip(x: int) -> int` qui prend en argument un entier `x` et renvoie :
— 0 si $x < 0$;
— x si $0 \leq x \leq 255$;
— 255 si $x > 255$.

Solution. Code

```
def clip(x):
    if x < 0:
        return 0
    elif x > 255:
        return 255
    else:
        return x
```

□

Question 22.

On souhaite appliquer une fonction à tous les pixels d'une image en niveaux de gris.
Écrire une fonction `map_image(img: list[list[int]], f: function) -> list[list[int]]` qui prend en argument une image `img` et une fonction `f`, et renvoie l'image obtenue en appliquant `f` à chaque pixel de `img`.

Solution. Code

```python
def map_image(img, f):
    H = len(img)
    L = len(img[0])
    res = []

    for i in range(H):
        ligne = []
        for j in range(L):
            ligne.append(f(img[i][j]))
        res.append(ligne)

    return res
```

□

Question 23.

On souhaite maintenant appliquer une convolution à une image en niveaux de gris.

Écrire une fonction `convolution(img: list[list[int]], noyau: list[list[float]]) -> list[list[int]]` qui prend en argument une matrice représentant une image en niveaux de gris `img` et un noyau carré `noyau` de taille impaire, et renvoie l'image obtenue par convolution avec prolongement par zéro. Le résultat est donc une matrice d'entiers entre 0 et 255.

On rappelle que, pour chaque pixel, la convolution s'obtient en calculant la somme des produits des coefficients du noyau avec ceux de la matrice extraite autour de ce pixel.

Solution. Code

```python
def convolution(img, noyau):
    H = len(img)
    L = len(img[0])
    k = len(noyau)
    res = []

    for i in range(H):
        ligne = []
        for j in range(L):
            bloc = matrice_autour(img, i, j, k)
            val = somme_produit(bloc, noyau)
            ligne.append(int(val))
        res.append(ligne)

    return res
```

□

Un noyau de floutage (ou *filtre de moyenne*) est donné par :

$$K = \frac{1}{25}\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

Ce noyau remplace chaque pixel par une moyenne pondéreé des pixels situés dans son voisinage. Il permet de lisser l'image et de réduire les variations locales, ce qui a pour effet de rendre l'image plus floue.

Plus généralement, un noyau de floutage effectue une moyenne pondérée des pixels voisins. la figure 5 donne un apperçu de l'image obtenue.

```python
# noyau de floutage
K = (1/25)* np.ones((5,5))
flou = convolution(imggris,K)
flou = map_image(flou,clip)
flou = np.array(flou,dtype=np.uint8)
plt.imshow(flou, cmap="gray")
plt.axis("off")
plt.show()
```

FIGURE 5 – La photo de l'astronaute floutée en niveau de gris

Pour détecter les contours d'une image, on peut utiliser les *filtres de Sobel*. Il s'agit de deux noyaux de convolution de taille 3×3.

Le premier noyau est

$$K_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}.$$

Il met en évidence les variations du niveau de gris de gauche à droite. Il permet donc de détecter les *contours verticaux*.

Le second noyau est

$$K_y = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}.$$

Il met en évidence les variations du niveau de gris de haut en bas. Il permet donc de détecter les *contours horizontaux*.

On applique ces deux noyaux séparément à l'image : si G_x et G_y désignent les images obtenues par convolution avec K_x et K_y, on peut alors obtenir une image des contours en combinant ces deux résultats.

Question 24.

Écrire une fonction `sobel(img: list[list[int]]) -> list[list[int]]` qui prend en argument une image en niveaux de gris `img` et renvoie l'image obtenue par la méthode de Sobel.

On utilise les noyaux

$$K_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \quad \text{et} \quad K_y = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix},$$

et on combine les deux convolutions en posant, pour chaque pixel,

$$G = |G_x| + |G_y|.$$

Solution. Code

```python
def sobel(img):
    Kx = [
        [-1, 0, 1],
        [-2, 0, 2],
        [-1, 0, 1]
    ]

    Ky = [
        [-1, -2, -1],
        [0, 0, 0],
        [1, 2, 1]
    ]
```

```python

    gx = convolution(img, Kx)
    gy = convolution(img, Ky)

    H = len(img)
    L = len(img[0])
    res = []

    for i in range(H):
        ligne = []
        for j in range(L):
            val = abs(gx[i][j]) + abs(gy[i][j])
            ligne.append(val)
        res.append(ligne)

    return res
```

□

La figure 6 montre bien les contours des éléments de la photographie de l'astronaute.

```python
contour = sobel(imggris)
contour = map_image(contour, clip)
contour = np.array(contour,dtype=np.uint8)
plt.imshow(contour, cmap="gray")
plt.axis("off")
plt.show()
```

FIGURE 6 – Les contours dans la photo de l'astronaute en niveau de gris

Question 25.

On souhaite transformer une image en une image binaire (noir et blanc) en ne conservant que les pixels dont la valeur est supérieure à un certain seuil.

1. Écrire une fonction `seuil(x: int, s: int) -> int` qui renvoie :
   — 255 si $x \geq s$;
   — 0 sinon.
2. En déduire une fonction `seuillage(img: list[list[int]], s: int) -> list[list[int]]` qui applique ce seuillage à toute l'image.

Solution. Code

```python
def seuil(x, s):#V1
    if x >= s:
        return 255
    else:
        return 0

def seuil(x, s):#V2 : en utilisant la coercion de type
    return 255 * (x >= s)


def seuillage(img, s):
    return map_image(img, lambda x: seuil(x, s))
```

□

```python
K = (1/25)* np.ones((5,5))
flou = convolution(imggris,K)
sobflou = sobel(flou)
contour=seuillage(sobflou,140)
contour = np.array(contour,dtype=np.uint8)
plt.imshow(contour, cmap="gray")
plt.axis("off")
plt.show()
```

FIGURE 7 – Les contours après floutage, filtre de Sobel et seuillage à 140

Pour aller plus loin Dans ce devoir, nous avons mis en place une chaîne simple de traitement d'image : passage en niveaux de gris, lissage (floutage), calcul du gradient à l'aide des filtres de Sobel, puis seuillage.

Cette méthode permet de mettre en évidence les contours de l'image. Cependant, les contours obtenus sont souvent épais, bruités et pas toujours fermés.

Pour améliorer le résultat, on peut affiner cette approche en combinant plus finement ces étapes, notamment en améliorant le lissage et le seuillage.

Des méthodes plus avancées, comme l'algorithme de Canny, permettent ainsi d'obtenir des contours fins, continus et bien localisés.