

1 Introducció

Les imatges estan codificades en arrays 2D (o 3D) de nombres, amb tantes files com píxels té d'alçada la imatge, i tantes columnes com píxels d'amplada té la imatge. Aquests nombres donen la intensitat amb què ha d'emetre el píxel que hi correspon, el 0 és que no emet (negre) i el màxim seria màxima emissió (blanc, p.ex.).

Les imatges en blanc i negre només tenen una capa a la array. És a dir, són matrius¹ planes, a cada píxel hi correspon un sol nombre. En canvi, les imatges en color tenen 3 canals, és a dir, a cada píxel hi corresponen tres nombres. Per exemple, si la imatge està codificada en els canals RGB, el primer nombre donaria la intensitat de vermell (R), el segon en verd (G) i el tercer en blau (B). Això, és clar, afegeix dues noves dimensions a la array, de manera que ara ja no ens ho imaginariem com una matriu plana, sinó com tres capes de matrius planes superposades.

Fins i tot, també es pot afegir un quart canal, *alpha*, que dóna informació sobre la transparència de la imatge. Ara bé, nosaltres no hi treballarem, ens restringirem a 1 o 3 canals.

I com són aquests nombres? Doncs la imatge pot estar codificada en:

1. Integers: són valors enters. Gairebé sempre, són un tipus de enters molt especials: sense signe (*unsigned*) i amb només 8 bits de memòria reservats per ells: `uint8`. Això limita els valors que poden prendre entre 0 i 255. També, és clar, podem trobar-les codificades en enters de 24 o 32 bits. Per exemple, el format JPG codifica la informació en `uint8`.
2. Nombres de coma flotant: són nombres decimals amb un tipus `float64` o `double` que permeten prendre molts més valors que en el cas dels enters. Per exemple, els `float64` donen unes 14 xifres decimals. És molt important que sempre que hi treballem ho fem entre 0 i 1. El format PNG utilitza imatges codificades en decimals.
3. Logical: només accepten dos valors: `True=0` (negre) i `False=1` (blanc). Per tant, només poden crear funcions en blanc i negre, com veurem a l'apartat de binaritzacions.

1.1 Operacions amb arrays

En tant que les arrays no són matrius, operacions com el quadrat d'aquesta array no és un quadrat d'una matriu (producte per ella mateixa), sinó que és el quadrat a cada punt (com si fos una funció discretitzada). Python ja entén $M^{**}2$ com això.

A l'hora de fer operacions, és molt important vigilar quan treballem amb `uint8`. Com que els valors que poden prendre aquests tipus de nombres oscil·len obligatòriament entre 0 i 255, si en qualsevol càlcul intermig fem una suma que superi aquest límit, aleshores cada cop que es superi el 255 es reinicia a 0. És a dir, si fem una suma tal que $155 + 101 = 256$, aleshores el valor no es guardarà com a 256, sinó que reiniciarà més enllà de 255, i donarà $155 + 101 = 0$. Podria passar, per això, que obtinguem un píxel negre on no toca.

Per evitar aquest tipus de problemes, podem optar per treballar sistemàticament des del principi en nombres de coma flotant. L'altra possibilitat és treballar amb nombres `int` en aquest tipus d'operacions (no estan limitats a 256 nombres), però cal vigilar, a l'hora d'acabar el resultat, caldrà normalitzar els nombres al rang $[0, 255]$.

De fet, en operacions com les potències és gairebé imprescindible treballar en nombres decimals $[0, 1]$, ja que serà molt diferent el que obtenim si elevem un 1 al quadrat que si ho fem el 255, encara que després vulguem normalitzar.

1.2 Entrada i sortida d'imatges

Per llegir imatges utilitzarem la funció `matplotlib.pyplot.imread`. L'argument serà la ruta de la imatge que volem llegir, i retornarà un array amb 1 o 3 canals, segons com sigui la imatge. Per exemple, guardem la imatge en una variable com:

```
1     im = imread('imatge.jpg')
```

¹Direm matrius a les arrays 2D per abús de llenguatge, en realitat no són matrius, són estructures amb forma matricial, no actuen com a matrius.

De primeres no sabem com està codificada la imatge. El que fem per saber quina classe de valors estan codificats a la imatge és fer `im.dtype`. L'atribut `im.shape` dóna un array de 2 o 3 valors. 2 vol dir que té un sol canal (alçada per amplada), 3 vol dir que té més canals (alçada per amplada per nº de canals). 1a component fila, 2a component columna. És convenient començar el programa amb:

```
1 nfil    = im.shape[0]
2 ncol    = im.shape[1]
3 ncanals = im.shape[2]
```

Si només hi ha un canal a la array, aleshores la tercera línia donarà error, compte.

Per mostrar una certa array adequada com una imatge utilitzem la funció `matplotlib.pyplot.imshow`.

A l'hora de mostrar les imatges, cal vigilar que els nombres que conté la array estiguin en el rang adequat. Si estem treballant en `uint8` no hi ha problema, perquè sempre es troben entre 0 i 255. Si estem treballant en `int`, com hem dit, han d'estar normalitzats de 0 a 255. Si són nombres de coma flotant, aleshores han d'estar acotats entre 0 i 1. Sinó la funció peta, tot nombre major que 1 ho converteix a 1.

Finalment, podem guardar la array com a imatge en l'ordinador amb la funció `matplotlib.pyplot.imsave`.

Accés a la informació dels píxels de l'array

Com ens adrecem a un píxel particular? Doncs si té un sol canal serà tan fàcil com `im[10,75]`. Si hi ha més d'un canal, per cridar un cert píxel cal ficar una tercera component (0 R, 1 G, 2B, 3α). Com podem agafar trossos de la imatge? Doncs utilitzant el *slicing*, per exemple `im[principi:final(: salt)]`. Si fem salt de 1 en 1 no fiquem el salt, si comencem a 0 no fiquem el principi, si acabem a n-1 no fiquem el final. Com podem dividir la imatge en els diferents canals, p.ex. de 3 canals. Doncs fent: `imR=im[:, :, 0]`. Agafa totes les files i columnes de la imatge, només en el canal 0 (R).

Funcions útils

Algunes funcions útils són `numpy.zeros` omple una array de 0, `numpy.ones` omple una array de 1, `numpy.empty` omple de res (reserva memòria). S'invoquen:

```
1 zeros([numfil, numcol, ...], dtype=...)
```

El `dtype` és el tipus de número que hi ha guardat (la memòria que s'ha de reservar depèn del tipus de número).

2 Manipulació bàsica de imatges (P2)

En la pràctica 2 ens basem en la manipulació bàsica d'una imatge, extraient la informació dels tres colors d'una imatge en color, i l'aplicació d'algunes correccions bàsiques. Podem utilitzar la llibreria `skimage.data` per agafar algunes imatges ja guardades offline en una llibreria. La podem importar amb:

```
1 from skimage.data import astronaut()
```

Si volem cridar la imatge de l'astronauta.

2.1 Canals

Primer de tot, podem guardar la informació dels tres canals en tres matrius diferents, que ara seran 2D. Per exemple, comencem fent:

```
1 im = plt.imread('Ulldret.jpg') #si volem una imatge guardada
2 im = astronaut() #utilitzem la imatge de l'astronauta de la llibreria skimage
3 nfil = im.shape[0]
4 ncol = im.shape[1]
5 print("Les dimensions de la imatge són: {}px x {}px".format(nfil,ncol))
6 print("La informació està codificada en nombres {}".format(im.dtype))
```

```

7
8 imR = im[:, :, 0] #agafem tots els nombres amb el index 0 pel canal R
9 imG = im[:, :, 1] #idem per G (index 1) i B (index 2)
10 imB = im[:, :, 2]

```

Cal anar en compte a l'hora de representar aquestes arrays. La funció `imshow` coloreja per defecte amb un mapa de color (`cmap`) una mica arbitrari si la array que li passem té només 8 bits, o sigui, té només un canal, com és el cas d'aquestes matrius. Per això, cal especificar-li quin mapa de color utilitzar, i el `cmap = 'gray'` és útil: inserta el mateix valor als 3 canals i, per tant, es mostra una imatge en escala de grisos. Aleshores:

```

1 #Representem-ho en escala de grisos
2 plt.figure()
3 plt.subplot(1,3,1)
4 plt.title("Canal R")
5 plt.imshow(imR,cmap='gray')
6 plt.subplot(1,3,2)
7 plt.title("Canal G")
8 plt.imshow(imG,cmap='gray')
9 plt.subplot(1,3,3)
10 plt.title("Canal B")
11 plt.imshow(imB,cmap='gray')

```

Com en la imatge de la P2 de l'ull dret domina molt el vermell, quan extraiem la informació dels canals, la que correspon al R es veu molt més lluminosa. En canvi, les altres dues estan bastant més fosques.

Si el que volem és representar les imatges R, G i B per separat, però cadascuna en el seu color corresponent (sense utilitzar mapes de colors, això és una mica inventar-se colors), el que hem de fer és crear una imatge en color que només contingui un dels tres canals alhora. Per fer-ho, creem una matriu de zeros, i omplim només un dels tres canals amb la informació de cada canal. Fem:

```

1 im0 = np.zeros([nfil,ncol,3],dtype=np.uint8)
2 imRR = np.copy(im0) #guardem una matriu de zeros a on fem la imatge en un color
3 imGG = np.copy(im0)
4 imBB = np.copy(im0)
5
6 imRR[:, :, 0] = np.copy(im[:, :, 0]) #guardem la info del canal R només al index que toca
7 imGG[:, :, 1] = np.copy(im[:, :, 1])
8 imBB[:, :, 2] = np.copy(im[:, :, 2])

```

Amb això aconseguim que Python entengui que la array que li donem correspon a una imatge en color. Però com només conté, per exemple, la informació del vermell, de manera que és veu vermella, sense els altres colors, com volíem.

Notem que hem utilitzat la funció `numpy.copy`. Per copiar una variable `x` a una altra variable `y` podem fer senzillament `y=x` (el qual copia l'adreça de `x` a `y`), en fer això si el valor que hi ha a `x` es sobreescriva, també es sobreescriva a `y`. L'altra manera, més adient, és fer `y=copy(x)`, que senzillament copia el valor que hi ha guardat a `x` i el passa a `y`, sense mirar les adreces.

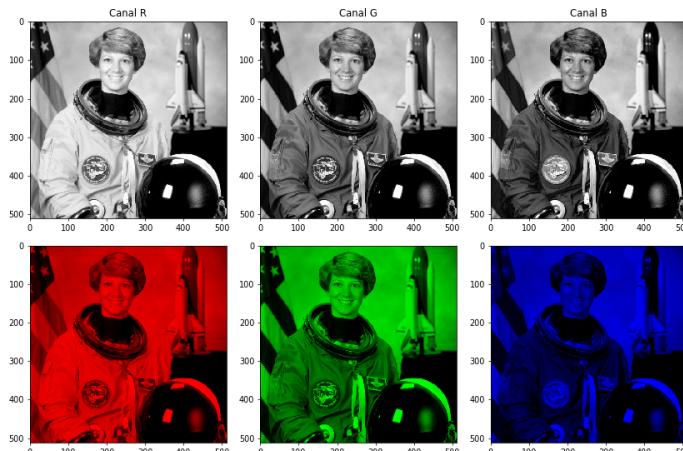


Figure 1: Imatge de l'astronauta en cadascun dels canals. A dalt, el canal extret i mostrat en escala de grisos (`cmap='gray'`), a sota mostrant en el seu color.

2.2 Imatges en escales de gris

La luminància i la mitjana busquen donar la millor imatge en blanc i negre d'una imatge en color donada. La luminància està basada en què els tres colors pesen diferent, ja que les persones no distingim igual de bé tots els colors (per exemple, distingim millor el verd). El màxim de sensibilitat està en el verd (555 nm), la zona dels blaus i vermells es veu pitjor. Aquests pesos de la luminància tenen això en compte.

L'estàndard JPG justament utilitza la luminància en un dels seus tres canals. Enlloc de RGB, empra Luminància, Cromitància I i Cromitància II (és un canvi de base). El canal L va associat purament a l'energia del píxel, mentre que els altres dos Cr sí que van al color. El còdec MPEG (de vídeos) és igual. Ens farà falta treballar en reals per fer els canvis de base (recordem que necessitarem que els nombre estiguin donats entre 0 i 1).

Per fer la mitjana i la luminància, cal anar en compte si treballem amb `uint8`:

```

1 #Fem la imatge en blanc i negre en un array 2D
2 impromig = (np.int_(imR)+np.int_(imG)+np.int_(imB))*0.333 #Si R+G+B>255 tindrem problema
3 imL = 0.299*imR+0.587*imG+0.114*imB #aqui mai superem el 255 en càcul intermid
4
5 #Ho representem forçant un mapa de color en blanc i negre
6 plt.figure()
7 plt.subplot(1,2,1)
8 plt.title("Escala de grisos fent el promig")
9 plt.imshow(impromig,cmap='gray')
10 plt.subplot(1,2,2)
11 plt.title("Escala de grisos fent la luminància")
12 plt.imshow(imL,cmap='gray')
```

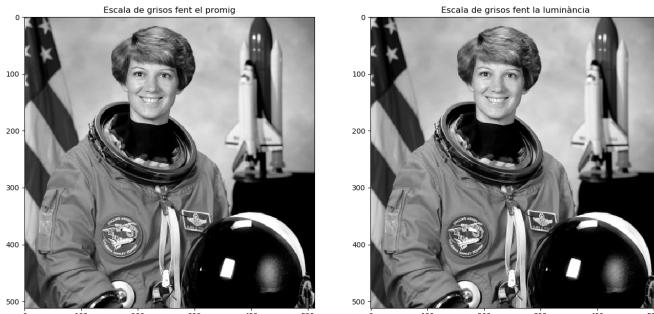


Figure 2: Imatge en escala de grisos mitjançant la mitjana (esquerra), i calculant-ne la luminància (dreta). Com veiem, o més aviat no veiem, la diferència és indistingible.

2.3 Contrast gamma

A l'apartat d) de la P2 treballem amb l'anomenat contrast gamma, que és elevar la imatge a un nombre. És un exemple de Look-Up Table (LUT), que bàsicament és una funció que associa a un determinat valor del nombre un altre, segons una regla, tabulada o formulada. És a dir, a un valor arbitrari x hi associa $x^{\frac{1}{\gamma}}$, essent γ la correcció gamma. Les $\gamma < 1$ ennegreixen la imatge, mentre que les $\gamma > 1$ la il·luminen. Igualment, fa falta utilitzar nombres reals per no anar arrastrant factors de normalització *everywhere*:

```

1 gamma = 0.5
2 #Primer normalitzem la imatge original entre 0 i 1, després fem la potència
3 #Després, perquè vull, la tornem a ficar uint8 entre 0 i 255 per imshow
4 imm = np.uint8(255*(im/255)**(1/gamma))
```

Com a extra de la pràctica, per utilitzar la càmera hem de instal·lar la llibreria opencv.

3 Binarització de imatges (P3)

La binarització és un processament de la imatge que consisteix en donar a cada píxel un de dos valors, 0 o 1. Perquè és important la binarització? Per exemple, les impressores, que només tenen



Figure 3: Diferents contrasts gamma aplicats a la mateixa imatge.

dues opcions: "ficar tinta/no ficar-ne" (0/1), però malgrat això són capaces de fer imatges en escala de grisos. Així, a més, es pot reduir molt l'emmagatzematge necessari per guardar la imatge, en tant que passem de `uint8` a un true/false.

En general, qualsevol LUT es fa canal a canal, no es fa als 3 alhora. En el cas de la binarització, no té cap sentit fer-ho canal a canal, i després fer 0/1, perquè en tornar a recombinar la imatge que surt seguirà sent en color. Per tant, treballarem sobre 8 bits, és a dir, ha de ser una fotografia en blanc i negre! Almenys de moment.

3.1 Binarització amb llindar global

Una manera de fer la binarització és fer-ho globalment:

1. Qualsevol píxel amb una certa intensitat > 128 (o qualsevol altre número), li donem 1 (blanc)
2. Si no, 0 (negre).

A l'hora de programar, ho podem fer amb:

```

1 #Comencem aplicant un llindar global (threshold) th a tota la imatge
2 th = 0.5
3
4 """Utilitzant un operador lògic >, fem que tots els valors per sota del th prenguin
5 valor False (que en passar a float és un 0), i els per sobre un True (= 1)
6 Així tota la funció pren valors 0 o 1 segons si el píxel és > o < que th."""
7 imbin0 = np.float64(imL>th) #imL>th és un array logic, fem-lo float

```

On `imL` és una array de 8 bits que guarda la imatge en escala de grisos segons el mètode de la luminància. Igual que hem fet `th=0.5`, podríem haver pres qualsevol altre valor segons com sigui la imatge i quina intenció tinguem.

3.2 Binarització amb llindar variable

Ara bé, sovint les imatges tenen una lluminositat variable (p.ex. en il·luminació rasant - una bombeta-, si fem llindar fixe la zona propera a la bombeta tindrà molta més llum i obtindrà 1 segur, i la resta 0, perdrem molta informació).

Una manera més correcta és utilitzar un llindar variable, que sigui local, en funció d'un criteri estadístic. Per exemple, prenc un píxel i miro el seu veïnat, i aleshores hi apliqui algun descriptor estadístic (mitjana o mediana p.ex.). Aleshores, p.ex., si el valor del píxel supera el de la mitjana, hi donem 1. Si no, 0. Així ens estem adaptant una mica a la intensitat de la zona. Des del punt de visualització no tenen perquè sortir coses gaire maques, perquè en això estem buscant altres coses. Potser en un context mèdic, veiem que a la foto de l'ull del pdf ens queden les benes de l'ull marcades molt bé, el que facilita el seu comptatge. El que més s'utilitza és la mediana, de manera que un punt que tingui molt pes i que no sigui d'importància, la mediana se'l carrega però la mitjana no.

Per programar-ho no cal anar píxel a píxel, podem utilitzar la funció `scipy.signal.medfilt`, que crea una array en què a cada píxel hi associa la mediana del seu voltant. Aquesta array s'anomena imatge de llindars, que és com la imatge original, però una mica desenfocada. Un cop feta, la utilitzem com a llindar per fer la binarització en sí. Programat, queda:

```

1 #Per establir un llindar variable compararem la imatge amb la mediana a cada píxel.
2
3 #Creem un array que contingui a cada píxel el valor mediana dels pixels del seu entorn
4 immedfilt = sp.signal.medfilt(imL,5) #ho fem a un entorn 5x5 (+ lent, + precís)
5
6 #Utilitzem els valors en la array anterior com a threshold per cada píxel
7 imbin1 = np.float64(imL>immedfilt)

```

El problema d'aquest mètode és que queda molt afectada pel soroll. Si utilitzem un veïnat més gran (de 3x3 a 5x5 p.ex.), a costa d'utilitzar un temps de computació més llarg, es suavitza la imatge.

Això encara es pot refinar una mica, per exemple per què no tingui tant soroll. Si a l'hora de fer la mediana escollim el valor central de la sèrie ordenada dels valors del voltant, podem fer el programa de manera que, enllot d'escollir el valor central, escolleixi un altre índex que li diguem. Això equivaldria a augmentar/disminuir el llindar, d'acord amb la variabilitat de l'entorn. Per programar-ho, utilitzem la funció `scipy.signal.order_filter`, que crea una imatge en què a cada píxel es pren els valors del voltant segons la matriu de l'argument `domain`, i se'n pren un valor concret de la llista ordenada de les intensitats dels píxels. Per exemple:

```

1 """Enllot de prendre la mediana, prendrem com a píxel que faci de llindar un que
2 ocupa una posició arbitrària a la sèrie ordenada de pixels de l'entorn.
3 En aquest cas, prendrem un entorn 5x5, i dels 25 valors ordenats, prenem el 10è (el 1r és 0)"""
4 imorderfilt = sp.signal.order_filter(imL,np.ones([5,5]),9)
5
6 imbin2 = np.float64(imL>imorderfilt) #Apliquem binarització

```

Compte que si canviem la mida del veïnat, el nombre que haurem d'escollir és diferent (en una 5x5 l'índex 5 ja no queda a la meitat). Una altra avantatge de la funció és que et permet modificar la geometria del veïnatge. Si fem una matriu `numpy.ones` tindrem un quadrat, però si fem una matriu del tipus

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

aleshores només prendrem els punts immediatament al costat.

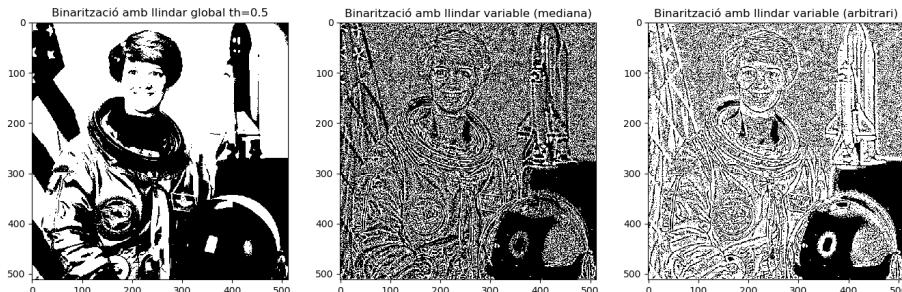


Figure 4: Llindars de binarització aplicats a la mateixa imatge. Esquerra global, centre amb mediana, dreta arbitrari.

3.3 Algorisme de difusió d'error (*dithering*)

Amb aquest algorisme s'aconsegueixen imatges molt realistes. Per exemple, és el que utilitzen les impressores. Podrem reaprofitar l'algorisme per crear imatges en color. però ja ho veurem.

Quan binaritzes segons un llindar fixe, podem cometre errors diferents en funció de la diferència del píxel respecte el llindar. Per exemple, si el llindar fixe és 128, un píxel de 30 que portem a 0 té un error de 98, mentre que un de 100 té un error de 28. La idea darrera dels algorismes de difusió d'error és reinjectar aquest error de la binarització cap endavant, de manera que m'influenciï la binarització dels píxels següents. El llindar es mantindrà fixe, però anirem afegint correccions als valors dels píxels, i veurem que, efectivament, funciona. Com per art de màgia, la imatge s'adapta a la binarització.

Aquesta algorisme actua seqüencialment, per aquesta raó cal implementar-lo píxel a píxel, amb un parell de bucles. Comencem per el píxel de dalt a l'esquerra, i anem baixant. Hi ha dos maneres senzilles de recórrer, una seria sempre d'esquerra a dreta i anar baixant; i una altra fer una fila

d'esquerra a dreta, la següent de dreta a esquerra, i anar alternant. Es veu que la segona va millor, però tampoc es nota molt la diferència. Igualment, nosaltres recorrem normal: de dalt a baix, esquerra a dreta.

Definim l'error de la binarització del píxel com:

$$e = \begin{cases} p_{ij} - \max(\text{image}) & \text{si } p_{ij} > th \\ p_{ij} & \text{si } p_{ij} < th \end{cases} \quad (1)$$

On th és el llindar fixe (0.5 o 128 p.ex.), i $\max(\text{image})$ 1 o 255, depenent si treballem amb coma flotant o enter. Pot ser que apareguin píxels amb valors negatius, però no ens n'hem de preocupar, perquè a l'hora d'aplicar el llindar, els valors negatius s'aniran a 0 igualment. El mateix per valors majors que 1, que aniran a 1 en el llindar.

La fórmula amb què propaguem l'error cap endavant, però, no és única. Per primer banda, l'algorisme Floyd-Steinbeig ho fa segons la matriu:

$$FS = \frac{1}{16} \begin{pmatrix} 0 & 0 & 0 \\ 0 & p & 7 \\ 3 & 5 & 1 \end{pmatrix} \cdot e \quad (2)$$

Aquests pesos no deixen de ser una mica arbitraris. Això vol dir que, a partir de l'error en el píxel ij , la propagació es fa en el seu entorn segons:

$$\begin{pmatrix} p_{[i-1,j-1]} & p_{[i-1,j]} & p_{[i-1,j+1]} \\ p_{[i,j-1]} & p_{[i,j]} & p_{[i,j+1]} \\ p_{[i+1,j-1]} & p_{[i+1,j]} & p_{[i+1,j+1]} \end{pmatrix} \rightarrow \begin{pmatrix} p_{[i-1,j-1]} & p_{[i-1,j]} & p_{[i-1,j+1]} \\ p_{[i,j-1]} & p_{[i,j]} & p_{[i,j+1]} + \frac{7e}{16} \\ p_{[i+1,j-1]} + \frac{3e}{16} & p_{[i+1,j]} + \frac{5e}{16} & p_{[i+1,j+1]} + \frac{e}{16} \end{pmatrix}$$

Notem que la notació de subíndexs no és ben bé la matricial, si no que l'ajustem a com es crida a Python els elements d'una array 2D. Fixem-nos que la propagació no s'aplica als píxels que ja hem recorregut abans, i que la suma de tots els pesos és 1, de manera que no estem modificant l'energia total dels píxels.

Per programar aquests algorisme, ho fem com:

```

1 imFS = np.copy(imL) #treballen en escala de grisos
2
3 #Hem d'aplicar l'algorisme píxel a píxel
4 for ii in np.arange(1,nfil-1): #no recorrem les vores
5     for jj in np.arange(1,ncol-1):
6         #Definim l'error segons si el píxel és < o > de th:
7         if (imFS[ii,jj]>=th):
8             error = imFS[ii,jj]-1.
9         if (imFS[ii,jj]<th):
10             error = imFS[ii,jj]
11         #Definit l'error, el propaguem als següents pixels segons la matriu FS
12         imFS[ii,jj+1] += 7.*error/16.
13         imFS[ii+1,jj-1] += 3.*error/16.
14         imFS[ii+1,jj] += 5.*error/16.
15         imFS[ii+1,jj+1] += 1.*error/16.
16
17 imbin3 = np.float64(imFS>th)

```

Aquests algorismes treballen de manera que al final s'hi apliqui un llindar fixe. Si utilitzem un llindar variable com el de la mediana, el resultat no serà bo.

Un altre manera de propagar l'error és fer-ho amb els pesos donats per l'algorisme Jarvis-Judice-Ninke:

$$JJN = \frac{1}{48} \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & p & 7 & 5 \\ 3 & 5 & 7 & 5 & 3 \\ 1 & 3 & 5 & 3 & 1 \end{pmatrix} \cdot e \quad (3)$$

La programació és exactament la mateixa, amb una mica més de línies de codi:

```

1 imJJN = np.copy(imL)
2
3 for ii in np.arange(2,nfil-2): #tampoc recorrem les vores
4     for jj in np.arange(2,ncol-2):
5         if (imJJN[ii,jj]>=th):
6             error = imJJN[ii,jj]-1.

```

```

7     if (imJJN[ii,jj]<th):
8         error = imJJN[ii,jj]
9 #El mateix que abans, però ara la propagació s'estén més lluny
10    error = error/48
11    imJJN[ii,jj+1]   += 7.*error
12    imJJN[ii,jj+2]   += 5.*error
13    imJJN[ii+1,jj-2] += 3.*error
14    imJJN[ii+1,jj-1] += 5.*error
15    imJJN[ii+1,jj-0] += 7.*error
16    imJJN[ii+1,jj-0] += 5.*error
17    imJJN[ii+1,jj+1] += 3.*error
18    imJJN[ii+2,jj-2] += 1.*error
19    imJJN[ii+2,jj-1] += 3.*error
20    imJJN[ii+2,jj-0] += 5.*error
21    imJJN[ii+2,jj-0] += 3.*error
22    imJJN[ii+2,jj+1] += 1.*error
23
24 imbin4 = np.float64(imJJN>th)

```

Els dos mètodes no ofereixen resultats apreciablement molt diferents:

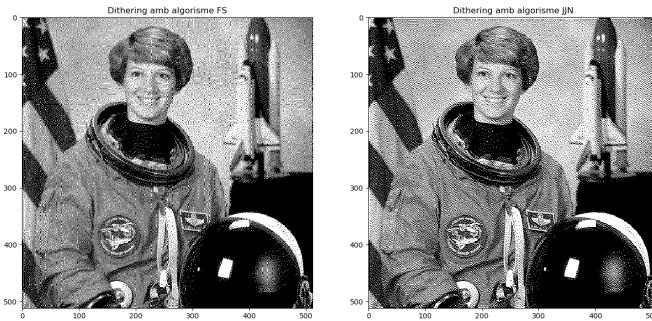


Figure 5: Aplicació dels dos algorismes de *dithering* a la mateixa imatge.

3.4 Degradació a escales de 6 colors

Fa un cert temps, les pantalles i projectors no podien donar una escala infinita de colors. Concretament, només eren capaços de donar 256 valors de colors. Com les imatges en general en tenen més, per poder mostrar-les calia preprocessar la imatge. Per això, cada canal quedava discretitzat a només 6 colors. Per cadascun dels 3 canals, el total de combinacions és $6^3 = 216 < 256$, i per tant tenien menys colors dels màxims ques podien mostrar. Per exemple, si treballem entre 0 i 1, si només podem prendre 6 valors discrets, aquests seran 0, 0.2, 0.4, 0.6, 0.8, 1.0. Això són imatges GIFs. Si tu redueixes la imatge a 6 colors per canal, les zones uniformes queden força bé, però a les zones amb degradats la gamma de color que tenim és insuficient, i no queda gaire bé.

Probablement, la manera més senzilla d'obtenir una imatge en 216 colors és a partir de la següent fórmula:

```

1 im6s = np.copy(im) #treballem amb la imatge en 3 canals RGB
2
3 """Volem passar valors continus [0,1] a valors discrets {0,0.2,0.4,0.6,0.8,1.}
4 Per fer-ho, multipliquem per 5, arrodonim i tornem a dividir entre 5. P. ex:
5 0.13 --> 0.13*5 = 0.65 --> rint(0.65) = 1 --> 1/5 = 0.2
6 0.54 --> 0.54*5 = 2.70 --> rint(2.70) = 3 --> 3/5 = 0.6 """
7 im6s = np.rint(im6s * 5) / 5

```

3.5 Binarització en colors

Per atacar a imatges en colors, enllloc de treballar amb els 3 canals RGB, ho farem en l'espai HSV, on H i S són Hue (el to) i Saturation (quant de barrejat amb blanc i negre està el color). El valor V Value (dóna idea de la lluminositat) es calcula fent el màxim dels tres canals RGB. H i S només contenen informació del color. Per passar d'unes coordenades a les altres s'han de fer unes operacions lineals. A Python, aquestes operacions venen donades en dues funcions, `matplotlib.colors.rgb_to_hsv` i `matplotlib.colors.hsv_to_rgb` per passar una matriu en coma flotant de RGB a HSV, i `matplotlib.colors.hsv_to_rgb`

Com a extra, els càlculs que s'han de fer per passar d'un a l'altre són:

$$V = M = \max\{R, G, B\} \quad m = \min\{R, G, B\} \quad S = \begin{cases} 0 & \text{si } V = 0 \\ \frac{C}{V} & \text{si } V \neq 0 \end{cases} \text{ on } C = M - m \quad (4)$$

Hem definit C com el croma. I finalment:

$$H = 60^\circ \cdot \begin{cases} 0 & \text{si } C = 0 \\ \frac{G-B}{C} \mod 6 & \text{si } M = R \\ \frac{B-R}{C} + 2 & \text{si } M = G \\ \frac{R-G}{C} + 4 & \text{si } M = B \end{cases} \quad (5)$$

Recordem que $\mod 6$ vol dir el residu de la divisió entre 6. Aquest sistema dóna per fet que el té simetria de revolució, de manera que el to està en graus (va de 0 a 360 graus). És un sistema més natural, més adaptat a com construeix els colors una persona, a diferència del RGB, que està construït per les pantalles, que tenen un píxel de cada, R,G i B.

Aleshores, si volem binaritzar una imatge en color, la passem a HSV, i l'apliquem al canal V, els altres dos canals ni els toques, ells només porten informació dels colors. Així, s'obté una imatge binària en color. Programat, queda:

```

1 #Passem la imatge (ha de tenir valors [0,1]) a format HSV
2 imhsv = colors.rgb_to_hsv(im)
3
4 #Farem una binarització només del canal V, el copiem en una matriu per fer-hi feina.
5 imv = np.copy(imhsv[:, :, 2])
6
7 #Apliquem l'algorisme de FS que hem fet abans a la imatge el canal V
8 for ii in np.arange(1, imv.shape[0]-1):
9     for jj in np.arange(1, imv.shape[1]-1):
10        if (imv[ii, jj] >= th):
11            error = imv[ii, jj] - 1.
12        if (imv[ii, jj] < th):
13            error = imv[ii, jj]
14        imv[ii, jj+1] += 7.*error/16.
15        imv[ii+1, jj-1] += 3.*error/16.
16        imv[ii+1, jj] += 5.*error/16.
17        imv[ii+1, jj+1] += 1.*error/16.
18
19 imv = np.float64(imv > th)
20
21 #Substituim a la imatge HSV d'abans el canal V antic pel nou, binaritzat
22 imhsv[:, :, 2] = np.copy(imv)
23
24 #Recuperem el format RGB per poder representar-la
25 imrgb = np.float64(colors.hsv_to_rgb(imhsv))

```

Un cop tenim la imatge binària en color, podem degradar-la a 6 colors per canal per tenir la informació més comprimida:

```
1 imbin6c = np.float64(np.rint(imrgb*5)/5)
```

Com ens diu el guió, pot ser interessar calcular el SSIM (paràmetre de similaritat estructurada) per calcular quantitativament quina de les imatges que hem obtingut és millor des d'una perspectiva bastant antropocèntrica (és a dir, com ho veuria una persona). Aquest paràmetre dóna un número entre 0 i 1, essent 1 la màxima similaritat a la imatge original. Podem calcular aquest índex a partir de la funció `skimage.measure.compare_ssim`:

```

1 #Recuperem el format RGB per poder representar-la
2 imrgb = np.float64(colors.hsv_to_rgb(imhsv))
3 imbin6c = np.float64(np.rint(imrgb*5)/5)

```

Els valors que obtenim de $\approx 0,2$ per les imatges binaritzades són baixos, però ja és més o menys l'esperable, i no és tan dolent com sembla, perquè la progressió del SSIM no és lineal.

Veiem que la cara, per exemple, té un color molt de còmic, lleig, quan ho degradem a 6 colors directament. Encara que els resultats de l'algorisme de binarització no ens semblin gaire bons, a la pràctica les densitats de píxels són molt grans (poca resolució a l'hora de mostrar), com per exemple a les impressores. Els resultats són més correctes si treballem en poca resolució (impressores).

Apunt: A vegades no se'n mostren prou bé algunes imatges amb variacions ràpides entre píxel i píxel. Per exemple, les binaritzacions varien entre 0 i 1 de manera molt brusca, el qual provoca que

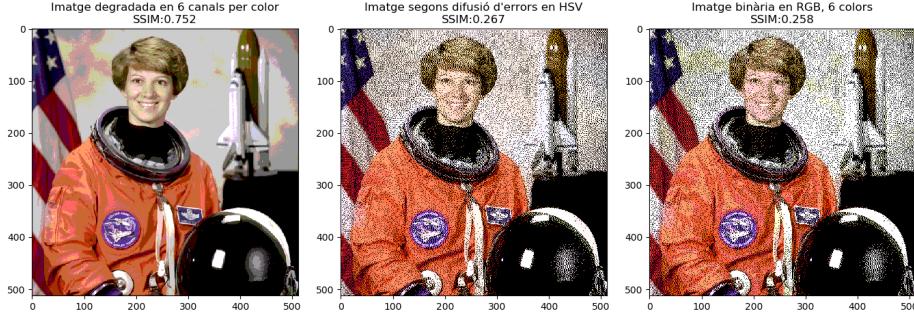


Figure 6: Esquerra: imatge degradada a 216 colors. Centre: imatge amb el canal V de HSV binaritzat. Esquerra: imatge binaritzada i reduïda a 216 colors.

la pantalla no pugui ser 100% fiel a la imatge i en haver d'ometre alguns píxels la imatge es vegi malament (això s'anomena submostreig). Així doncs, que les imatges no es vegin bé pot no ser culpa nostra, sinó del submostreig que escull la pantalla per representar-les. Així que millor treballar amb imatge que no siguin super bona resolució, de manera que les nostres pantalles les puguin processar guay. O, si no, treballar a pantalla completa i fer zoom.

4 Més sobre colors i transformacions de canals (P4)

4.1 Coordenades CIE

Sovint un associa el color amb la longitud d'ona. Però, no és tan immediat, perquè aquesta sola no permet donar totes les tonalitats possibles que percebem. Per intentar aclarir això, farem un exemple on intentarem passar de l'espectre a una imatge en RGB. Per fer-ho, emprarem enmig el sistema CIE, que té una relació directa amb l'espectre de les longituds d'ones, de manera que és més antropocèntric que no pas el RGB (com el HSV). Com a curiositat, el CIE ens permet veure com es modifica la longitud d'ona a què correspon un mateix color dependent del tipus d'il·luminació que hi apliquem, de manera que, efectivament, la relació longitud d'ona-color no és simple ni directa.

Per calcular els canals del sistema CIE, emprem les següents integrals:

$$x = \int E(\lambda)x(\lambda)d\lambda \quad y = \int E(\lambda)y(\lambda)d\lambda \quad z = \int E(\lambda)z(\lambda)d\lambda \quad (6)$$

Les funcions $x(\lambda), y(\lambda), z(\lambda)$ que apareixen són les anomenades *color matching functions*. Són funcions conegudes i tabulades. Les coordenades CIE són aquestes mateixes xyz que hem calculat integrant, normalitzades tal que $X + Y + Z = 1$:

$$X = \frac{x}{x+y+z} \quad Y = \frac{y}{x+y+z} \quad Z = \frac{z}{x+y+z} \quad (7)$$

Per calcular aquestes coordenades en el cas de l'emissió d'una làmpada, utilitzem un fitxer que podem trobar al Campus Virtual, on trobem 5 columnes:

1. Longitud d'ona en nanòmetres.
2. Valor de la funció $x(\lambda)$ per la longitud d'ona corresponent.
3. Valor de la funció $y(\lambda)$ per la longitud d'ona corresponent.
4. Valor de la funció $z(\lambda)$ per la longitud d'ona corresponent.
5. Quantitat d'energia que emet una làmpada en aquella longitud d'ona.

Podem passar aquest arxiu de dades a una array podem utilitzar la funció `numpy.loadtxt`, que és molt convenient. Hem d'indicar-li a la funció que el delimitador de l'arxiu (el símbol que separa columnes) amb l'argument `delimiter=";"`. En aquest cas: `delimiter=";"`.

Un cop tenim cada columna en una array, podem integrar utilitzant la funció `scipy.integrate.simps`. Aquesta funció utilitza com a argument els valors $y(x)$ (les imatges), que és el que introduïm en una array com a argument. Un cop construïdes les arrays que conformen l'integrand, hi utilitzem la funció, especificant quina és l'interval de λ que separa les dades en l'argument `dx`.

Programat, quedaria:

```

1 #Agafem les dades i la guardem a una matriu 2D de 4 columnes
2 data = np.loadtxt("Colorfunctions.csv", delimiter=";")
3
4
5 #Construïm els integrands de cada integral multiplicant x(lambda)*E(lambda)
6 intx = data[:,1]*data[:,4]
7 inty = data[:,2]*data[:,4]
8 intz = data[:,3]*data[:,4]
9
10 x = sp.integrate.simps(intx,dx=5) #Integrem especificant el dlambda
11 y = sp.integrate.simps(inty,dx=5)

```

Aleshores, un cop ja tenim la coordenada CIE d'un píxel de llum de la làmpada, cal passar-ho a RGB. Per fer-ho, és senzillament aplicar una matriu, tal que:

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 3.240479 & -1.537150 & -0.498535 \\ -0.969256 & 1.875992 & 0.041556 \\ 0.0556486 & -0.204043 & 1.057311 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \quad (8)$$

Un cop hem passat les coordenades CIE a RGB, tindrem un píxel de la làmpada. Per poder-ho visualitzar, necessitem més que un píxel, o no veurem res. Per fer-ho, creem una array 3D de 3 canals, de dimensions arbitràries, i hi apliquem a cada píxel els valors que hem trobat. Programat:

```

1 #Construïm una matriu [nfil,ncol] on repetim aquest pixel
2 nfil = 500
3 ncol = 500
4
5 #Farem una matriu de uns i multiplicarem cada nombre pel valor del pixel que toca
6 imlamp = np.ones([nfil,ncol,3],dtype=np.float64)
7 imlamp[:, :, 0] *= R
8 imlamp[:, :, 1] *= G

```

Surt un bonic color blau.

4.2 Equalització d'histograma

És fàcil d'implementar, però no tan fàcil d'entendre què estem fent. Consisteix en generar un automatisme per aprofitar al màxim el rang dinàmic ², de manera que és útil per imatges concentrades en un cert rang de valors (de poc contrast). Quan hi apliquem a una imatge com aquestes, obtenim una imatge amb més contrast, de manera que les zones que semblen més uniformes a l'original, queden més degradades. Per exemple, una imatge de poc contrast pot tenir els valors repartits entre 50 i 100, després d'equalitzar-ne l'histograma, ho estan entre 0 i 255. Un cas concret on això té molta utilitat són les imatges més fosques, on podem obtenir zones més clares.

Per fer l'equalització d'histograma, serà molt convenient treballar amb nombres enteros. Primer hem de crear l'histograma de la imatge, consistent en un comptatge del nombre de cops que surt un determinat nivell de gris, o la seva freqüència, si ho normalitzem. L'histograma ens dóna informació de quins valors són més abundants en la imatge (foscós, clars, grisos mitjans...). D'esquerra a dreta els valors van de fosc a clar.

Després, a partir de l'histograma, calclem l'histograma acumulat, que compta tots els punts anteriors a cada valor. Aquest histograma tindrà a la dreta de tot sempre un valor igual al nombre de píxels³. Si dividim entre el nombre de píxels, obtindrem que el màxim queda a 1. Com veurem, ens interessarà que aquest màxim correspongui a 255, serà molt còmode si treballem en enteros.

Si tots els nivells de grisos tinguessin la mateixa freqüència, l'histograma seria pla, i l'acumulat seria una línia recta. Equalitzar l'histograma és modificar la imatge de manera que l'histograma acumulat sigui el més lineal possible, el qual implicaria que l'histograma normal fos constant. Malgrat l'equalització sí aconsegueix un histograma lineal, l'histograma no serà constant, sinó que tindrà un

²L'interval de valors que poden prendre els nostres píxels (0-255,0-1).

³Calculem el nombre de píxels d'una imatge MxN fent aquest mateix producte $M \cdot N$, és a dir, $nfil \cdot ncol$.

envolvent amb la mateixa forma que l'original, molt més pla que aquest. Malgrat es mantenen alguns nivells de grisos que segueixen sense aparèixer, ara el rang dinàmic s'estén en un interval major, haurem estirat l'histograma.

Aleshores, a la pràctica, com ho programem? L'histograma ja t'ho fa una funció `scipy.ndimage.measurements.histogram`⁴ directament, no cal fer bucles ni comptatges, així que per aquí cap problema. A aquesta funció se li ha de dir quants intervals ha de prendre. Ens interessa que sigui un interval per cada valor del rang dinàmic, per tant entre 0 i 255 (per això és convenient treballar amb integers, en coma flotant tindríem problemes per aquí segons com).

Per fer l'acumulat, apliquem a sobre la variable de l'histograma el mètode `.cumsum()`. Un cop tenim l'histograma acumulat, després és recomanable normalitzar-ho, a 255, si treballem en enter. Si està normalitzat així, podem utilitzar una fórmula ben senzilla per equalitzar l'histograma:

```
1 imequal = hist_acum[im]
```

Quan fem `hist_acum[im]` estem demanant l'índex de `hist_acum` amb una array 2D, és lícit encara que a primeres no ho sembli. El que interpreta NumPy és agafar el nombre amb l'índex corresponent al valor de cadascun dels píxels. Per exemple, si a l'índex [12,34] de `im` el píxel té un valor 50, aleshores es sobreescriva l'element corresponent a l'índex 50. Així per cada píxel, de manera que `hist_acum[im]` acaba sent una array 2D corresponent a una imatge.

Què és el que ha passat, però? Doncs suposem que el valor de l'histograma acumulat a l'índex 50 és 120. Doncs el que fa és enviar aquest 120 a l'índex que li tocaria si fos una recta (si està normalitzat a 255, l'envia a 120, perquè el pendent de la recta és 1). Així, enviem cada píxel a una recta, l'histograma acumulat queda una recta.

Per exemple:

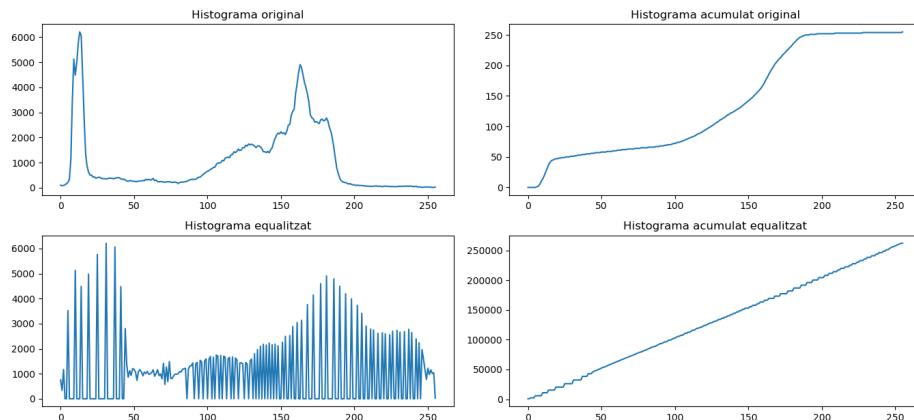


Figure 7: Els histogrames corresponents a les imatges original i equalitzada de la imatge `skimage.data.camera`.

Per programar-ho, podem fer un programa tal que:

```
1 # Treballarem en integer per comoditat
2 im = camera()
3 nfil = im.shape[0]
4 ncol = im.shape[1]
5
6 # Fem els histogrames corresponents, normal i acumulat
7 hist = sp.ndimage.measurements.histogram(im,0,255,256) #Prenem un interval per cada valor possible
8 cumhist = hist.cumsum()
9 cumhist = np.uint8(255*cumhist/np.max(cumhist)) #Normalitzem l'histograma a 255
10
11 #Creem la imatge equalitzada
12 eq = cumhist[im]
13
14 #Fem els histogrames de la imatge equalitzada per comparar-los amb els originals
15 eqhist = sp.ndimage.measurements.histogram(eq,0,255,256)
16 eqcumhist = eqhist.cumsum()
```

El resultat el podem veure a la figura 8.

⁴Segons, l'Artur, vigilem si importem la llibreria amb un *raw import*, aquesta funció està a dues llibreries i podem tenir resultats inesperats. Millor fer un *import explícit*.

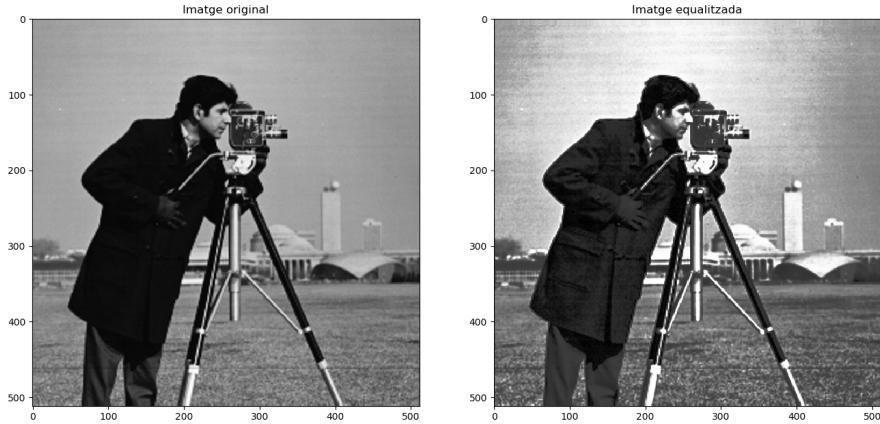


Figure 8: Imatge original (esquerra) vs. imatge equalitzada (dreta).

4.3 Entropia d'una imatge

Al final dels anys 40 es va desenvolupar la teoria de la informació, de la qual Shannon en va ser el pare. L'entropia de Shannon dóna idea de la quantitat de informació rellevant que porta un cert símbol (així com paraules de l'estil *que*, *i o a* són poc entròpiques, altres com *nen* o *espai dual* ho són més). O, com explica l'Artur, més tècnicament, ens dóna la cota superior en la qual nosaltres podem comprimir la informació: l'entropia màxima en una imatge de 8 bits serà 8.

En tot canal on hi ha informació hi ha redundàncies, per exemple una que comença essent molts 000000000 (per exemple, vint 0s). Si aquesta informació la convertim en una array per tenir-la guardat en un arxiu, tindrem que tots aquests 0s ocupen bastant espai i potser hi ha una manera més astuta de guardar-los: fer 20,0. Això és possible en zones poc entròpiques, i és la base de la compressió d'imatges. Quan un calcula l'entropia està donant una cota teòrica sobre quina és la quantitat d'informació que necessites real, eliminant les redundàncies.

A nivell de càlcul, l'entropia de Shannon d'una imatge en `uint8` ve donada per:

$$S = - \sum_{i=0}^N P_i \log_2 P_i \quad (9)$$

On P_i són les probabilitats de cada valor del rang dinàmic (de 0 a 255), i la probabilitat correspon la freqüència de cada píxel, és a dir, el nombre de comptatges (justament els valors que dóna l'histograma), dividit pel nombre total de píxels. El resultat d'aquest càlcul és un nombre promig de l'entropia de la imatge.

En la pràctica ens demana calcular l'entropia global del canal verd de la imatge `skimage.data.astronaut`, mitjançant el següent programa obtenim 7.413:

```

1 from skimage.filters.rank import entropy
2
3 im = astronaut()
4 nfil = im.shape[0] #Necessitarem nfil i ncol per normalitzar l'histograma
5 ncol = im.shape[1]
6 img = np.copy(im[:, :, 1]) #Calcularem l'entropia només del canal G
7
8 # Calculem l'histograma i el normalitzem, és a dir, obtenim les probabilitats en un array
9 hist = sp.ndimage.measurements.histogram(img, 0, 255, 256)/(nfil*ncol)
10 hist = hist[hist != 0] #Eliminem possibles 0 de probabilitat per evitar errors log(0)
11 # Calculem l'entropia global amb la fórmula
12 entrop = -np.sum(hist*np.log2(hist))

```

A més de calcular-la globalment, podem calcular l'entropia localment, i d'alguna manera expressar el fet que la informació de la imatge no està uniformement repartida al llarg de tota la imatge, és a dir, que no a totes les zones necessitarem el mateix nombre de píxels per ser fidels en guardar la imatge. Si ens volem posar, es pot fer manualment amb un càlcul de l'entropia a un entorn arbitrari de cada píxel, és un càlcul local. Si no en tenim ganes, ho podem fer directament amb funció ja implementada, `skimage.filters.rank.entropy`. Aquesta funció retorna una imatge amb diferents valors segons l'entropia de cada zona. Els resultats que trobem si ho fem també en manual poden ser una mica diferents dels resultats que obtinguem amb aquesta funció.

Podem calcular l'entropia local de la imatge que ens demana a la pràctica mitjançant: En la pràctica ens demana aplicar-ho al canal verd de la imatge `skimage.data.astronaut`, mitjançant el següent programa obtenim 7.413:

```

1 from skimage.filters.rank import entropy
2
3 im = astronaut()
4 nfil = im.shape[0] #Necessitarem nfil i ncol per normalitzar l'histograma
5 ncol = im.shape[1]
6 imG = np.copy(im[:, :, 1]) #Calcularem l'entropia només del canal G
7
8 # Calculem l'histograma i el normalitzem, és a dir, obtenim les probabilitats en un array
9 hist = sp.ndimage.measurements.histogram(imG, 0, 255, 256)/(nfil*ncol)
10 hist = hist[hist != 0] #Eliminem possibles 0 de probabilitat per evitar errors log(0)
11 # Calculem l'entropia global amb la fórmula
12 entrop = -np.sum(hist*np.log2(hist))
13
14 # Calculem una imatge d'entropia local a un entorn de 11x11 mitjançant la funció entropy
15 imS = entropy(imG, np.ones([11,11]), dtype=np.uint8)
16 imS /= np.max(imS) #Normalitzem la imatge entre 0 i 1
17
18 plt.figure()
19 implot = plt.imshow(imS, cmap="jet")
20 plt.colorbar()

```

La imatge que obtenim és una cosa així:

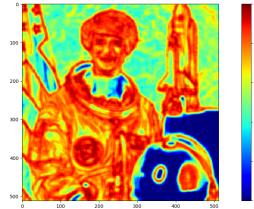


Figure 9: Entropia global del canal verd de l'astronauta. Hem utilitzat `cmap="jet"`.

Com a curiositat, de la mateixa manera que hem calculat l'entropia d'una imatge, podem calcular tant l'histograma com l'entropia d'un text (utilitzant com a probabilitat quantes vegades surt cada lletra respecte el total), i, per exemple, amb això es pot saber en quin idioma està escrit.

4.4 Esteganografia

L'esteganografia és una tècnica d'**amagar** informació sota una imatge, de manera que no sigui accessible sense saber com ha estat amagada. Així com la criptografia cerca escriure en codificacions secrets, en l'esteganografia es fa servir la imatge com a receptora de la informació i s'hi amaga qualsevol cosa dins seu. Per poder amagar una imatge dins una altra, sovint cal degradar-la una mica (per exemple, treure algun color) per poder encabir-la dins d'una altra imatge.

Repàs de bits i codificació en binari

Si no ets un alumne de la menció fonamental que no tens ni idea de com funciona la codificació dels nombres per tal que ho entengui l'ordinador, o has aconseguit que l'Artur t'ajudi, pots passar aquesta subsecció. Si no tens aquesta sort, has de saber que els nombres estan codificats en binari, és a dir, seqüències ordenades de 0 i 1. Si descomponem el nombre en sumes de potències de 2, aquests 0 i 1 ens donen si hem de sumar la potència corresponent o no:

$$25 = \dots + 16 + 8 + 1 = \dots 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \rightarrow 25 = 11001$$

Aleshores, cadascun d'aquests nombres 0 o 1 ocupa un bit de memòria. Un nombre `uint8` pot arribar fins a 255 perquè és el nombre màxim que es pot obtenir amb la suma de 8 potències de 2: $255 = 1111111$, a partir d'aquest cal un altre bit per guardar-lo. Un `uint6` tindrà un rang dinàmic entre 0 i 63.

En l'esteganografia, ens interessarà modificar lleugerament els valors de manera que reservem certs bits i en aquests bits poder ficar-hi la imatge secreta. Si volem reservar n bits a la dreta del nombre

(és a dir, que els n bits de la dreta siguin 0s), farem:

$$x = 2^n \int \left(\frac{x}{2^n} \right) \quad (10)$$

És a dir, aconseguim que x sigui múltiple de 2^n , de manera que no necessitarà potències menors de 2^n per ser escrit.

També ens pot interessar reservar els n valors de l'esquerra, aleshores senzillament cal fer:

$$x = \int \left(\frac{x}{2^n} \right) \quad (11)$$

Aquesta operació es pot fer immediatament amb Python mitjançant l'operador `//`, que dóna la part entera de la divisió. Degradar una imatge de `uint8` a `uint6` implica fer aquest càcul.

Si hem de decidir quins bits perdre, és clar que els bits menys significatius són els de la dreta: sumar o restar 2^0 o 2^1 no canviarà tant el valor com si canviem un 2^7 . Aquests seran, doncs, els bits on hem de guardar la imatge secreta, i per això serà útil l'equació (10).

Esteganografia dins una imatge en color

L'esteganografia que nosaltres treballarem serà la d'una imatge de 8 bits en escala de grisos (per tant, 1 canal `uint8`) amagada dins una imatge en color (24 bits en total per píxel, 8 bits a cada canal). Aquestes imatges de 24 bits poden assolir un total de $256^3 \approx 16$ milions de colors possibles, mentre que nosaltres els humans podem veure uns 100000 colors, de manera que tenim un cert marge per poder encabir informació que nosaltres no podem percebre. La imatge pot perdre nombre de bits sense que ho percebem de forma clara, si aquests bits són pocs significatius.

Primer de tot, ens assegurarem que les dues imatges tinguin la mateixa mida, si no ens trobarem píxels que no tenen hoste o píxels desocupats. És molt senzill, utilitzem la funció `skimage.transform.resize`:

```

1 from skimage.transform import resize
2
3 imhost = astronaut() #Aquesta és una imatge en color, de 3 canals, que amagarà (hoste)
4 imhid = chelsea() #Aquesta és la imatge a amagar, la volem en escala de grisos
5 imhid = 0.299*imhid[:, :, 0]+0.587*imhid[:, :, 1]+0.114*imhid[:, :, 2]
6
7 #A l'hora de redimensionar és comprimir la imatge de les dues que sigui més gran
8 if imhost.shape[0:2] == imhid.shape:
9     print("Les imatges tenen la mateixa mida")
10    #Les imatges tenen la mateixa mida, no cal redimensionar
11 else:
12     imhid = resize(imhid, imhost.shape[0:2])
13
14 nfil = imhost.shape[0]
15 ncol = imhost.shape[1]
```

Notem que la imatge que redimensionem sempre és la secreta, perquè si redimensionem la imatge hoste, que és la que mostrem, pues com que es notarà de primeres que hi hem fet algun canvi.

Un cop tinguin la mateixa mida, degradarem la imatge secreta que volem amagar de 8 bits a una de 6 bits (passem de 256 valors possibles a només 64), de manera que amb 6 0s o 1s per píxel ja tenim tota la informació de la imatge. Aquests 6 bits els hem de passar a la imatge hoste, de manera que a cada canal d'aquesta reservem els 2 bits menys significatius dels 8 (és a dir, els dos darrers). En aquest espai buit, col-loquem els 6 bits de la imatge secreta, dos a cada canal. El més probable que el valor del píxel varii, però no de manera significativa com perquè ho puguem veure.

El millor és veure-ho com ho podem programar:

```

1 nfil = imhost.shape[0]
2 ncol = imhost.shape[1]
3
4 #A cadascun dels valors dels píxels, descomponem en bits (en binari), fent:
5 bitshost = np.unpackbits(imhost, axis=-1)
6 #Tindrem una matriu de 0s i 1s, on les agrupacions de 8 en 8, ordenades, són el valor
7 # del píxel en binari. La forma no és la bona, la redimensionem a una forma més natural
8 bitshost = bitshost.reshape([nfil, ncol, 3, 8])
9 #Serà la forma d'una array d'imatge, però on hi havia el valor del píxel,
10 #hi ha una array de 8 valors, els 8 bits d'aquell píxel.
11
12 """Volem degradar la imatge de uint8 a uint6. uint6 va de 0 a 63, per tant,
```

```

13 renormalitzem els valors dels píxels en aquest interval."""
14 imhid = np.uint8(imhid/4)
15 #Fem el mateix que abans:
16 bitshid = np.unpackbits(imhid, axis=-1)
17 bitshid = bitshid.reshape([nfil, ncol, 8])
18
19 #Veurem que, com els valors van fins a 63, només necessitem 6 bits per guardar
20 #els valors, de manera que els indexs 0 i 1 ( $2^7$  i  $2^8$ ) són 0, els podem retallar:
21 bitshid = bitshid[:, :, 2:]
22 #Ara tenim una imatge (nfil, ncol, 6), a cada píxel el valor en una array de 6 bits
23
24 """Ara substituirem a tots els píxels (per tant recorrem tots els dos primers indexs).
25 Per cada pixel els bits són:
26 receptors: els dos darrers bits (signifiquen  $2^0=1$  i  $2^1=1$ ) de cada canal del host
27 introduits, els 6 píxels de la imatge a amagar de dos en dos, dos a cada canal."""
28 bitshost[:, :, 0:6] = bitshid[:, :, 0:2] #Amaguem al canal R
29 bitshost[:, :, 1:6] = bitshid[:, :, 2:4] #Amaguem al canal G
30 bitshost[:, :, 2:6] = bitshid[:, :, 4:6] #Amaguem al canal B
31
32 #Ara reconstruïm els paquets de 8 bits en valors uint8
33 imsteg = np.packbits(bitshost)
34 #Fem que tingui forma de imatge en 3 canals
35 imsteg = imsteg.reshape([nfil, ncol, 3])

```

La imatge resultant és indistingible de la imatge original:

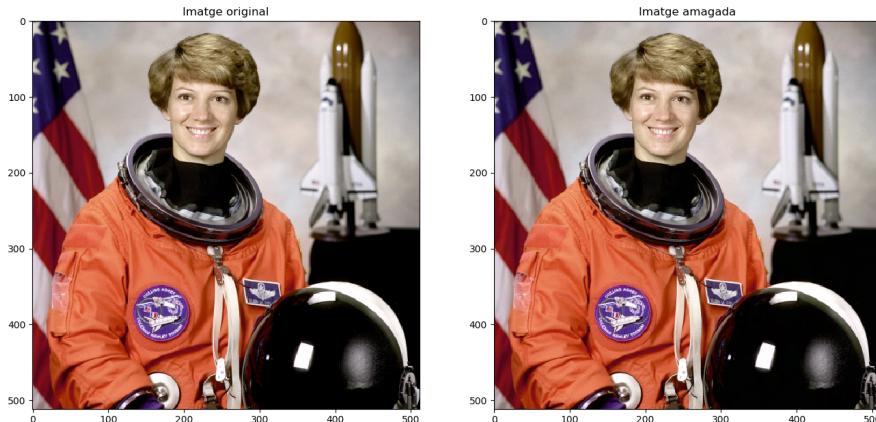


Figure 10: Comparació de la imatge original amb la imatge amb la informació secreta amagada.

Ara bé, la gràcia d'aquest mètode ha de ser poder recuperar la imatge secreta, sempre i quan sapiguem com ha estat guardada la seva informació. En aquest cas, no ens ho hem currat gaire, així que és bastant senzill, jo no hi guardaria cap secret d'estat o exemple de Planta8:

```

1 #Fins aquí la imatge ja està amagada, mirem si la podem recuperar
2 #Només disposem de la imatge amb el secret dins, desfem-la en bits i redimensionem-la
3 bitssteg = np.unpackbits(imsteg, axis=-1)
4 bitssteg = bitssteg.reshape([nfil, ncol, 3, 8])
5
6 #Creem una array on guardarem la imatge secreta recuperada.
7 bitsrec = np.zeros([nfil, ncol, 8], dtype=np.uint8)
8
9 #Sabem a quins píxels està guarda la informació secreta, només hem de desfer
10 #els passos de l'encriptació
11 bitsrec[:, :, 2:4] = bitssteg[:, :, 0:6] #Recuperem del canal R
12 bitsrec[:, :, 4:6] = bitssteg[:, :, 1:6] #Ídem per G
13 bitsrec[:, :, 6: ] = bitssteg[:, :, 2:6] #Ídem per B
14
15 #Reconstruïm els bits en els valors que hi toquen, li donem forma, i passem a uint8
16 imrec = np.packbits(bitsrec)
17 imrec = imrec.reshape([nfil, ncol])
18 imrec = np.uint8(imrec/63*255)

```

I la imatge que recuperem (figura 11) és la correcta.

Podem jugar amb les posicions on col-loquem cada píxel i fer-ho random, de manera que el píxel [0,0] no coincideixi amb el píxel [0,0] de la imatge secreta. La llavor amb què generem aquests nombres aleatoris serà la nostra clau d'encriptació.

El codi ASCII és una codificació numèrica de tots els caràcters en 7bits, de manera que els primers 30 números són no utilitzable. Està fet per l'anglès, de manera que al principi no hi havia accents (en

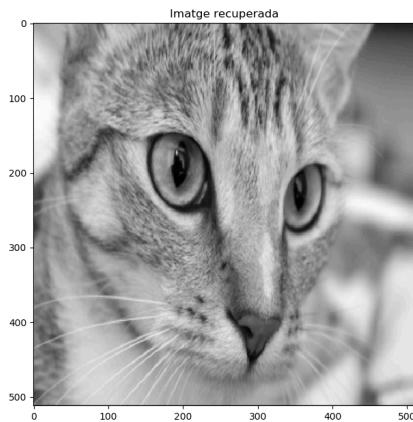


Figure 11: Recuperació de la imatge original que havíem amagat sota la imatge hoste. Està deformada perquè hem hagut de redimensionar-la per poder-la encabir.

7bits hi ha poc espais). Cada caràcter és un byte, anàleg a un píxel. També podem treballar amb UTF-8, que ens cal dos bytes per codificar la lletra. Així doncs, podem també amagar un text a dins una imatge si ho codifiquem bé.