

AIV1 - TP02  
Acquisition d'images couleur  
et dématricage

Suliac Lavenant et Antoine Nollet

25 January 2022

# Table des matières

<b>1</b>	<b>Interprétation et simulation d'une image CFA</b>	<b>3</b>
1.1	Interprétation d'une image CFA . . . . .	3
1.2	Simulation d'une image CFA à partir d'une image couleur . . . . .	4
1.3	Exercice optionnel . . . . .	5
<b>2</b>	<b>Dématriçage par interpolation bilinéaire</b>	<b>6</b>
2.1	Présentation . . . . .	6
2.1.1	$H^R$ et $H^B$ . . . . .	6
2.1.2	$H^G$ . . . . .	7
2.2	Programmation du dématriçage . . . . .	7
2.2.1	génération le plan k du canal k . . . . .	7
2.2.2	interpolation bilinéaire par convolution . . . . .	8
<b>3</b>	<b>Dématriçage basé sur l'estimation locale d'un gradient</b>	<b>10</b>
3.1	Estimation du plan G . . . . .	10
3.2	Évaluation de la qualité de l'image estimée . . . . .	11

# 1 Interprétation et simulation d'une image CFA

## 1.1 Interprétation d'une image CFA



FIGURE 1 – Image lighthouse et une image de son CFA d aprs Bayer

La question est maintenant la suivante : quelle disposition du filtre CFA a été utilisé ?

En effet, avec le filtre de Bayer, qui utilise deux fois plus de vert que de rouge ou de bleu, nous avons 4 dispositions possibles (chacun des noms de dispositions correspond à la ligne centrale du carré 3x3 de pixels située en haut à gauche de l'image) :

- GRG
- GBG
- RGR
- BGB

Les valeurs RGB du carré 3x3 de pixels située en haut à gauche de l'image de base, sous forme de (x,y), relevées avec ImageJ, ainsi que les valeurs de niveau de gris du même carré de pixels mais de l'image CFA, aussi relevées avec ImageJ, sont représentées dans le tableau suivant :

coordonnée du pixel	composition RGB de lighthouse du pixel	valeur niveau de gris de lighthouse_cfa du pixel
(0,0)	75, 93, 94	94
(1,0)	78, 95, 104	95
(2,0)	76, 92, 107	107
(0,1)	75, 93, 94	93
(1,1)	76, 93, 102	76
(2,1)	74, 90, 104	90
(0,2)	78, 94, 96	96
(1,2)	80, 94, 102	94
(2,2)	77, 90, 106	106

En analysant les valeurs du précédent tableau, on remarque que les valeurs de niveau de gris de lighthouse\_cfa correspondent avec certaines valeurs de compositions de lighthouse. Voici ce qui est visuellement obtenu si nous faisons ce lien :

B	G	B
G	R	G
B	G	B

Ainsi, on remarque que cette disposition correspond à la disposition GRG du filtre CFA de Bayer. On peut donc conclure que c'est la disposition GRG qui est ici utilisée.

## 1.2 Simulation d'une image CFA à partir d'une image couleur

On complète donc la fonction pour retourner un tableau numpy avec les mêmes largeurs et hauteurs que du tableau d'origine. On attribue à chaque emplacement du tableau la valeur de la composante correspondante de l'image de base (d'après GRBG). On obtient le code suivant :

```

1 def cfa_image(img, cfa="GRBG"):
2     """
3         Return the CFA image from color image <img> (numpy array) with given <cfa>
4     """
5     imgCfa = np.zeros((img.shape[0],img.shape[1]), dtype=np.uint8)
6     imgCfa[:,::2,::2] = img[:,::2,::2,1]#vert 1
7     imgCfa[:,::2,1::2] = img[:,::2,1::2,0]#rouge
8     imgCfa[1::2,::2] = img[1::2,::2,2]#bleu
9     imgCfa[1::2,1::2] = img[1::2,1::2,1]#vert 2
10
11     return imgCfa

```

On applique maintenant ce code sur l'image lighthouse et on obtient l'image CFA suivante :



FIGURE 2 – CFA produit avec le code précédent sur l'image lighthouse



FIGURE 3 – zoom de l'image précédente

En zoomant un peu, on peut voir de gros contraste entre des pixels adjacents, on a donc bien obtenu notre image CFA. Les pixels blancs, donc de haute valeur, correspondent à la composante rouge sur la boué.

### 1.3 Exercice optionnel

Afin de pouvoir réutiliser notre précédente fonction avec différentes dispositions du CFA de Bayer, voici ce qui est proposé :

```

1 COLORS = { "R": 0, "G": 1, "B": 2}
2
3 def cfa_image(img, cfa="GRBG"):
4     """
5         Return the CFA image from color image <img> (numpy array) with given <cfa>
6     """
7     imgCfa = np.zeros((img.shape[0],img.shape[1]), dtype=np.uint8)
8     imgCfa[:,::2,::2] = img[:,::2,::2, COLORS[cfa[0]]]
9     imgCfa[:,::2,1::2] = img[:,::2,1::2, COLORS[cfa[1]]]
10    imgCfa[1::2,::2] = img[1::2,::2, COLORS[cfa[2]]]
11    imgCfa[1::2,1::2] = img[1::2,1::2, COLORS[cfa[3]]]
12
13    return imgCfa

```

Le dictionnaire COLORS en constante correspond aux indices auquels se trouvent les valeurs R, G et B dans la liste de taille 3 contenant ces trois valeurs (exemple la valeur rouge "R" se trouve toujours en indice 0) (car dans le tableau numpy d'entrée, les valeurs ont 3 composantes et sont donc des listes de taille 3). Ainsi, on n'utilise plus d'entiers directement dans la fonction de manière à avoir ce qui est souhaité. Il faut désormais entrer une chaîne de caractère de type "CCCC" où les C ont une valeur inclus dans "R","G","B" afin d'appliquer une certaine disposition de Bayer.

## 2 Dématriçage par interpolation bilinéaire

### 2.1 Présentation

#### 2.1.1 $H^R$ et $H^B$

Dans le cas du dématriçage du rouge ou du bleu par son filtre de convolution :

$$H^R = H^B = \frac{1}{4} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Il y a 3 cas d'application :

0	0	0
R	0	R
0	0	0

Dans le cas ci-dessus, nous appliquons le calcul suivant :

$$\begin{bmatrix} 0 & 0 & 0 \\ R & 0 & R \\ 0 & 0 & 0 \end{bmatrix} * H^R = \begin{bmatrix} 0 & 0 & 0 \\ R & 0 & R \\ 0 & 0 & 0 \end{bmatrix} * \frac{1}{4} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} = \frac{1*0+2*0+1*0+2*R+4*0+2*R+1*0+2*0+1*0}{4} = \frac{2R+2R}{4} = R$$

Nous avons donc dans ce cas ci, une valeur centrale égale à R, ce qui est l'objectif attendu par l'utilisation du masque  $H^R$ .

R	0	R
0	0	0
R	0	R

Dans le cas ci-dessus, nous appliquons le calcul suivant :

$$\begin{bmatrix} R & 0 & R \\ 0 & 0 & 0 \\ R & 0 & R \end{bmatrix} * H^R = \begin{bmatrix} R & 0 & R \\ 0 & 0 & 0 \\ R & 0 & R \end{bmatrix} * \frac{1}{4} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} = \frac{1*R+2*0+1*R+2*0+4*0+2*0+1*R+2*0+1*R}{4} = \frac{R+R+R+R}{4} = R$$

Nous avons donc dans ce cas ci, une valeur centrale égale à R, ce qui est l'objectif attendu par l'utilisation du masque  $H^R$ .

0	0	0
0	R	0
0	0	0

Dans le cas ci-dessus, nous appliquons le calcul suivant :

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & R & 0 \\ 0 & 0 & 0 \end{bmatrix} * H^R = \begin{bmatrix} 0 & 0 & 0 \\ 0 & R & 0 \\ 0 & 0 & 0 \end{bmatrix} * \frac{1}{4} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} = \frac{1*0+2*0+1*0+2*0+4*R+2*0+1*0+2*0+1*0}{4} = \frac{4R}{4} = R$$

Nous avons donc dans ce cas ci, une valeur centrale égale à R, ce qui est l'objectif attendu par l'utilisation du masque  $H^R$ .

Ainsi, chaque pixel final aura sa valeur attendu en rouge R (car à chaque fois nous faisons la moyenne des valeurs de rouge autour). La logique restera la même pour l'utilisation du masque  $H^B$  (même cas d'application) et donc chaque pixel aura également sa valeur attendu en bleu B (car à chaque fois nous faisons la moyenne des valeurs de bleu autour).

## 2.1.2 $H^G$

Dans le cas du dématriçage du vert par son filtre de convolution :

$$H^G = \frac{1}{4} \begin{bmatrix} 0 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Il y a deux cas d'application :

Le premier cas :	<table border="1" style="border-collapse: collapse; text-align: center;"> <tr><td>G</td><td>0</td><td>G</td></tr> <tr><td>0</td><td>G</td><td>0</td></tr> <tr><td>G</td><td>0</td><td>G</td></tr> </table>	G	0	G	0	G	0	G	0	G
G	0	G								
0	G	0								
G	0	G								

Dans ce premier cas on ignore les diagonales (0) car on a déjà dans ce cas un G sur le pixel traité et on veux le conserver sans qu'il soit alterer par ses voisins. les 1 sur les cotés et au dessus dessous sont multiplié par 0 donc n'impacte pas le resultat. Ensuite on a le 4 au centre de la matrice pour que notre valeur de base "survive" a la division par 4.

$$\begin{bmatrix} G & 0 & G \\ 0 & G & 0 \\ G & 0 & G \end{bmatrix} * H^G = \begin{bmatrix} G & 0 & G \\ 0 & G & 0 \\ G & 0 & G \end{bmatrix} * \frac{1}{4} \begin{bmatrix} 0 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 0 \end{bmatrix} = \frac{0*G+1*0+0*G+1*0+4*G+1*0+0*G+1*0+0*G}{4} = \frac{4G}{4} = G$$

Le deuxième cas :	<table border="1" style="border-collapse: collapse; text-align: center;"> <tr><td>0</td><td>G</td><td>0</td></tr> <tr><td>G</td><td>0</td><td>G</td></tr> <tr><td>0</td><td>G</td><td>0</td></tr> </table>	0	G	0	G	0	G	0	G	0
0	G	0								
G	0	G								
0	G	0								

Dans ce deuxième cas on ignore aussi les diagonales car elles sont de valeur 0 (pas de valeur). La valeur centrale, malgrès le coefficient 4, reste 0 (comme on a pas de valeur). La valeur du pixel va donc venir de ses voisins à droite, à gauche, en bas et au dessus (1). C'est une moyenne de ces valeurs qui va devenir la nouvelle valeur du pixel (4 coefficient 1 potentiellement non nul qui ce font ensuite divisé par 4).

$$\begin{bmatrix} 0 & G & 0 \\ G & 0 & G \\ 0 & G & 0 \end{bmatrix} * H^G = \begin{bmatrix} 0 & G & 0 \\ G & 0 & G \\ 0 & G & 0 \end{bmatrix} * \frac{1}{4} \begin{bmatrix} 0 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 0 \end{bmatrix} = \frac{0*0+1*G+0*0+1*G+4*0+1*G+0*0+1*G+0*0}{4} = \frac{G+G+G+G}{4} = \frac{4G}{4} = G$$

## 2.2 Programmation du dématriçage

### 2.2.1 génération le plan k du canal k

```

1 def cfa_samples(cfa_img, channel, cfa="GRBG"):
2     """
3         Return the CFA samples of <channel> ("R".."B") from CFA image <cfa_img> (numpy array) with given <cfa>
4     """
5     cfa_img_sample = np.zeros(cfa_img.shape, dtype=np.uint8)
6
7     if (channel == "G"):
8         cfa_img_sample[1::2, 1::2] = cfa_img[1::2, 1::2]
9         cfa_img_sample[::2, ::2] = cfa_img[::2, ::2]
10
11    else:
12        if (channel == "R"):
13            cfa_img_sample[::2, 1::2] = cfa_img[::2, 1::2]
14        elif (channel == "B"):
15            cfa_img_sample[1::2, ::2] = cfa_img[1::2, ::2]
16
17    return cfa_img_sample

```

Le programme récupère les échantillons de l'image cfa pour renvoyer une image avec les valeurs de la composante k uniquement.

L'exécution de ce programme ci-dessus nous donne les images des composantes suivantes :



FIGURE 4 – Canal rouge, vert et bleu extrait de la CFA de l'image lighthouse

### 2.2.2 interpolation bilinéaire par convolution

```

1  def demat_bilin(img_name, cfa="GRBG", show=False, write=False):
2      """
3          Return the image estimated by bilinear interpolation from <img_name> (numpy array) with given <cfa>
4      """
5      img = plt.imread("lighthouse.png") * 255
6      img = np.array(img, dtype=np.uint8)
7
8      cfa_img = cfa_image(img, cfa)
9      cfa_img_R = cfa_samples(cfa_img, "R", cfa)
10     cfa_img_G = cfa_samples(cfa_img, "G", cfa)
11     cfa_img_B = cfa_samples(cfa_img, "B", cfa)
12
13     cfa_img_R_dest = cv2.filter2D(cfa_img_R, -1, HRB)
14     cfa_img_G_dest = cv2.filter2D(cfa_img_G, -1, HG)
15     cfa_img_B_dest = cv2.filter2D(cfa_img_B, -1, HRB)
16
17     img_dest = np.zeros(img.shape, dtype=np.uint8)
18     img_dest[:, :, 0] = cfa_img_R_dest
19     img_dest[:, :, 1] = cfa_img_G_dest
20     img_dest[:, :, 2] = cfa_img_B_dest
21
22     if(show):
23         plt.figure()
24         plt.imshow(img_dest)
25         plt.show()
26
27     if(write):
28         plt.imsave("image_couleur_estime.png", img_dest)
29
30     return img_dest

```

Ce programme transforme une image couleur en image son image CFA puis extrait les différents canaux de cette image pour ensuite leur appliquer les filtres vu précédemment (pour remplacer les trous de l'image) pour finir par créer une image avec ces composantes.



FIGURE 5 – Image couleur estime du CFA de l'image lighthouse

On remarque un effet de "fausses couleurs" sur cette image obtenue, notamment sur les barrières au fond, qui proviennent de l'écart important entre la couleur de référence et celle estimée. Et l'effet "fermeture éclair" est aussi visible par exemple en haut du phare.

### 3 Dématriçage basé sur l'estimation locale d'un gradient

#### 3.1 Estimation du plan G

Nous avons implémenté HA97 en python pour l'appliquer sur notre image avec le code suivant :

```
1 def est_G_ha(cfa_img, cfa="GRBG"):
2     """ Return the G plane estimated by H&A algorithm from <cfa_img> (numpy array) with given <cfa>
3     """
4     R = cfa_samples(cfa_img, "R", cfa)
5     G = cfa_samples(cfa_img, "G", cfa)
6     B = cfa_samples(cfa_img, "B", cfa)
7
8     gPlane = np.zeros(cfa_img.shape, dtype=np.uint8)
9     gPlane += G #ajoute a notre image verte les composant vert extrait du cfa
10
11    #####calcul GRG
12    GRG = np.zeros(cfa_img.shape, dtype=np.uint8)
13    for y in range(3,cfa_img.shape[1]-2,2):
14        for x in range(2,cfa_img.shape[0]-2,2):
15
16            deltaX = abs(G[x-1,y]-G[x+1,y])+abs(2*R[x,y]-R[x-2,y]-R[x+2,y])
17            deltaY = abs(G[x,y-1]-G[x,y+1])+abs(2*R[x,y]-R[x,y-2]-R[x,y+2])
18
19            if(deltaX < deltaY):
20                GRG[x,y]=(G[x-1,y]+G[x+1,y])/2+(2*R[x,y]-R[x-2,y]-R[x+2,y])/4
21            elif(deltaX > deltaY):
22                GRG[x,y]=(G[x,y-1]+G[x,y+1])/2+(2*R[x,y]-R[x,y-2]-R[x,y+2])/4
23            elif(deltaX == deltaY):
24                GRG[x,y]=((G[x,y-1]+G[x,y+1]+G[x-1,y]+G[x+1,y])/4)+(4*R[x,y]-R[x,y-2]-R[x,y+2]-R[x-2,y]-R[x+2,y])/4
25
26    gPlane += GRG
27
28    #####calcul GBG
29    GBG = np.zeros(cfa_img.shape, dtype=np.uint8)
30    for y in range(2,cfa_img.shape[1]-2,2):
31        for x in range(3,cfa_img.shape[0]-2,2):
32
33            deltaX = abs(G[x-1,y]-G[x+1,y])+abs(2*B[x,y]-B[x-2,y]-B[x+2,y])
34            deltaY = abs(G[x,y-1]-G[x,y+1])+abs(2*B[x,y]-B[x,y-2]-B[x,y+2])
35
36            if(deltaX < deltaY):
37                GBG[x,y]=(G[x-1,y]+G[x+1,y])/2+(2*B[x,y]-B[x-2,y]-B[x+2,y])/4
38            elif(deltaX > deltaY):
39                GBG[x,y]=(G[x,y-1]+G[x,y+1])/2+(2*B[x,y]-B[x,y-2]-B[x,y+2])/4
40            elif(deltaX == deltaY):
41                GBG[x,y]=((G[x,y-1]+G[x,y+1]+G[x-1,y]+G[x+1,y])/4)+(4*B[x,y]-B[x,y-2]-B[x,y+2]-B[x-2,y]-B[x+2,y])/4
42
43    gPlane += GBG
44    return gPlane
```

Comme on peut le voir sur l'image suivante ce n'est pas un succès et après de longues recherches et de nombreux tests nous n'avons pas trouvé ce problème...



FIGURE 6 – Image obtenue par HA97

### 3.2 Évaluation de la qualité de l'image estimée



FIGURE 7 – Comparaison des plan G par le gradient et par interpolation

En comparant les deux images on voit pas mal de différences qui sont confirmées par le PSNR de 13.797253121331234.



FIGURE 8 – Comparaison des images couleurs par le gradient et par interpolation avec l'image de base

On peut ainsi conclure, qu'avec un PSNR de 18.038784282394165 entre l'image de base et l'image obtenue par le gradient et avec un PNSR de 28.011787447598376 entre l'image de base et l'image obtenue par interpolation, qu'ici la solution de dématricage par le gradient n'est pas satisfaisante. On gardera l'image ayant un PSNR avec l'image de base le plus grand possible (en pratique l'idéale est environ 40db ).