Elaborazione di Segnali Multimediali

Elaborazioni nel dominio spaziale (2)

L.Verdoliva, D.Cozzolino

In questo laboratorio proseguiamo lo studio sulle elaborazioni spaziali, in particolare studieremo il bit-plane slicing ed esamineremo sia le operazioni aritmetiche che quelle geometriche.

1 Bit-plane slicing

Consideriamo un'immagine in cui ogni livello è rappresentato su 8 bit. E' possibile suddividere l'immagine in bit-plane, cioè in piani in cui si rappresentano ognuno dei bit da quello meno significativo (bit-plane 0) a quello più significativo (bit-plane 7). La decomposizione di un'immagine digitale in bit-plane (*bit-plane slicing*) è molto utile per comprendere l'importanza che ogni bit ha nel rappresentare l'immagine e quindi se il numero di bit usato nella quantizzazione è adeguato. Estraiamo i bit-plane dell'immagine `frattale.jpg` usando la funzione `bitget` del file `bitop.py` fornito:

```
from bitop import bitget
B = bitget(x, 7)                        # estrazione bit-plane più significativo
plt.imshow(B, clim=[0,1], cmap='gray') # visualizzazione del bit-plane
```

Scrivete uno script dal nome `bit_plane.py` in cui estraete e visualizzate tutti i bit-plane dell'immagine. Potete memorizzare i bit-plane in una struttura tridimensionale in cui `bitplane[:,:,i]`.

1.1 Esercizi proposti

1. *Ricostruzione mediante bit-plane*. Ponete a zero i bit-plane meno significativi di un'immagine (usate la funzione `bitset` del file `bitop.py`) e visualizzate il risultato al variare del numero di bit-plane che utilizzate nel processo di ricostruzione. Questo esperimento vi permette di stabilire fino a che punto (almeno da un punto di vista percettivo) sia possibile diminuire il numero di livelli usati nel processo di quantizzazione.

2. *Esempio di Watermarking*. Provate adesso a realizzare una forma molto semplice di watermarking, che consiste nell'inserire una firma digitale all'interno di un'immagine. Sostituite il bit-plane meno significativo dell'immagine lena.y con l'immagine binaria marchio.y. Quest'ultima ha dimensioni 350×350 quindi è necessario estrarre una sezione delle stesse dimensioni dell'immagine lena.y. Provate poi a ricostruire l'immagine e visualizzatela, noterete che da un punto di vista visivo l'immagine non ha subito modifiche percettibili.

   Ripetete l'esperimento inserendo il watermark in un diverso bit-plane.

2 Operazioni aritmetiche

Le operazioni aritmetiche (somma/sottrazione, prodotto/divisione) coinvolgono una o più immagini e si effettuano pixel per pixel. Per esempio, fare la sottrazione tra due immagini vi permette di scoprire le differenze che esistono tra due immagini. Provate allora a visualizzare l'immagine frattale.jpg, e quella in cui sono stati posti a zero i 4 bit-plane meno significativi. Noterete come da un punto di vista visivo sono molto simili, fatene allora la differenza e visualizzatela a schermo.

3 Operazioni geometriche

Le operazioni geometriche consentono di ottenere, a partire da un'immagine x una nuova immagine y nella quale non si modificano i valori di luminosità, ma solo la posizione dei pixel.

3.1 Ridimensionamento

Rimpicciolire un'immagine di un fattore intero è estremamente semplice da realizzare in Numpy. Supponiamo per esempio di volerla ridurre di un fattore 2 lungo entrambe le direzioni, allora:

```
y = x[::2,::2]                              # decimazione per 2
plt.imshow(y, clim=[0,255], cmap='gray'); # visualizzazione
```

Questa operazione ci permette di modificare la risoluzione spaziale dell'immagine e consiste di fatto nell'abbassare (in numerico) la frequenza di campionamento del segnale. Se volessimo invece rimpicciolirla di un fattore non intero, realizzando per esempio la trasformazione $y[m,n] = x\left[\frac{3}{2}m, \frac{3}{2}n\right]$ bisogna fare più attenzione perché occorre assegnare correttamente i valori di intensità in uscita, come per esempio $y[1,1] = x\left[\frac{3}{2}, \frac{3}{2}\right]$. Quest'ultimo valore non è definito nell'immagine in ingresso per cui bisogna determinarlo mediante interpolazione. Possiamo utilizzate la funzione `rescale` del modulo `skimage.transform` che ridimensiona l'immagine con interpolazione bilineare:

```
from skimage.transform import rescale
y = rescale(x, 2/3, order=1)
plt.imshow(y, clim=[0,255],cmap='gray');
```

Il parametro `order` è il tipo di interpolazione. Con l'opzione `order=0` si effettua un'interpolazione nearest neighbor, mentre con `order=1` di tipo bilineare. Chiaramente si può anche ingrandire l'immagine se il fattore scelto è maggiore di 1; inoltre, con la funzione `skimage.transform.resize` si possono fissare le dimensioni che deve avere la nuova immagine (se però non si fa attenzione a conservare il rapporto d'aspetto, si crea distorsione nell'immagine).

In Python è possibile effettuare una trasformazione geometrica affine specificando direttamente la matrice di trasformazione $\mathbf{A}$ attraverso la funzione di `skimage.transform.warp`. Supponiamo di voler ingrandire una sezione di 25×50 pixel intorno all'occhio di lena:

```
from skimage.transform import warp

x = np.float32(io.imread('lena.jpg'))
x = x[252:277,240:290];
M = x.shape[0]; N = x.shape[1]

A = np.array([ [0.5,0,0], [0,0.5,0], [0,0,1]], dtype=np.float32)
y1 = warp(x, A, output_shape=(2*M,2*N), order = 0)
y2 = warp(x, A, output_shape=(2*M,2*N), order = 1)
y3 = warp(x, A, output_shape=(2*M,2*N), order = 3)

plt.subplot(3,1,1);
plt.imshow(y1,clim=[0,255],cmap='gray'); plt.title('interpolazione nearest');
plt.subplot(3,1,2);
plt.imshow(y2,clim=[0,255],cmap='gray'); plt.title('interpolazione bilinear');
plt.subplot(3,1,3);
plt.imshow(y3,clim=[0,255],cmap='gray'); plt.title('interpolazione bicubic');
```

Notate l'effetto di blocchettatura causato dall'interpolazione con opzione 'nearest' rispetto a 'bilinear' e 'bicubic'. Alla funzione `warp` abbiamo fornito anche il parametro `output_shape` che indica le dimensioni dell'immagine di uscita. Se il parametro `output_shape` è omesso l'immagine ottenuta `y` avrà lo stesso numero di righe e colonne di `x` ottenendo l'ingrandimento solo di una parte dell'immagine. Fate attenzione al fatto che la matrice affine richiesta dalla funzione `warp` è diversa da quella che abbiamo definito in teoria. In particolare:

$$[m', n', 1] = [m, n, 1]\,\mathbf{T}$$

mentre per la funzione `warp` si ha:

$$[n', m', 1] = [n, m, 1]\,\mathbf{A}^t$$

Ci sono quindi due differenze fondamentali: un'inversione di righe e colonne e una trasposta della matrice stessa. I comandi Python che ci permettono di ottenere $\mathbf{A}$ a partire da $\mathbf{T}$ sono i seguenti:

```
A = T[[1,0,2],:][:,[1,0,2]].T
```

3.2 Traslazioni e Rotazioni

Proviamo a realizzare la traslazione di un'immagine ($m' = m + 50, n' = n + 100$):

```
x = np.float32(io.imread('lena.jpg'))
A = np.array([ [1,0,100], [0,1,50], [0,0,1]], dtype=np.float32)
y = warp(x, A, order = 1)
plt.subplot(1,2,1);
plt.imshow(x,clim=[0,255],cmap='gray'); plt.title('originale');
plt.subplot(1,2,2);
plt.imshow(y,clim=[0,255],cmap='gray'); plt.title('traslata');
```

E' anche possibile modificare il colore per i pixel esterni al dominio dell'immagine. Se per esempio si vuole che abbiano colore bianco:

```
y = warp(x, A, order=1, cval=255)
```

In questo modo si inserisce una gradazione di grigio specificando un valore tra 0 (nero) e 255 (bianco).
Per la rotazione di un'immagine si ha:

```
from skimage.transform import warp

x = np.float32(io.imread('lena.jpg'))

A = np.array([[np.cos(np.pi/4),np.sin(np.pi/4),0],
     [-np.sin(np.pi/4),np.cos(np.pi/4),0],
     [0,0,1]], dtype=np.float32)

y = warp(x, A, order = 1)
plt.subplot(1,2,1);
plt.imshow(x,clim=[0,255],cmap='gray'); plt.title('originale');
plt.subplot(1,2,2);
plt.imshow(y,clim=[0,255],cmap='gray'); plt.title('ruotata');
```

Notate che il centro di rotazione non è il centro dell'immagine, ma la posizione [0,0]. Confrontate questo risultato con quello che otterreste direttamente con la funzione `skimage.transform.rotate`.

3.3 Esercizi proposti

1. *Distorsione*. Scrivete la funzione che realizza la distorsione di un'immagine lungo la direzione verticale e orizzontale e che abbia il prototipo: `deforma(x,c,d)`. Scegliete un'immagine e al variare dei parametri `c` e `d` osservate il tipo di distorsione.

2. *Rotazione Centrale*. Scrivete una funzione con il prototipo `ruota(x, theta)` che utilizza la funzione `wrap` per ruotare di un'angolo `theta` l'immagine `x` rispetto al centro dell'immagine. A tal fine usarte una combinazione di traslazioni e rotazione. Ricordatevi che la combinazione di diverse trasformazioni affini è ancora una trasformazione affine, che può essere ottenuta tramite il prodotto (matriciale) delle matrici che le definiscono.

3. *Combinazione di operazioni geometriche*. Scrivete una funzione dal prototipo `rot_shear(x,theta,c)` per realizzare una rotazione e poi una distorsione verticale (attenzione all'ordine!). Fate le due operazioni rispetto al centro dell'immagine. Create l'immagine di ingresso usando il seguente comando `x = np.float64(skimage.data.checkerboard())` in modo da generare una scacchiera su cui le modifiche risultano essere più facilmente visibili.