

Immagini

17634 | VISIONE ARTIFICIALE



Le immagini digitali

- Matrici di valori (pixel: picture element): ognuno rappresenta il dato (campionato e quantizzato) misurato da un sensore



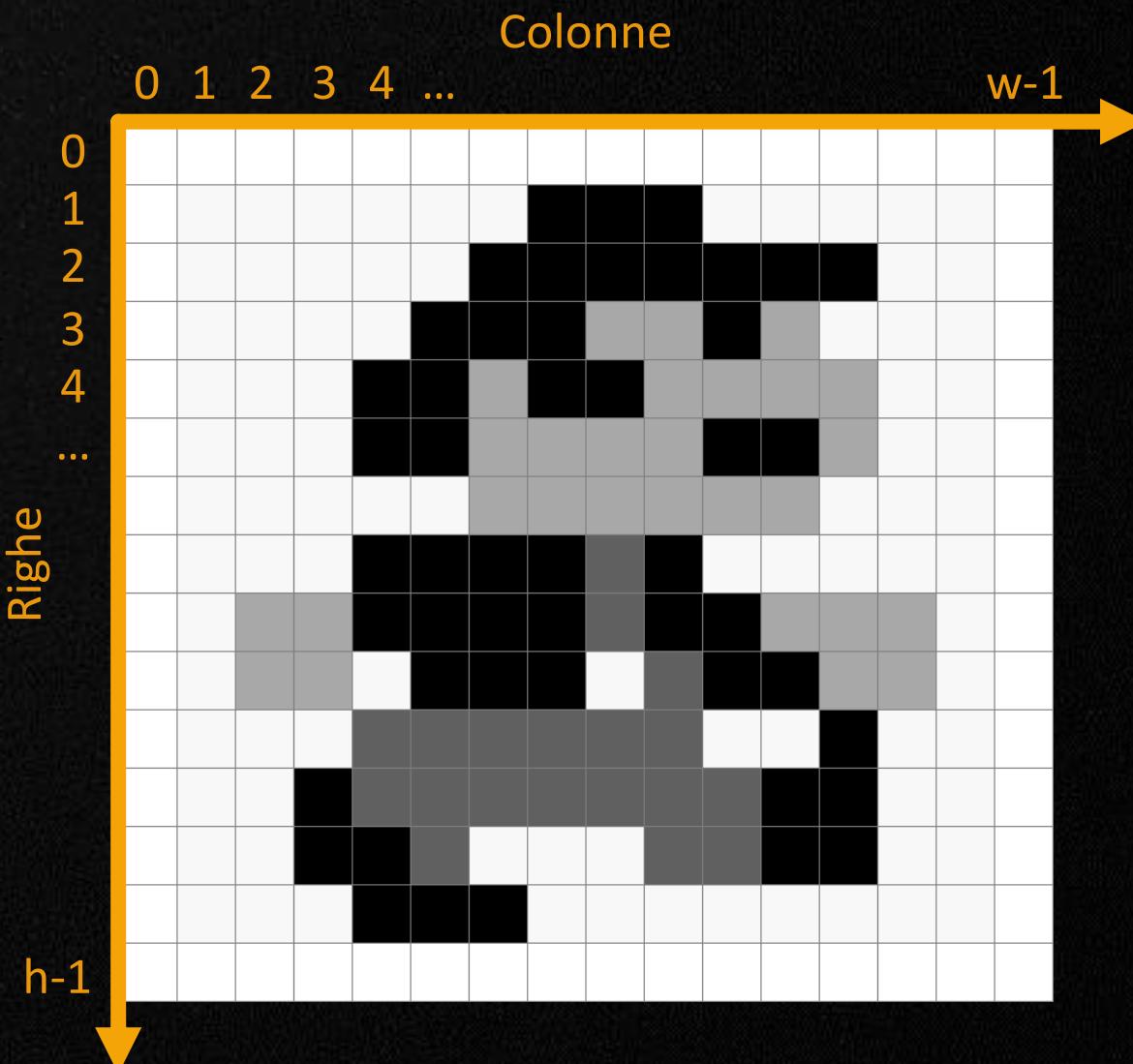
Le immagini digitali

- ▶ Dimensione (WxH) e Risoluzione (DPI)
- ▶ Formato dei pixel
(Bianco/Nero, Grayscale, Colore)
- ▶ Formati di memorizzazione
(JPG, PNG, BMP,...) e compressione
- ▶ Occupazione di memoria
(non compressa): WxHxDepth
(Depth = bit per pixel)

NY.PNG (46110 byte)
380x260
8 bpp
98800 byte (non compressa)



Immagine grayscale: matrice di «punti di luce»



Valore del pixel: quantità di luce

Colonne														
0	1	2	3	4	...	w-1								
0	248	248	248	248	248	248	248	248	248	248	248	248	248	248
1	248	248	248	248	248	248	0	0	0	248	248	248	248	248
2	248	248	248	248	248	248	0	0	0	0	0	0	248	248
3	248	248	248	248	248	0	0	0	168	168	0	168	248	248
4	248	248	248	248	0	0	168	0	0	168	168	168	248	248
...	248	248	248	248	0	0	168	0	0	168	168	168	248	248
Righe	0	1	2	3	4	...	h-1							
0	248	248	248	248	248	248	168	168	168	168	168	168	248	248
1	248	248	248	248	248	248	0	0	0	96	0	248	248	248
2	248	248	168	168	0	0	0	0	96	0	0	168	168	168
3	248	248	168	168	248	0	0	0	248	96	0	0	168	168
4	248	248	248	248	96	96	96	96	96	96	248	248	0	248
...	248	248	248	248	0	96	96	96	96	96	96	0	0	248
h-1	248	248	248	248	0	0	0	248	248	248	248	248	248	248



Valore del pixel: quantità di luce

- ▶ Valori più alti: maggiore luminosità
- ▶ Valori più bassi: minore luminosità
- ▶ Valore 0: nessuna luminosità
- ▶ Intervalli tipici:
 - Byte [0,255]
 - Float [0,1]



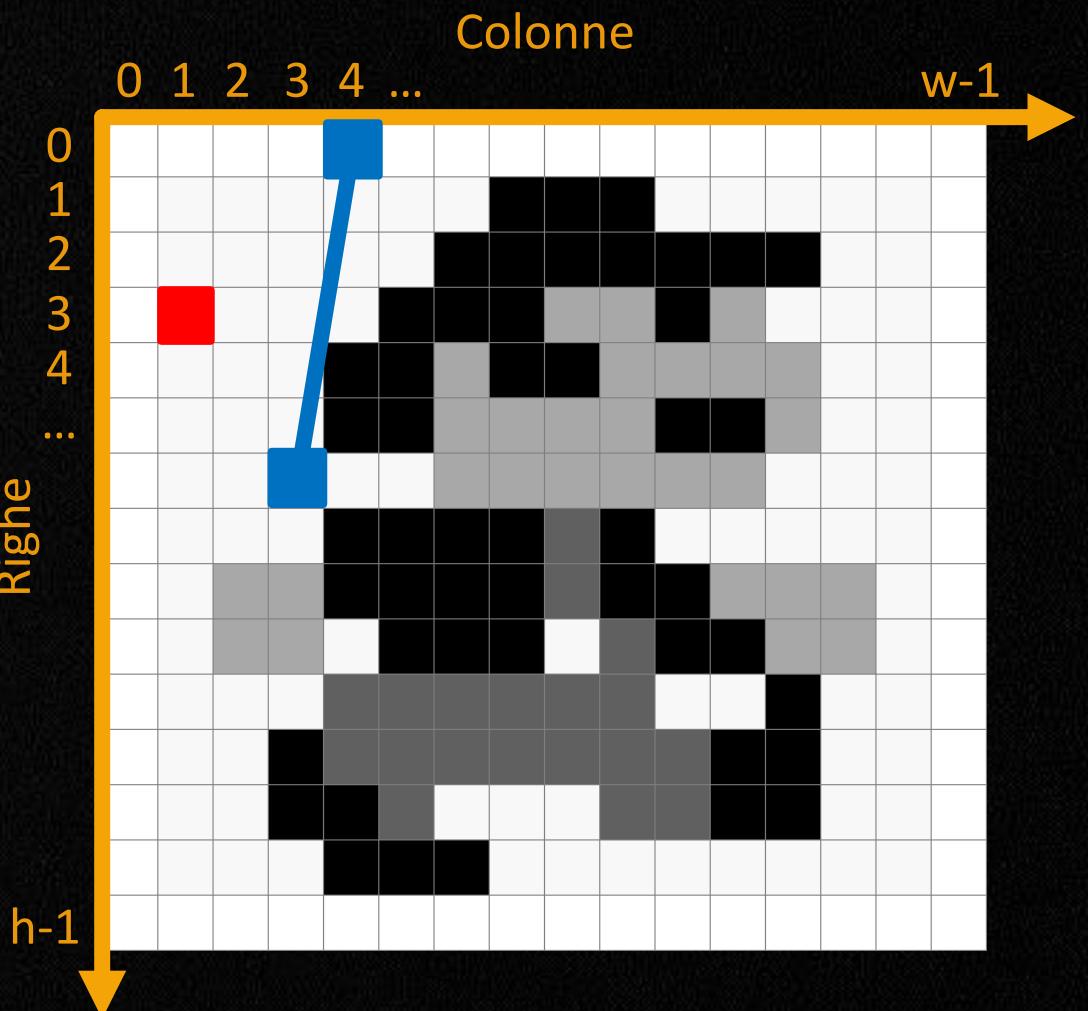
Colonne

	0	1	2	3	4	...	w-1
0	248	248	248	248	248	248	248
1	248	248	248	248	248	248	248
2	248	248	248	248	248	0	0
3	248	248	248	248	0	0	0
4	248	248	248	0	0	168	168
...	248	248	248	0	0	168	168
Righe	248	248	248	248	248	168	168
h-1	248	248	248	248	248	168	168

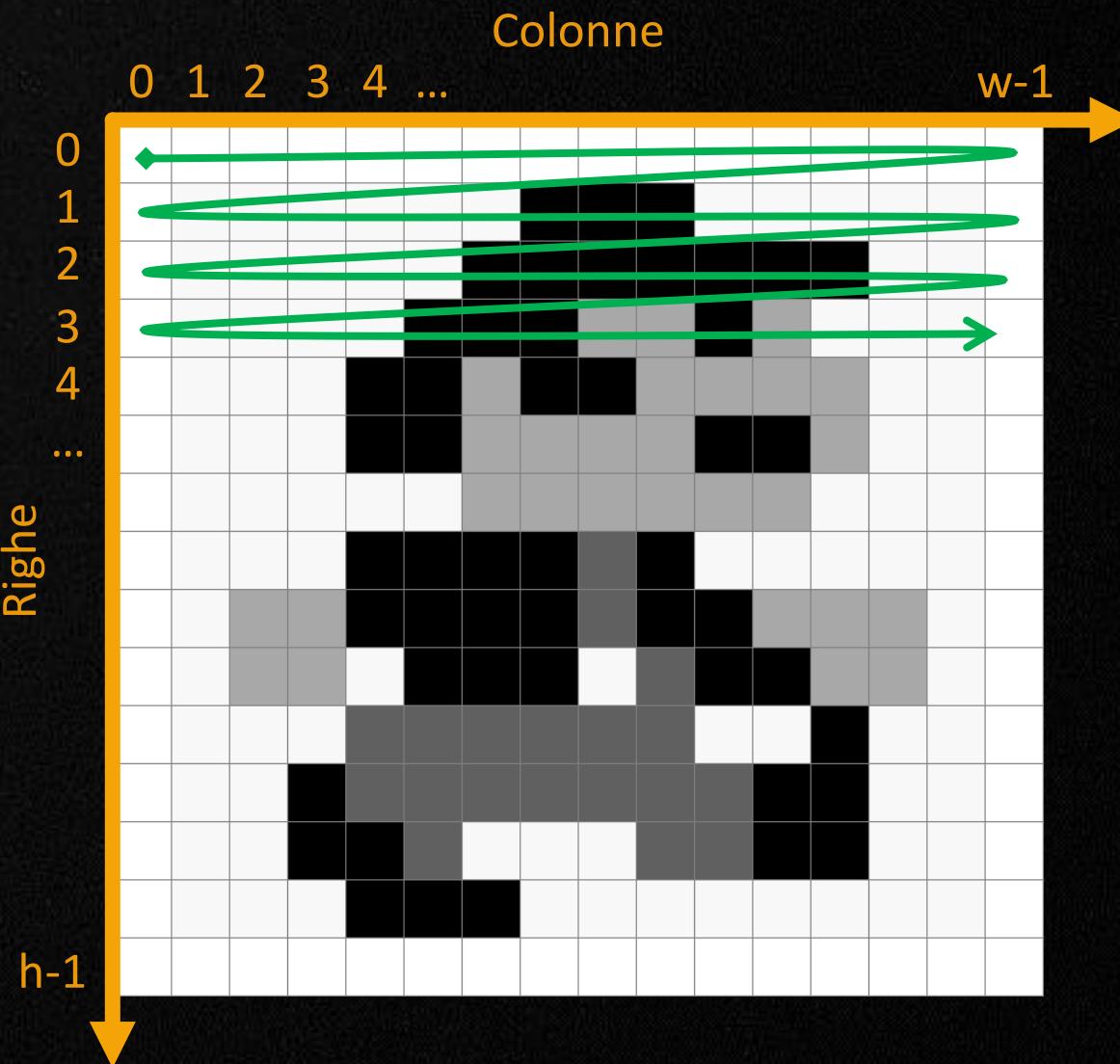
Coordinate dei pixel

- ▶ Coordinate cartesiane: x, y
 - `cv.line(img, (4,0), (3,6), color)`

- ▶ Notazione matriciale: r, c
 - `img[3,1] = 42`



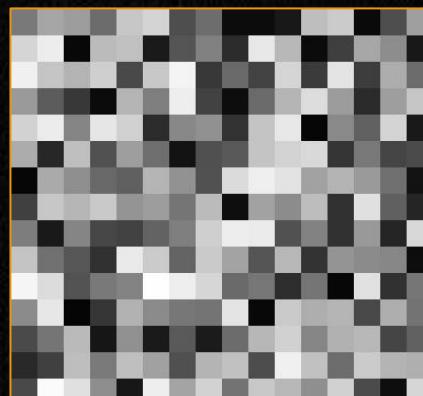
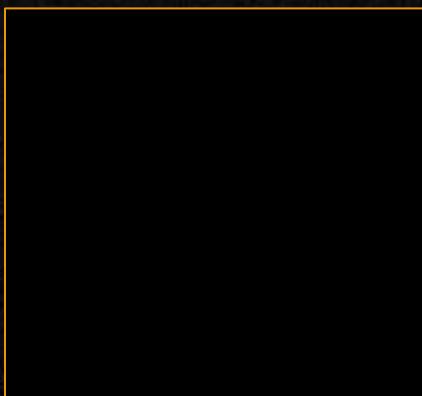
Organizzazione dei pixel in memoria: per righe



Immagini grayscale in Python/OpenCV: array NumPy bidimensionali

9

```
# Creazione di un'immagine grayscale 16x15 con un byte per pixel  
  
img1 = np.zeros((15, 16), dtype=np.uint8) # Tutti i pixel a 0  
img2 = np.full((15, 16), 255, dtype=np.uint8) # Tutti i pixel a 255  
img3 = np.random.randint(0, 256, (15, 16), dtype=np.uint8) # Valori dei pixel casuali  
  
  
# Caricamento di un'immagine grayscale da file (se è a colori viene convertita)  
# N.B. se il file non esiste non genera un errore ma restituisce None  
  
img4 = cv.imread('esempi/mario.png', cv.IMREAD_GRAYSCALE)
```



Immagini grayscale in Python/OpenCV: array NumPy bidimensionali

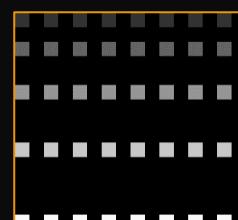
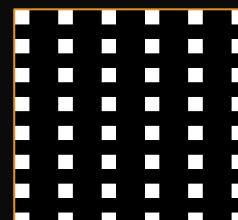
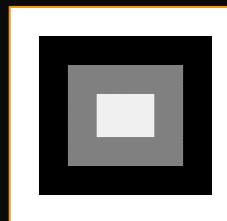
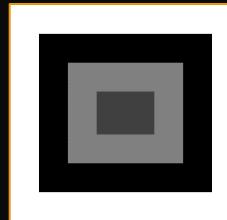
10

```
a = np.full((15, 16), 255, dtype=np.uint8)
# Slicing e broadcasting
a[2:-2, 2:-2] = 0
a[4:-4, 4:-4] = 128
a[6:-6, 6:-6] = 64

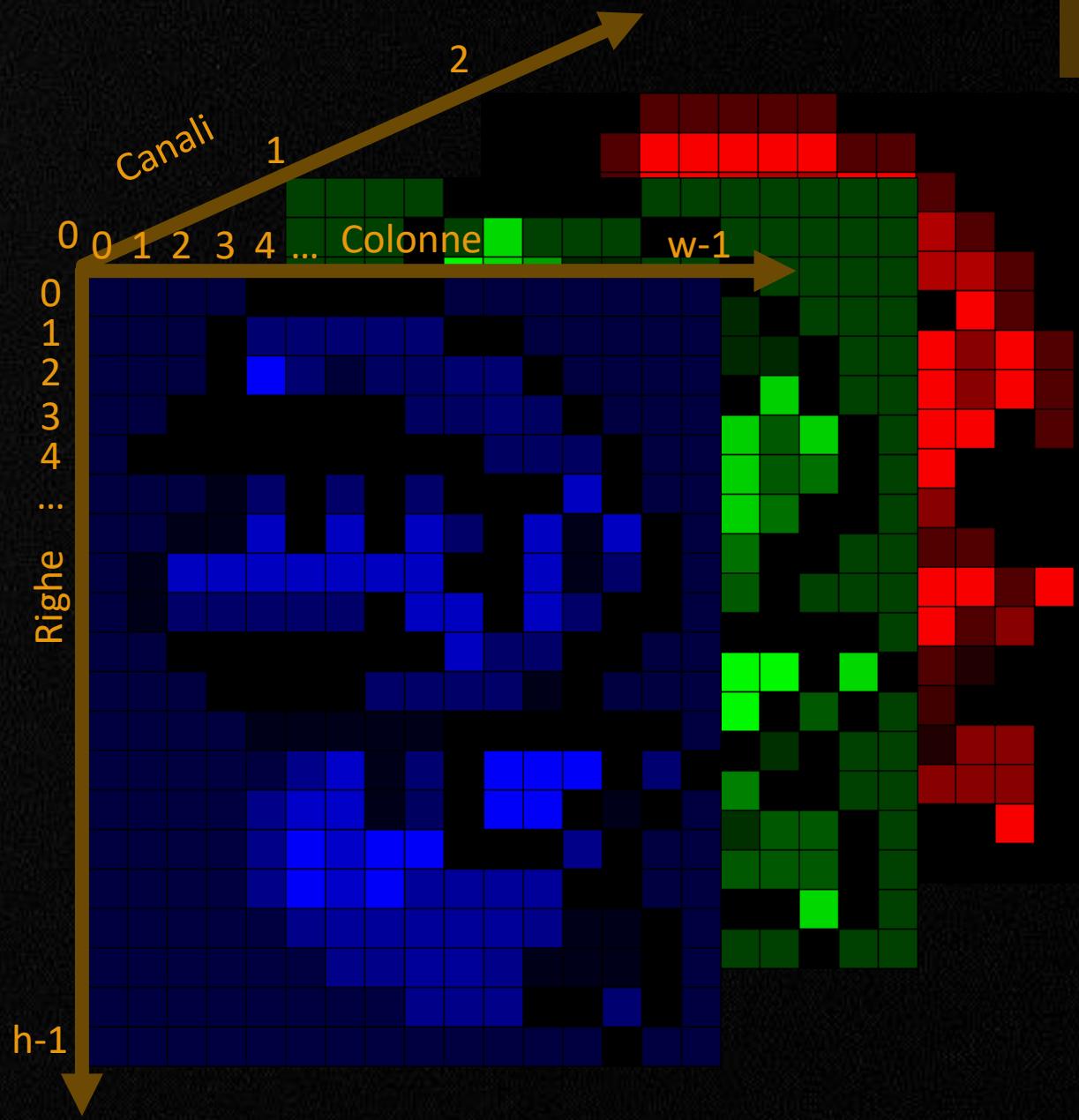
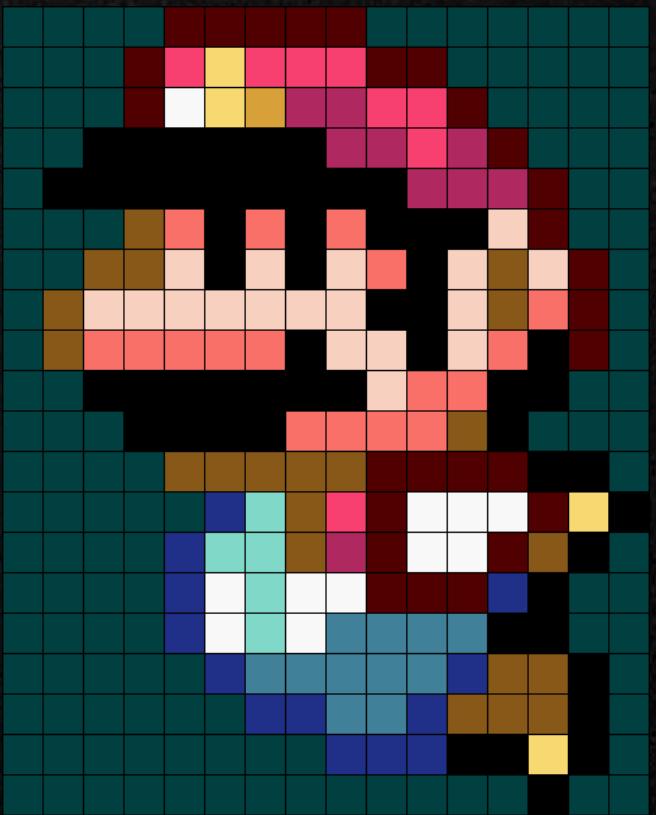
b = a.copy()
# Boolean indexing e broadcasting
mask = b==64 # mask è un array di bool
b[mask] = 240

c = np.zeros_like(a)
# Slicing e broadcasting
c[::-2, ::3] = 255

d = np.zeros((15, 16), dtype=np.uint8)
# Integer array indexing e broadcasting
y = np.array([0, 2, 5, 9, 14])
x = np.arange(0, 16, 2)
d[y[:, np.newaxis], x] = np.arange(50, 255, 50)[:,np.newaxis]
```



Immagini a colori: tensori 3D

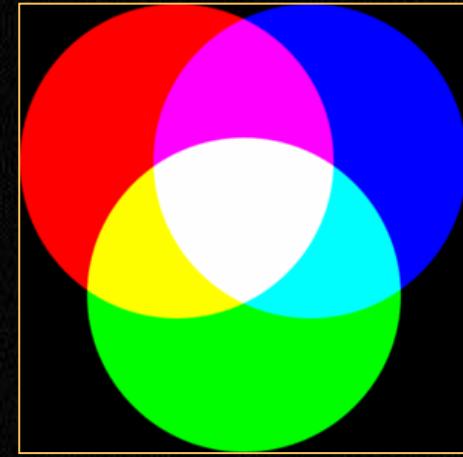


- ▶ Ogni canale contiene la luminosità di un colore primario (Red, Green, Blue)
- ▶ Sommando la luminosità dei tre colori primari si ottengono i vari colori



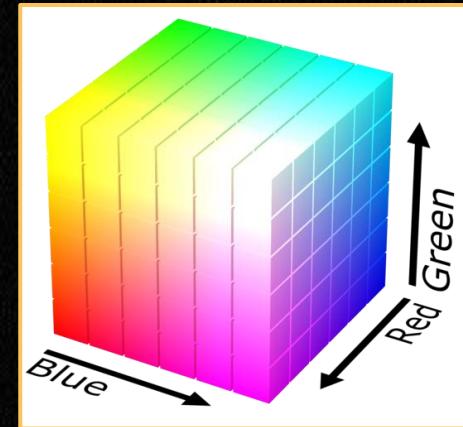
► Modello additivo

- I colori sono ottenuti mediante combinazione dei 3 colori primari Red, Green, Blue
- Il più utilizzato in informatica per la semplicità con cui si generano i colori



► Spazio RGB

- Ogni colore può essere considerato come un punto in uno spazio a tre dimensioni
- Non idoneo per il raggruppamento spaziale di colori percepiti come simili dall'uomo



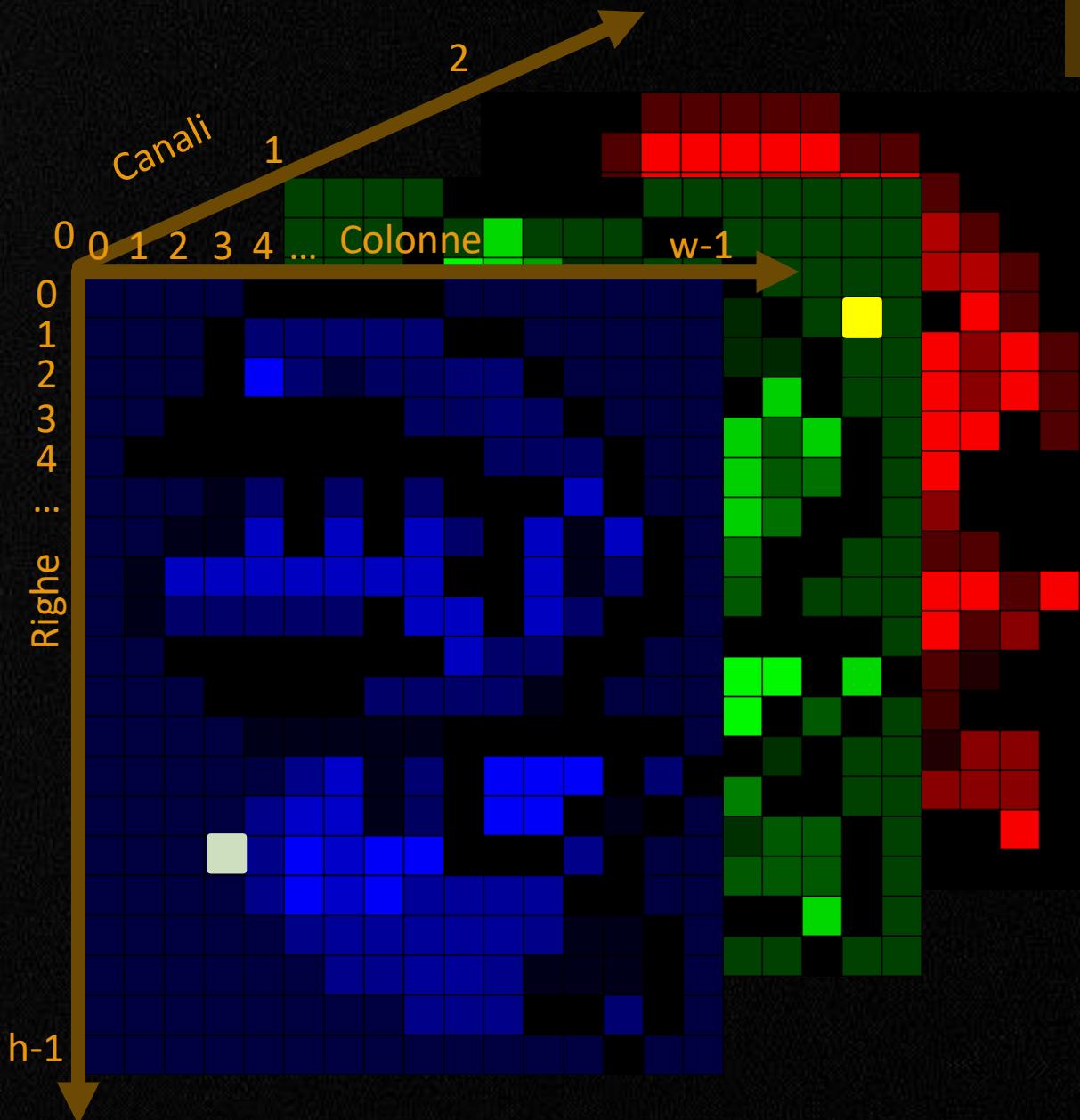
Coordinate dei pixel

► Coordinate cartesiane: x, y, canale

- $\text{pixel_pos} = (14, 3, 1)$

► Notazione matriciale: r, c, canale

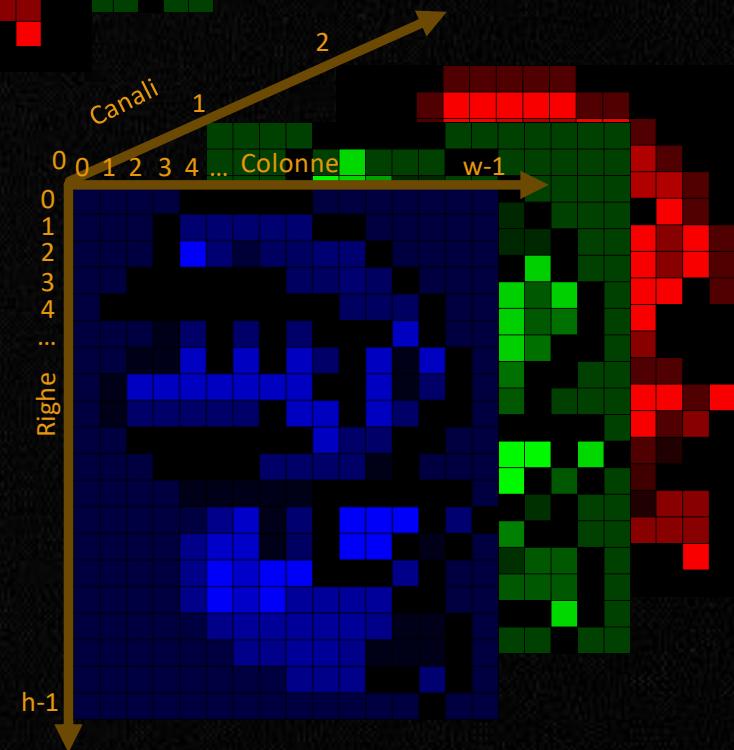
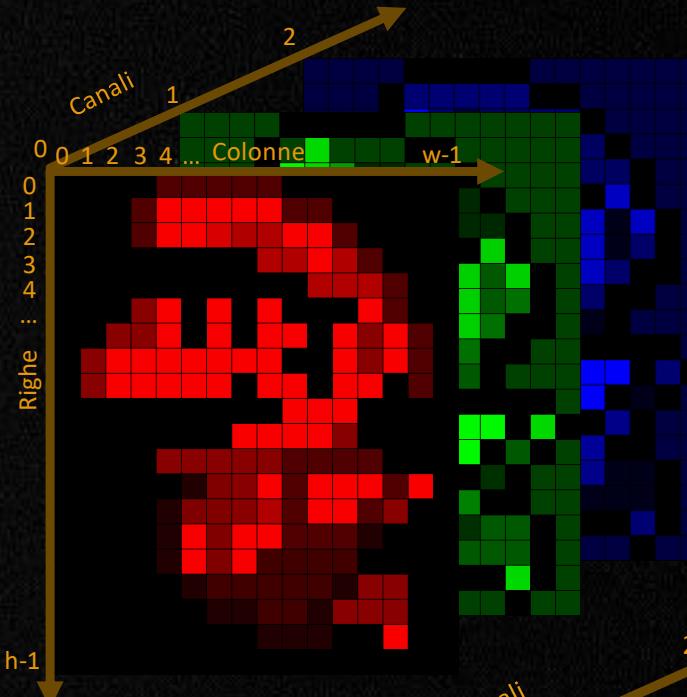
- $\text{img}[14, 3, 0] = 42$



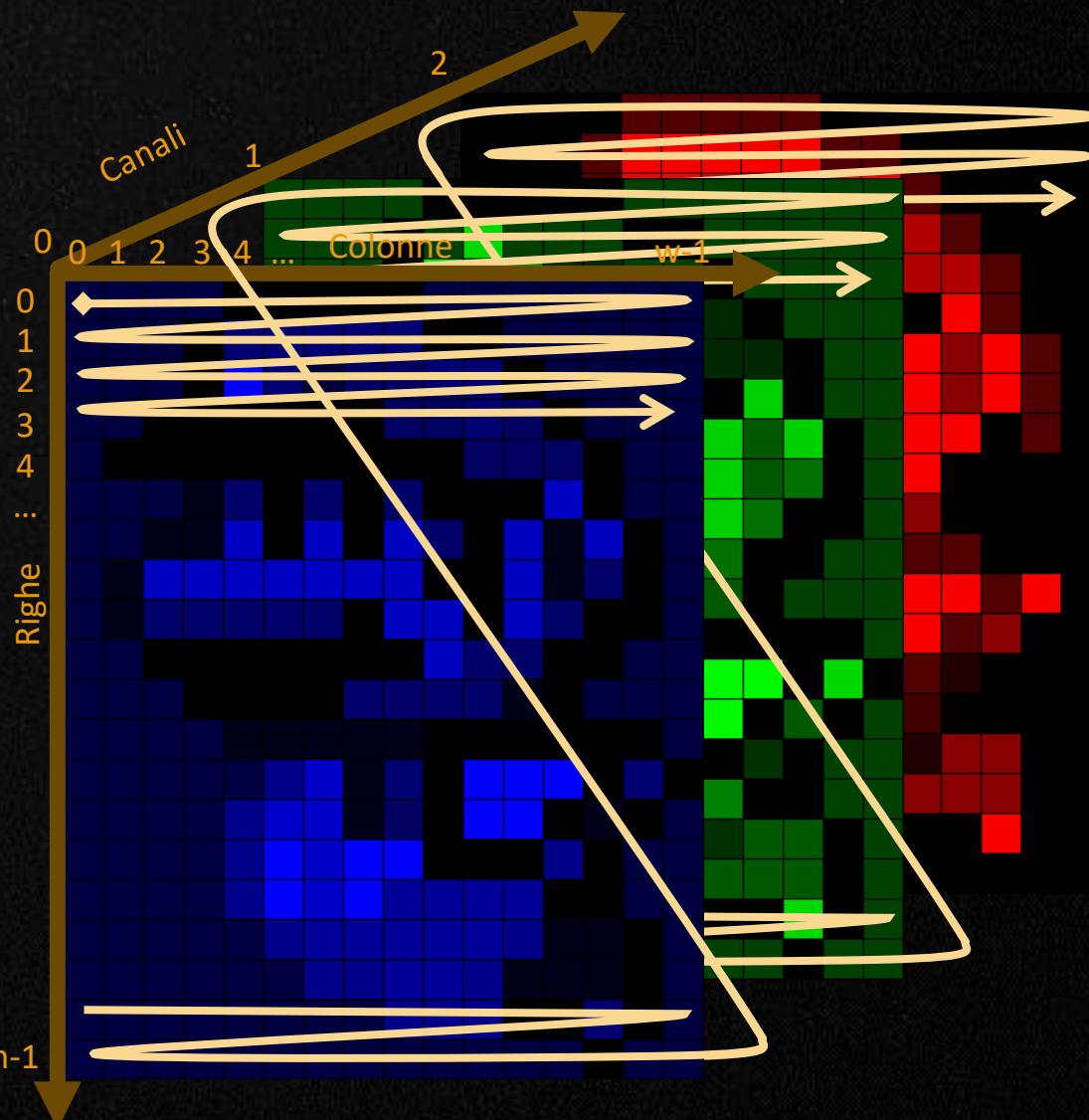
14

Ordine dei canali

- ▶ Il più usato è RGB
(R=0, G=1, B=2)
- ▶ Attenzione: OpenCV utilizza BGR
(B=0, G=1, R=2)

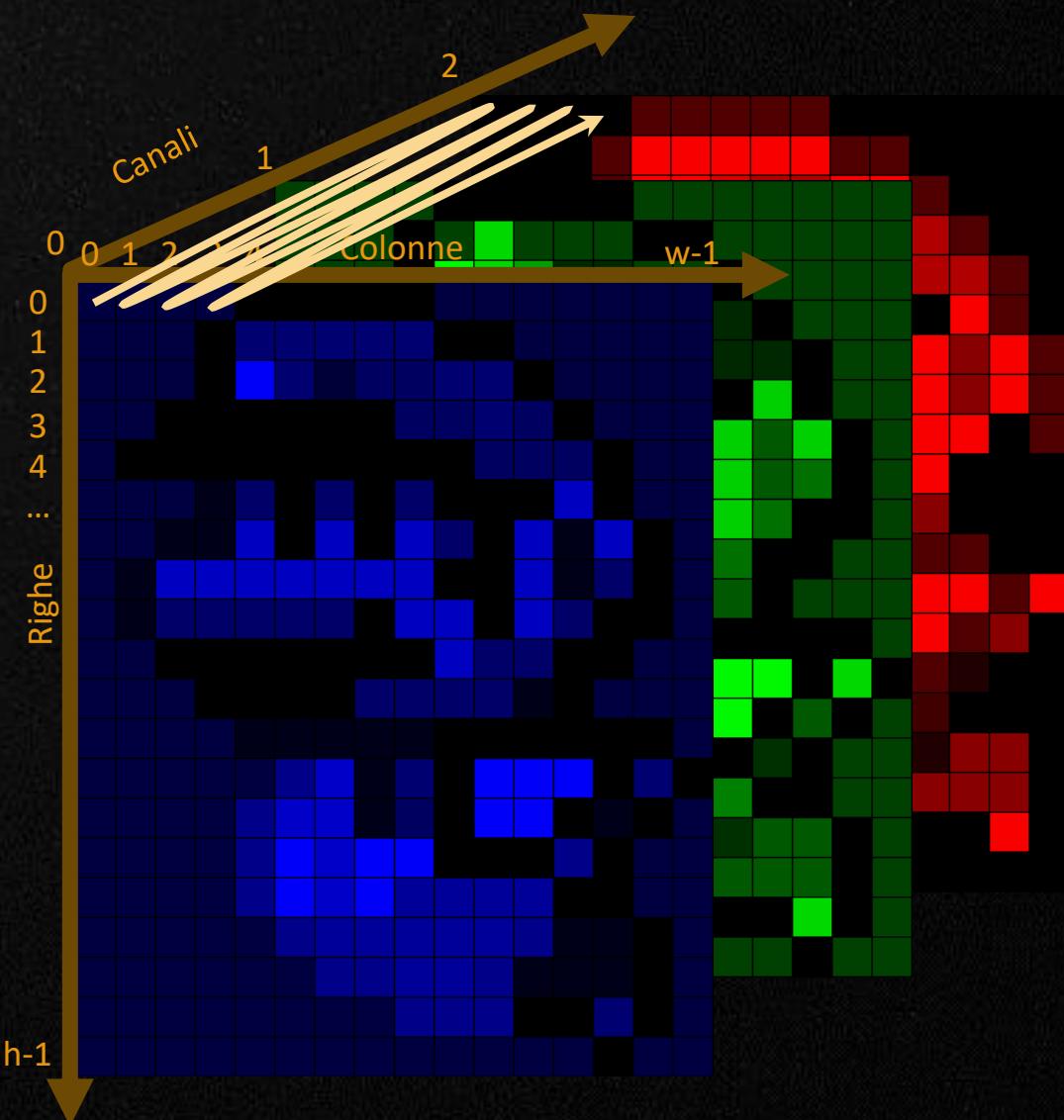


Organizzazione dei pixel in memoria: BBB...GGG...RRR



Organizzazione dei pixel in memoria: BGRBGR...BGR

17



Immagini a colori in Python/OpenCV: array NumPy tridimensionali

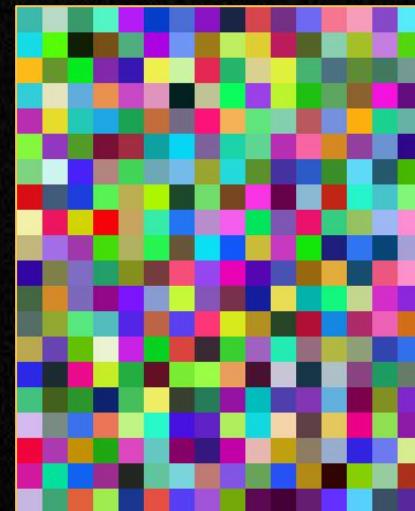
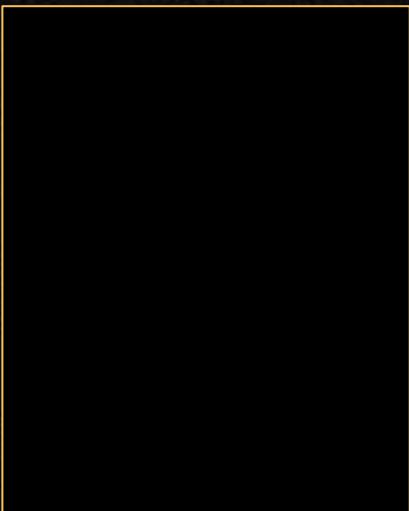
18

```
# Creazione di un'immagine 16x20 a 3 canali con valori di tipo byte (3 byte per pixel)

img1 = np.zeros((20, 16, 3), dtype=np.uint8) # Pixel a 0
img2 = np.full((20, 16, 3), 255, dtype=np.uint8) # Pixel a 255
img3 = np.random.randint(0, 256, (20, 16, 3), dtype=np.uint8) # Pixel casuali

# Caricamento di un'immagine da file (formato BGR)
# N.B.: se il file non esiste non genera un errore ma restituisce None

img4 = cv.imread('esempi/mario-c.png')
```



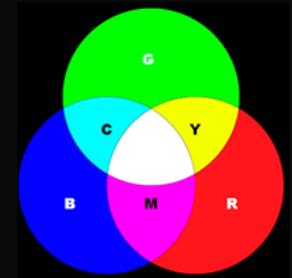
Immagini a colori in Python/OpenCV: array NumPy tridimensionali

19

```
# Caricamento di un'immagine da file (formato BGR)
m = cv.imread('esempi/mario-c.png')
```

```
# Crea tre immagini BGR ciascuna con valori solo in un canale e gli altri due a zero
b, g, r = m.copy(), m.copy(), m.copy()
b[...,1:3], g[...,0:3:2], r[...,0:2] = 0, 0, 0
```

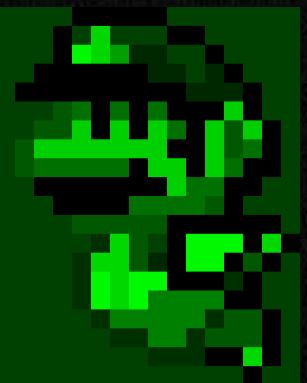
```
# Crea tre immagini BGR ciascuna corrispondente alla somma di due canali
c, m, y = b+g, b+r, g+r
```



m



b



g



r



c



m



y



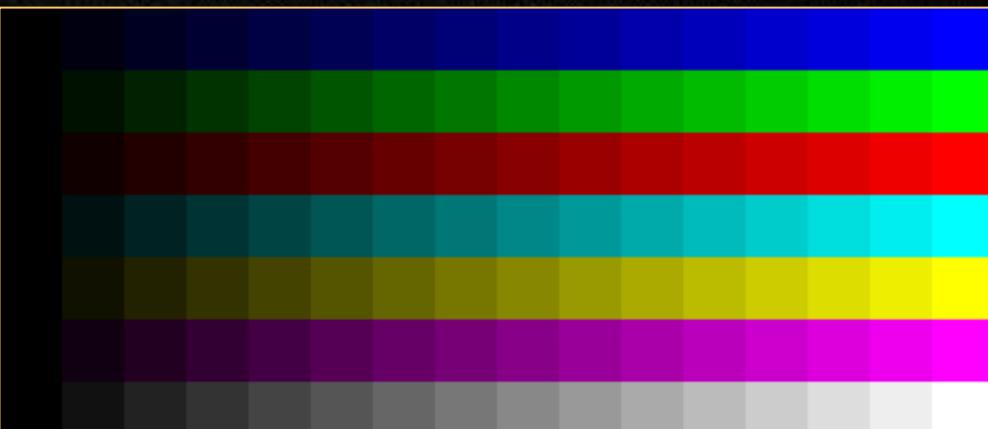
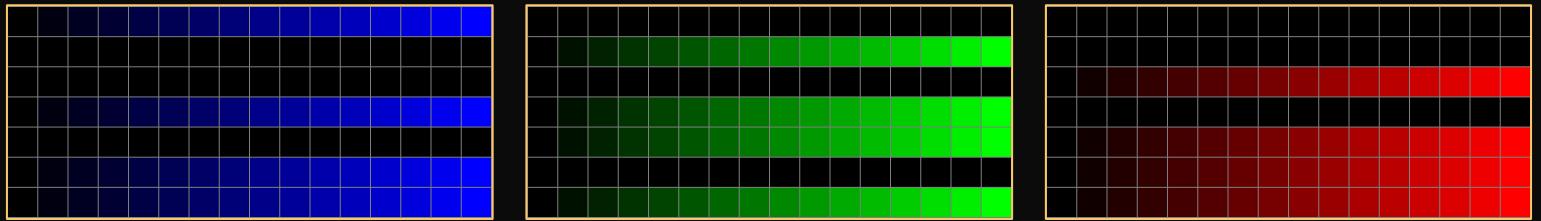
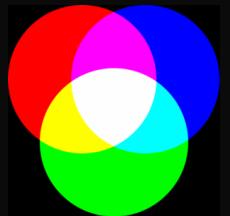
Immagini a colori in Python/OpenCV: array NumPy tridimensionali

20

```
# Crea l'array [0  17  34  51  68  85 102 119 136 153 170 187 204 221 238 255]
a = np.arange(0, 256, 17, dtype=np.uint8)

# Immagine BGR con 7 righe e tante colonne quanti sono gli elementi di a
img = np.zeros((7, a.size, 3), dtype=np.uint8)

# Combinazione di integer array indexing, slicing e broadcasting
img[[0,3,5,6],:,0] = a
img[[1,3,4,6],:,1] = a
img[[2,4,5,6],:,2] = a
```



img



Immagini a colori in Python/OpenCV: array NumPy tridimensionali

21

```
t = cv.imread('immagini/toys.png')
```

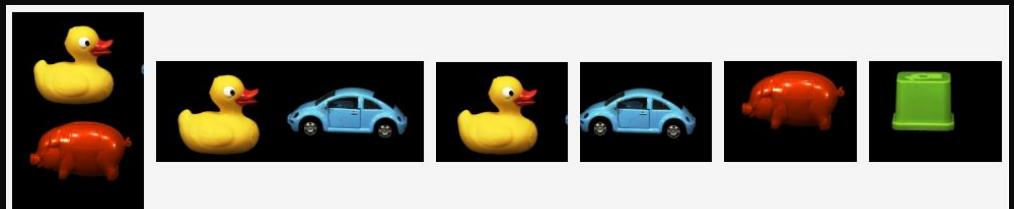


```
f = (t[::-1], t[:,::-1], t[::-1,::-1], t[...,:-1])
```

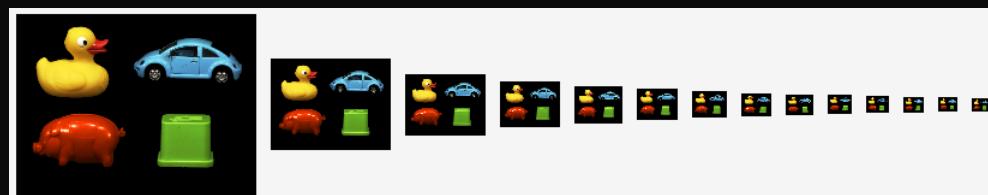


```
h, w = t.shape[:2]  
h2, w2 = h//2, w//2
```

```
s = (t[:, :w2], t[:h2], t[:h2, :w2], t[:h2, w2:], t[h2:, :w2], t[h2:, w2:])
```

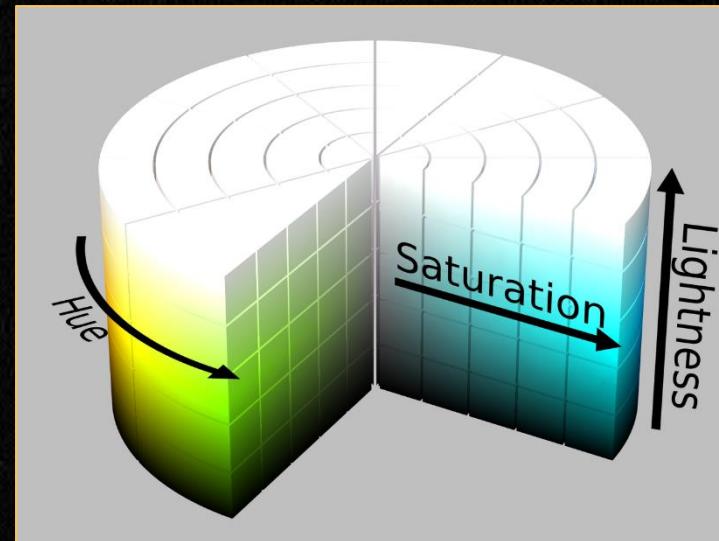
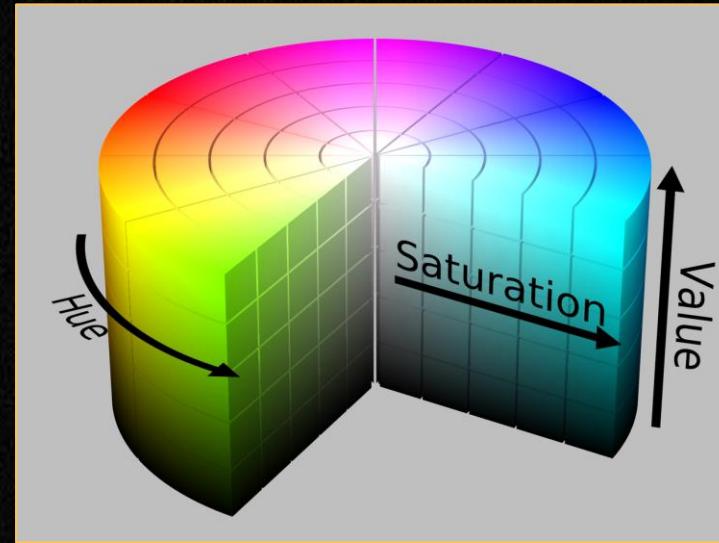


```
r = [t[::-k, ::k] for k in range(2, 30, 2)]
```



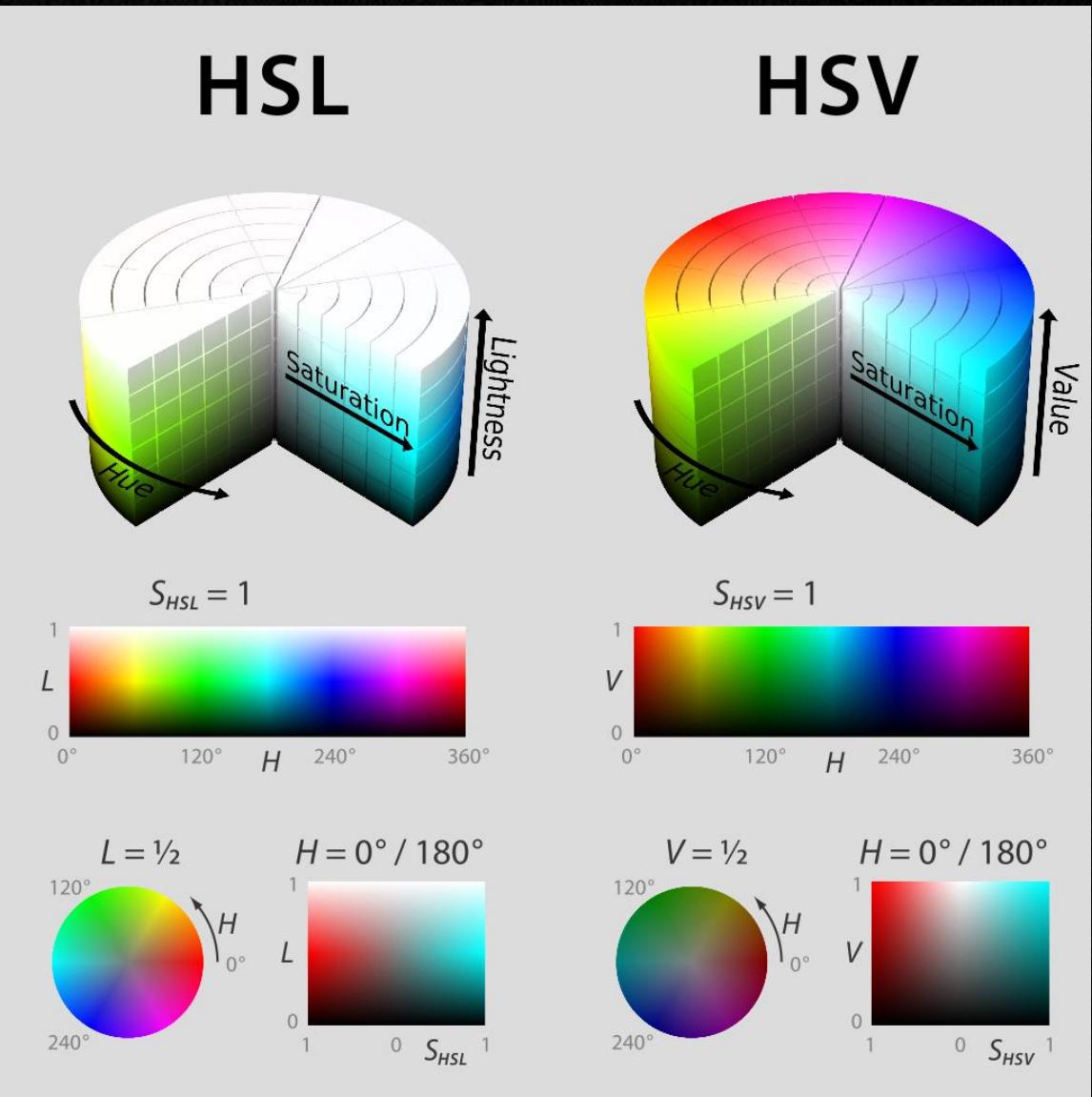
Rappresentazioni HSV e HSL

- ▶ Più vicine al modo con cui gli esseri umani percepiscono i colori.
- ▶ Basate su:
 - Tinta (Hue)
 - Saturazione
 - Luminosità (Value o Lightness)
- ▶ Vantaggi
 - Possibilità di specificare i colori in modo intuitivo
 - Possono essere utilizzate più efficacemente per localizzazione e riconoscimento di oggetti nelle immagini



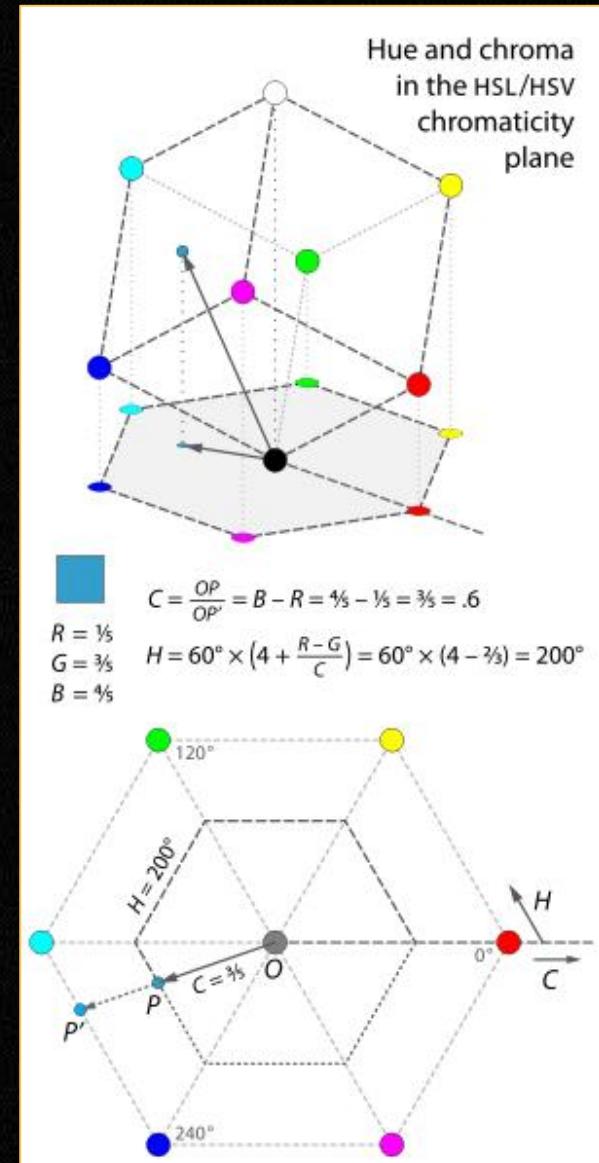
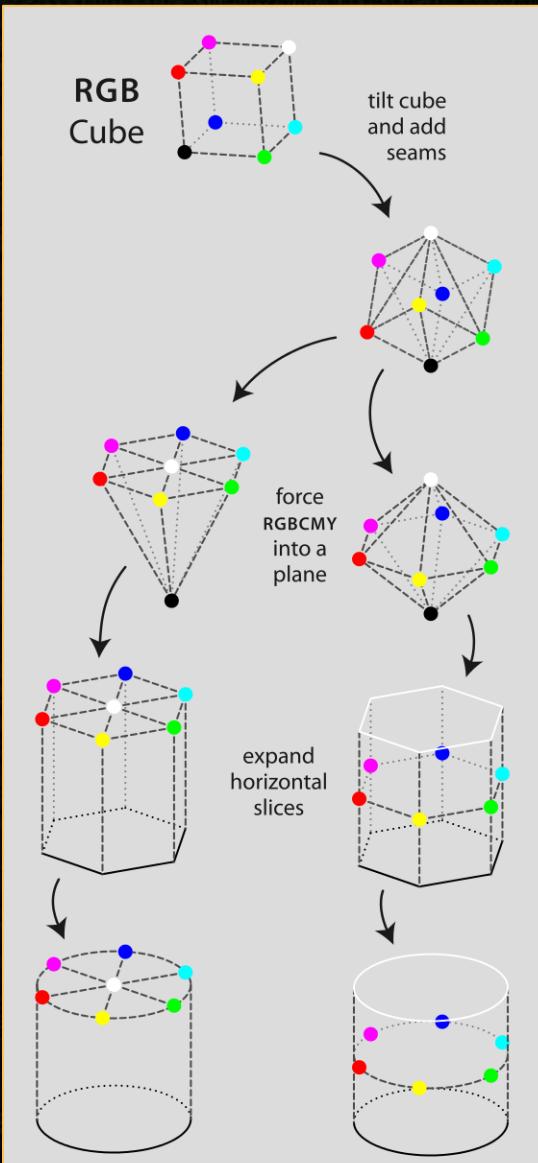
Rappresentazioni HSV e HSL

- Lo Hue è un angolo: rosso primario a 0° , verde primario a 120° e blu primario a 240° .
- L'asse verticale al centro comprende i grigi, dal nero (luminosità 0) al bianco (luminosità 1).
- I colori primari (RGB) e secondari (CMY) si trovano attorno al cilindro con saturazione 1.
 - In HSV questi colori saturi hanno luminosità 1
 - In HSL invece hanno luminosità 0.5



Da RGB a HSV/HSL

- ▶ Il cubo RGB è ruotato in modo da lasciare il vertice nero all'origine e quello bianco sopra di esso lungo l'asse verticale.
- ▶ HSV: tutti gli altri vertici sono posti sullo stesso piano, compreso il bianco al centro.
- ▶ HSL: gli altri vertici, escluso il bianco, sono posti sullo stesso piano.
- ▶ Espansione del vertice nero (HSV)
- ▶ Espansione dei vertici nero e bianco (HSL)
- ▶ Trasformazione in cilindro



Modifiche ai canali HSL: esempi



HSV e HSL in Python/OpenCV

26

- ▶ Anche in questo caso le immagini sono tensori 3D: cambia il significato dei 3 canali
- ▶ cvtColor per convertire da RGB a HSV/HSL e viceversa:
 - COLOR_HSV2BGR, COLOR_BGR2HSV
 - COLOR_HLS2BGR, COLOR_BGR2HLS
- ▶ Attenzione al caso HSL: in OpenCV l'ordine è HLS (H=0, L=1, S=2)
- ▶ Intervalli di valori per immagini di byte:
 - H: [0..179], corrispondente a [0.. 2π]
 - S, V/L: [0..255], corrispondente a [0..1]
- ▶ Per visualizzare o salvare le immagini, è necessario convertirle in formato BGR.

```
originale = cv.imread('immagini/toys.png')
# Converte in HLS
hls = cv.cvtColor(originale, cv.COLOR_BGR2HLS)

# Separa i tre canali in 3 immagini grayscale
h, l, s = cv.split(hls)

# Modifica alcuni dei valori HLS
# Dimezza la luminosità di tutti i pixel
l1 = l//2

# Valore di saturazione 64 per tutti i pixel
s1 = np.full_like(s, 64)

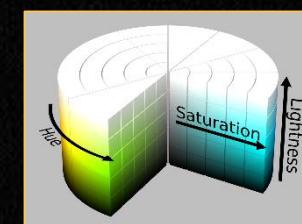
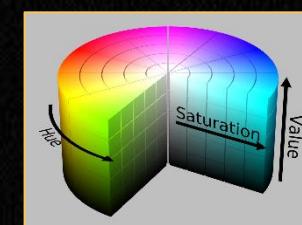
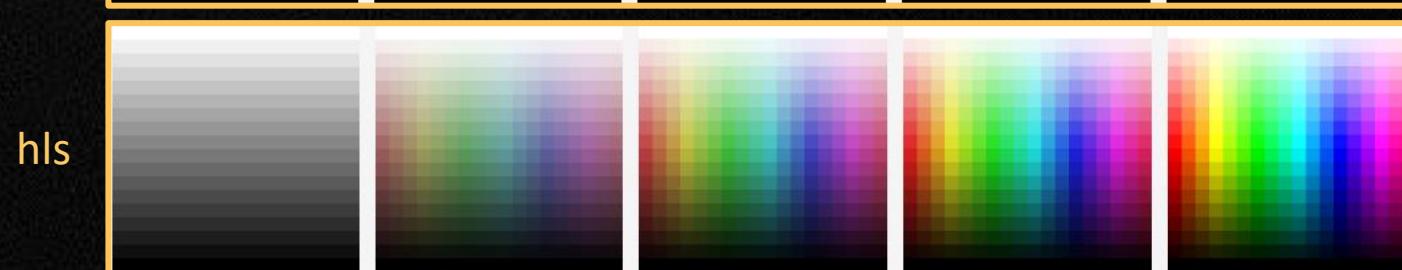
# Riunisce i canali e converte in BGR
risultato = cv.cvtColor(cv.merge((h, l1, s1)),
cv.COLOR_HLS2BGR)
```



HSV e HSL in Python/OpenCV

27

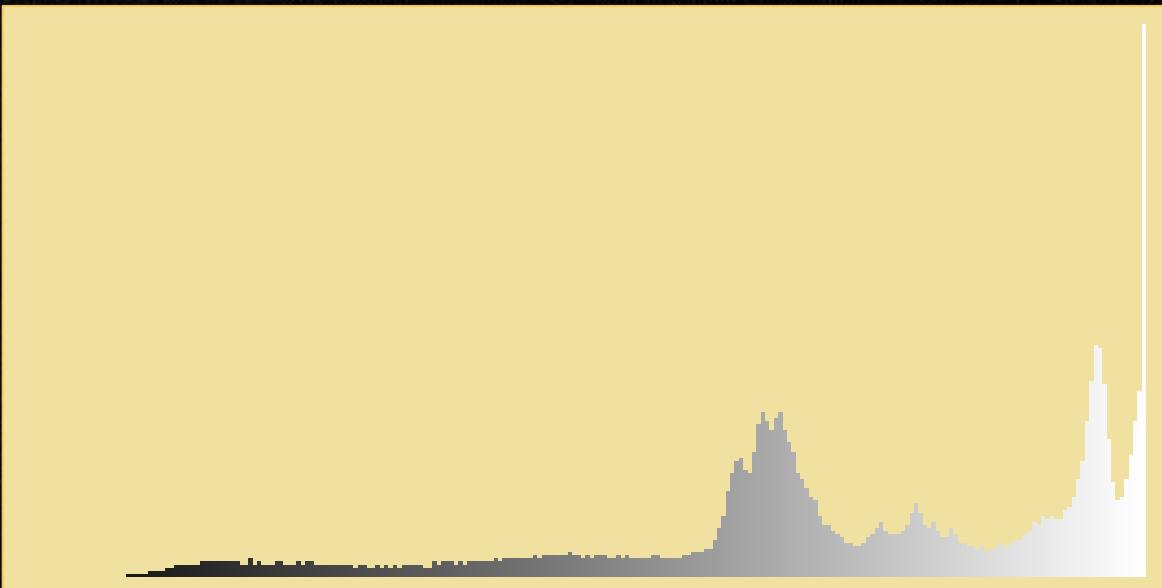
```
s = [0, 64, 128, 192, 255] # 5 livelli di saturazione  
  
# 18 valori di luminosità in ogni colonna  
v = np.tile(np.arange(255, -1, -15, np.uint8)[:,np.newaxis], (1, 18))  
  
# 18 valori di hue in ogni riga  
h = np.tile(np.arange(0, 180, 10, np.uint8), (18, 1))  
  
# Combina i 3 canali (si poteva usare anche cv.merge)  
hsv = [np.dstack((h, np.full_like(h, x), v)) for x in s]  
  
hls = [i[...,[0,2,1]] for i in hsv] # Scambia gli ultimi due canali
```



Istogramma di un'immagine grayscale

28

- ▶ Indica il numero di pixel dell'immagine per ciascun livello di grigio
- ▶ Dall'istogramma si possono estrarre informazioni interessanti:
 - se la maggior parte dei valori solo “condensati” in una zona, l'immagine ha uno scarso contrasto
 - se nell'istogramma sono predominanti le basse intensità, l'immagine è molto scura
 - ...



Esempi di istogrammi

29



Immagine scura

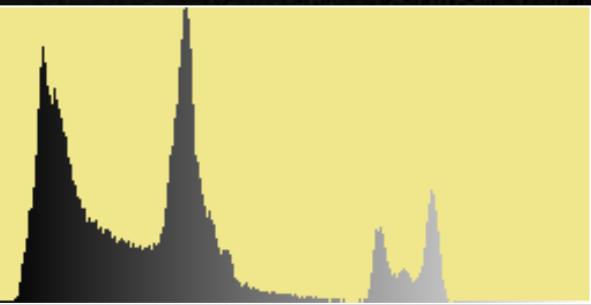


Immagine con poco contrasto

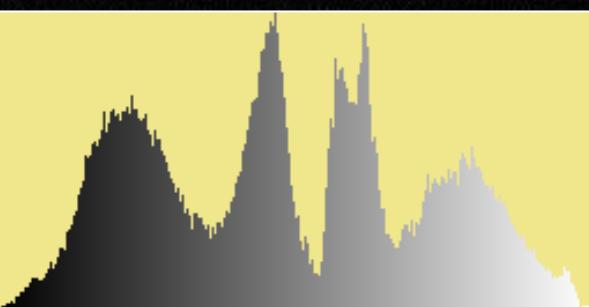
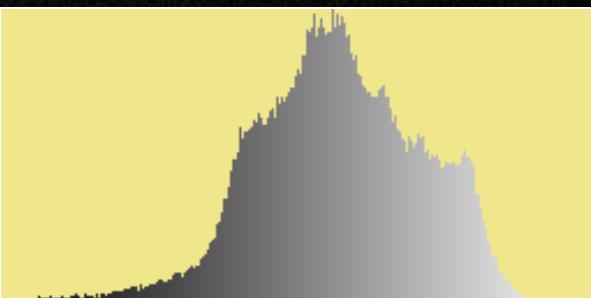


Immagine bilanciata



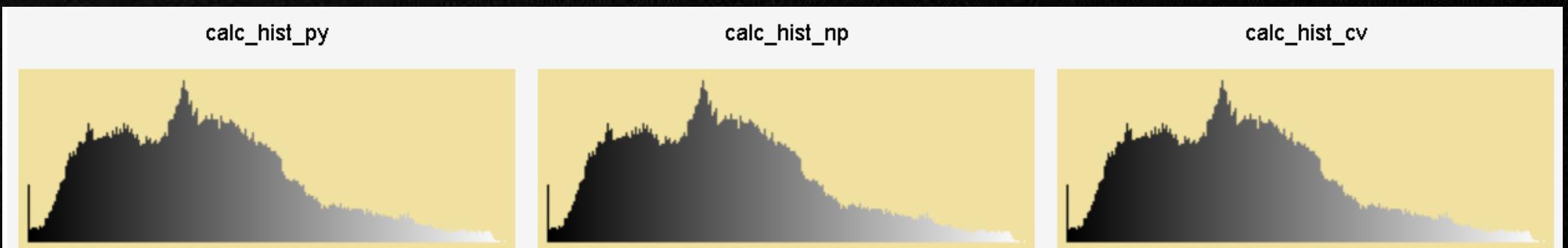
Calcolo dell'istogramma

30

```
def calc_hist_py(img):
    h = np.zeros(256, dtype=int)
    for p in np.nditer(img):
        h[p] += 1
    return h

def calc_hist_np(img):
    return np.histogram(img, 256, [0, 256])[0]

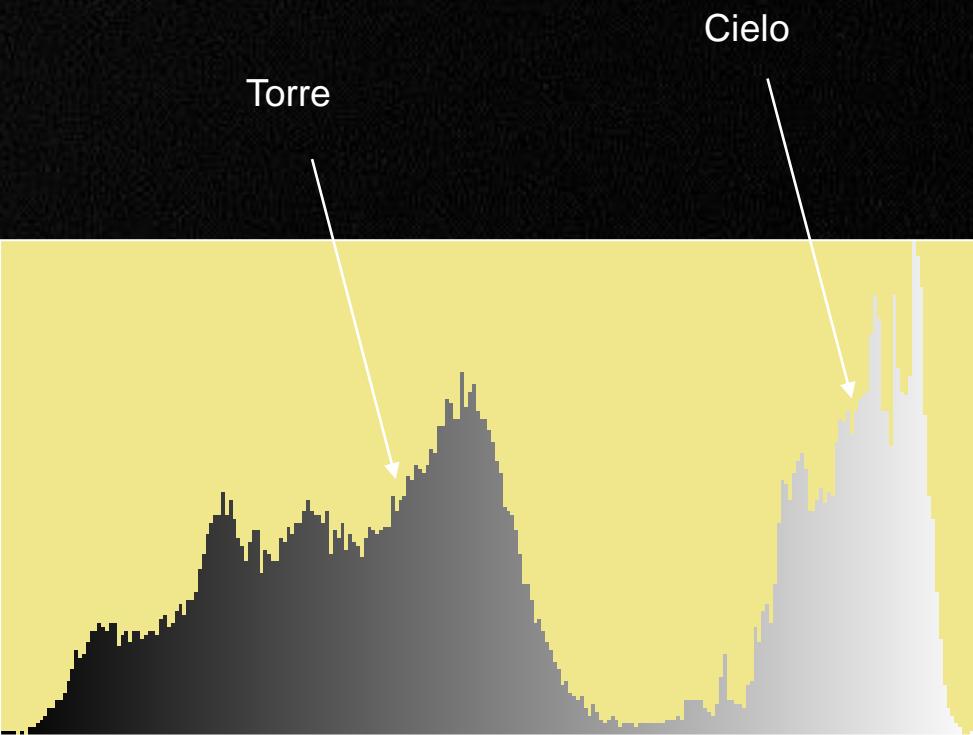
def calc_hist_cv(img):
    return cv.calcHist([img], [0], None, [256], [0, 256]).squeeze()
```



calc_hist_py: 117 ms ± 4.38 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
calc_hist_np: 3.36 ms ± 17.2 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
calc_hist_cv: 121 µs ± 1.08 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)

Analisi dell'istogramma

- Se i diversi oggetti in un'immagine hanno livelli di grigio differenti, l'istogramma può fornire un primo semplice meccanismo separazione degli oggetti dallo sfondo
 - Esempio: un istogramma bimodale può indicare la presenza di un oggetto abbastanza omogeneo su uno sfondo di luminosità pressoché costante.



- ▶ Su una singola immagine: $\mathbf{I}'[y, x] = f(\mathbf{I}[y, x])$
 - Ogni pixel dell'immagine di uscita è funzione solo del corrispondente pixel dell'immagine di input
 - Esempi principali:
 - Variazione della luminosità
 - Variazione del contrasto
 - Conversione da livelli di grigio a (pseudo)colori
 - Binarizzazione con soglia globale
- ▶ Su più immagini: $\mathbf{I}[y, x] = f(\mathbf{I}_1[y, x], \mathbf{I}_2[y, x], \dots)$
 - Ogni pixel dell'immagine di uscita è funzione solo dei corrispondenti pixel delle immagini di input
 - Esempio: operazioni aritmetiche fra immagini: somma, sottrazione, AND, OR, XOR, ...



Variazione luminosità e contrasto

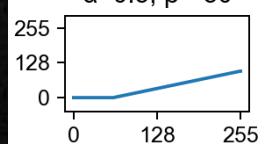
► Tipica funzione: $f(v) = \alpha \cdot v + \beta$

- α controlla il contrasto, β la luminosità
- Valori di output forzati in $[0,255]$

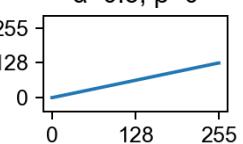
► Esempi variando α e β

- N.B. ai valori degli histogrammi è stata applicata la radice quadrata per meglio visualizzare l'effetto delle varie operazioni

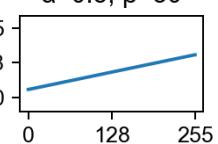
$$\alpha=0.5, \beta=-30$$



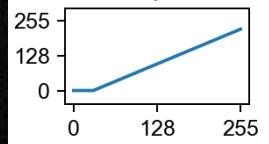
$$\alpha=0.5, \beta=0$$



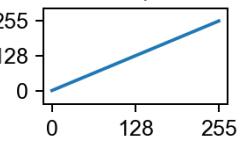
$$\alpha=0.5, \beta=30$$



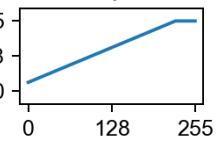
$$\alpha=1, \beta=-30$$



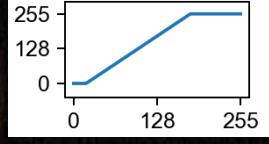
$$\alpha=1, \beta=0$$



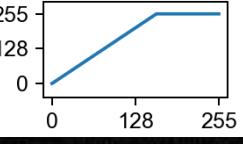
$$\alpha=1, \beta=30$$



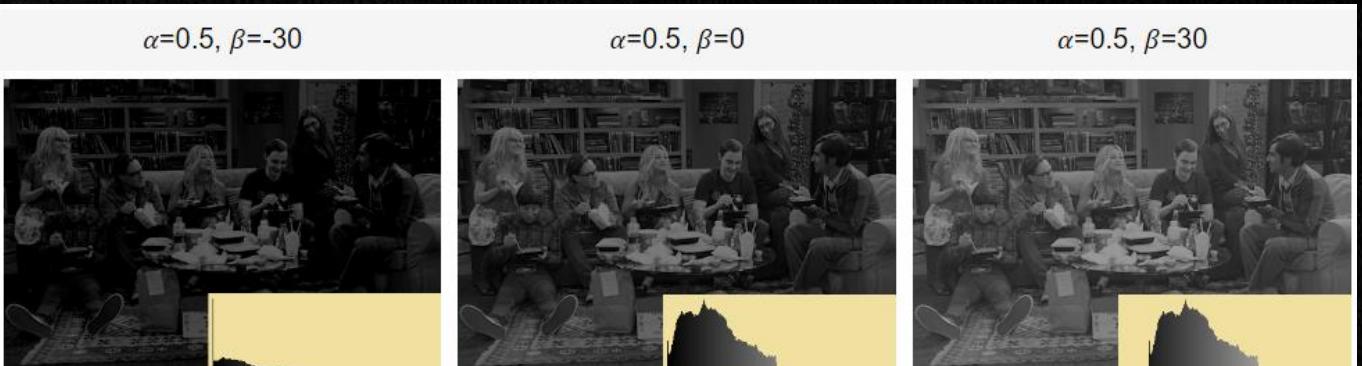
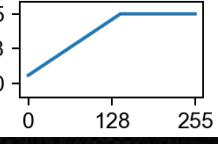
$$\alpha=1.6, \beta=-30$$



$$\alpha=1.6, \beta=0$$



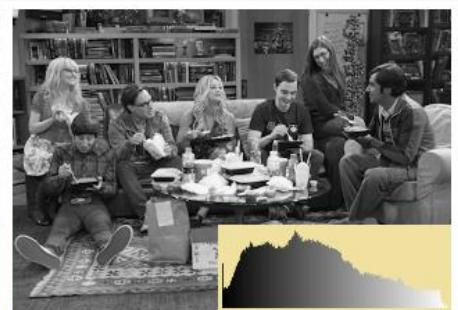
$$\alpha=1.6, \beta=30$$



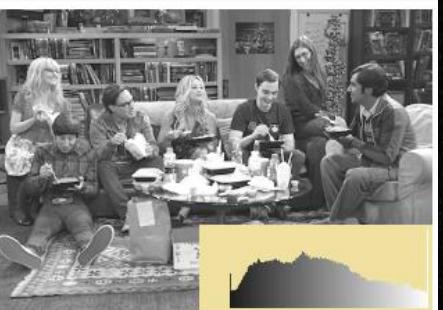
$$\alpha=1, \beta=-30$$



$$\alpha=1, \beta=0$$



$$\alpha=1, \beta=30$$



$$\alpha=1.6, \beta=-30$$



$$\alpha=1.6, \beta=0$$



$$\alpha=1.6, \beta=30$$



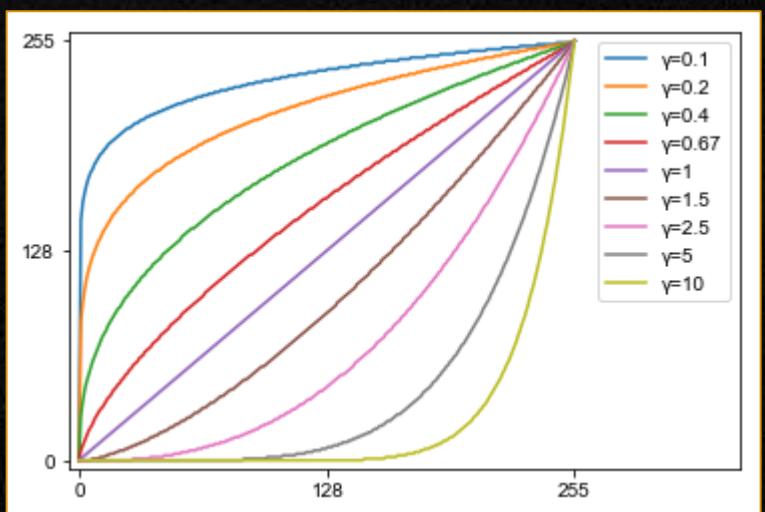
Esempio di funzione non lineare: Gamma correction

$$\blacktriangleright f(v) = \left(\frac{v}{255}\right)^{\gamma} \cdot 255$$

- $\gamma < 1$: aumenta luminosità toni scuri
- $\gamma > 1$: diminuisce luminosità toni chiari

\blacktriangleright Esempi variando γ

- Anche in questo caso, ai valori degli istogrammi è stata applicata la radice quadrata per meglio visualizzare l'effetto delle varie operazioni



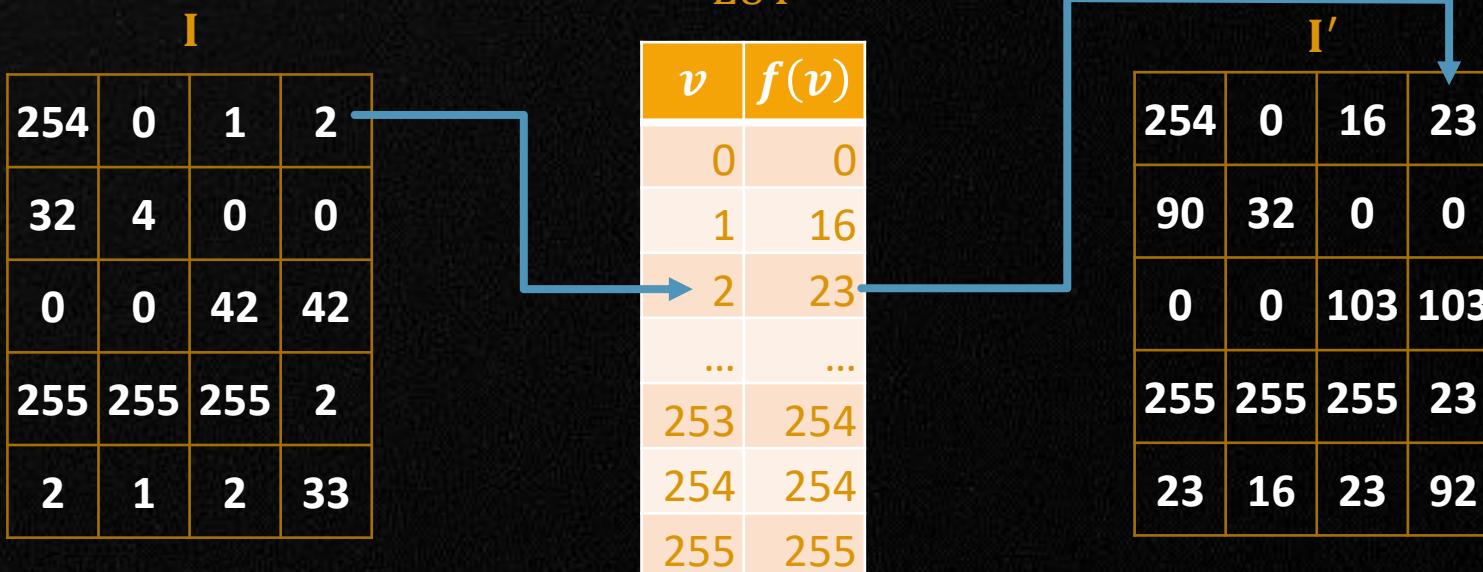
Lookup Table (LUT)

- Se il numero di colori o livelli di grigio è inferiore al numero di pixel nell'immagine, è più efficiente memorizzare il risultato della funzione di mapping f per ogni input in un array, da utilizzare poi come tabella di lookup per eseguire l'operazione su tutti i pixel.

$$\mathbf{I}'[y, x] = f(\mathbf{I}[y, x]) \longrightarrow \mathbf{I}'[y, x] = \text{LUT}[\mathbf{I}[y, x]]$$

$$f(v) = \left(\frac{v}{255} \right)^{0.5} \cdot 255$$

$$\text{LUT} = [f(i)] \\ 0 \leq i \leq 255$$



LUT in Python/NumPy/OpenCV

36

```
# Un esempio di funzione f (Gamma correction per un certo valore di γ)
γ = 0.5

# Calcolo di un singolo valore di f
f = lambda p: 255 * (p/255.0)**γ

# Calcolo di f su tutti i valori di un array NumPy
f_np = lambda a: f(a).astype(np.uint8)

# Calcolo dell'array LUT
lut = f_np(np.arange(256))

# Una semplice implementazione Python
def applica_py_f(img):
    res = np.empty_like(img)
    h, w = res.shape
    for y in range(h):
        for x in range(w):
            res[y,x] = f(img[y,x])
    return res
```



LUT in Python/NumPy/OpenCV

37

```
# Semplice implementazione Python con LUT
def applica_py_lut(img):
    res = np.empty_like(img)
    h, w = res.shape
    for y in range(h):
        for x in range(w):
            res[y,x] = lut[img[y,x]]
    return res

# Implementazione NumPy senza LUT
def applica_np_f(img):
    return f_np(img)

# Implementazione NumPy con LUT
def applica_np_lut(img):
    return lut[img]

# Funzione LUT di OpenCV
def applica_cv_lut(img):
    return cv.LUT(img, lut)
```

Metodo	Tempo (ms)	Speed-up
py_f	823,00	1
py_LUT	88,50	9
np_f	3,04	271
np_LUT	0,58	1419
cv_LUT	0,10	8230



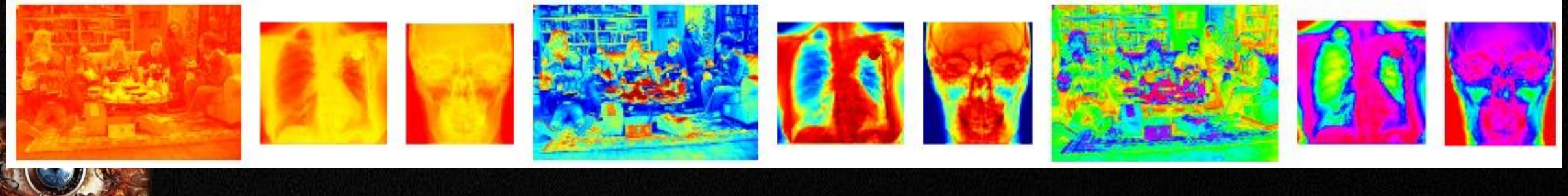
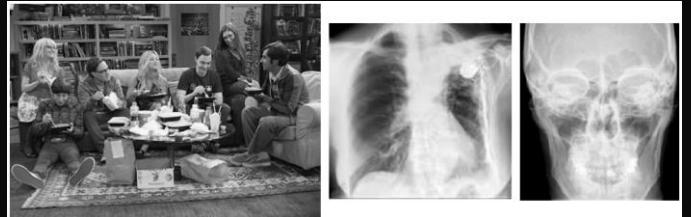
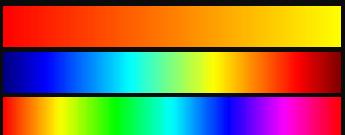
LUT da grayscale a RGB

38

- ▶ Percezione umana non adatta a osservare piccole variazioni fra toni di grigio
 - I nostri occhi sono più sensibili a variazioni fra colori
 - In molte applicazioni si ricolorano immagini grayscale per renderle meglio fruibili
- ▶ OpenCV mette a disposizione una funzione apposita (`applyColorMap`) e anche una serie di mappe di colori già pronte all'uso

```
imgs = [cv.imread('immagini/'+n, cv.IMREAD_GRAYSCALE)
        for n in ('tbbt.jpg', 'radio1.png', 'radio2.png')]
maps = (cv.COLORMAP_AUTUMN, cv.COLORMAP_JET, cv.COLORMAP_HSV)

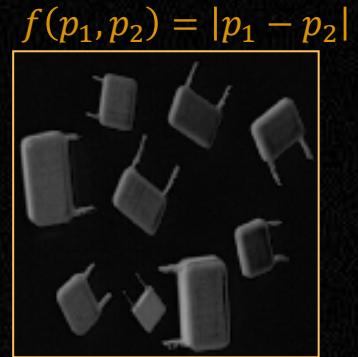
colored_imgs = [cv.applyColorMap(i, m)
                for m in maps for i in imgs]
```



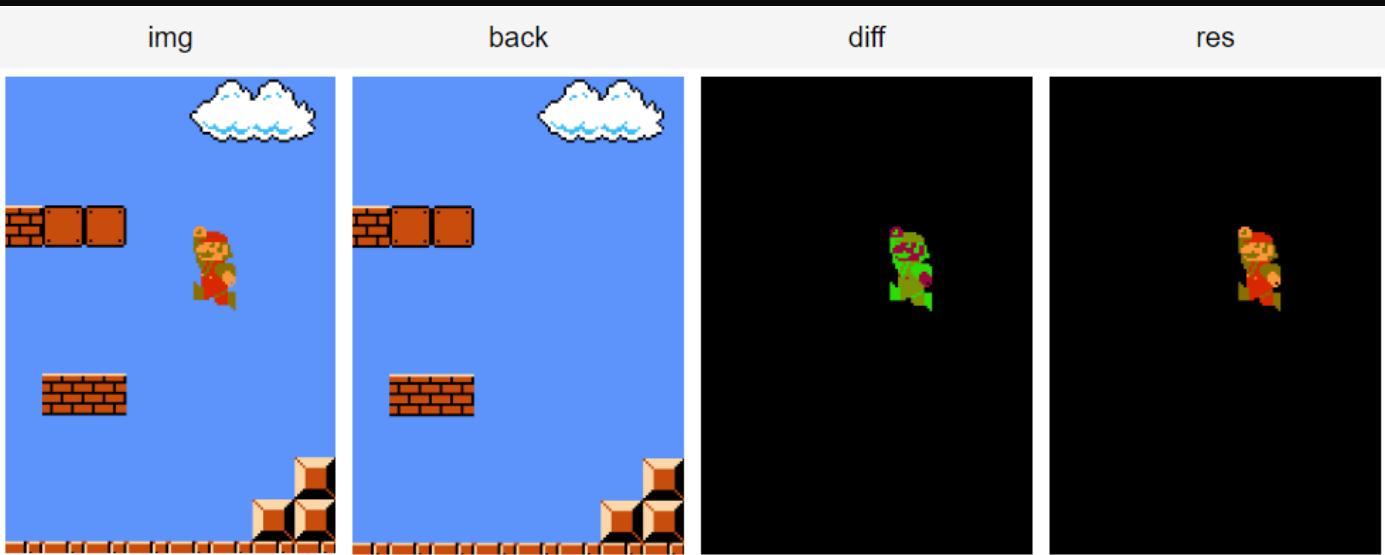
Operazioni aritmetiche fra immagini: differenza

- ▶ La sottrazione dello "sfondo" può consentire di individuare gli oggetti di interesse

$$\mathbf{I}[y, x] = f(\mathbf{I}_1[y, x], \mathbf{I}_2[y, x])$$



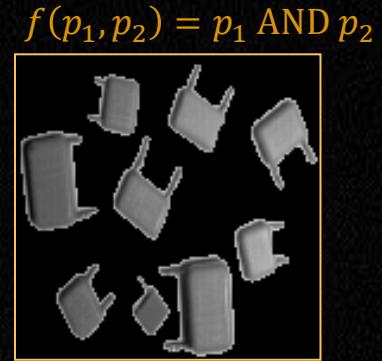
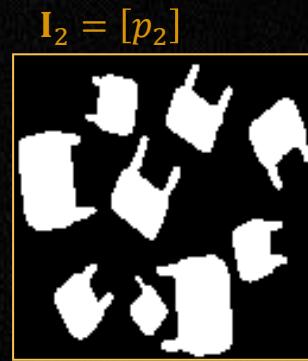
```
img, back = cv.imread('esempi/mario-game.png'), cv.imread('esempi/mario-back.png')
diff = img - back # N.B. diff = cv.subtract(img, back) è più efficiente
mask = diff!=0
res = np.zeros_like(img)
res[mask] = img[mask]
```



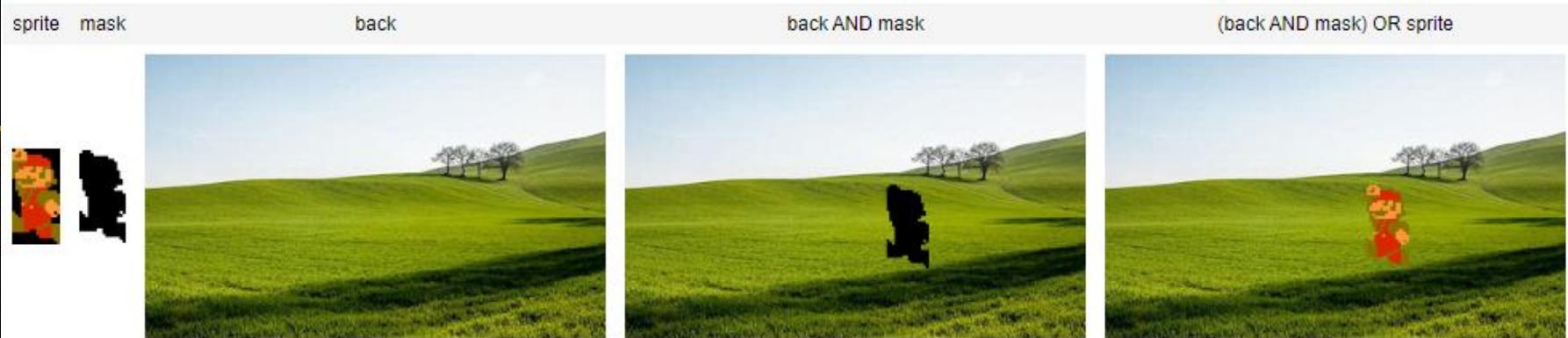
Operazioni aritmetiche fra immagini: operatori bitwise

- Analogamente alle maschere di bit, l'operatore AND consente di azzerare selettivamente alcuni i pixel, l'operatore OR consente di impostarne il valore, etc.

$$\mathbf{I}[y, x] = f(\mathbf{I}_1[y, x], \mathbf{I}_2[y, x])$$



```
sprite = cv.imread('esempi/sprite.png')
mask = cv.imread('esempi/mask.png')
back = cv.imread('esempi/hill.jpg')
res = back.copy()
x, y = 200, 100
h, w = sprite.shape[:2]
roi = res[y:y+h,x:x+w]
roi &= mask
roi |= sprite
```



Operazioni aritmetiche fra immagini: Alpha blending

41

- ▶ Combinazione fra uno sfondo (RGB) e un'immagine (RGB) con abbinato un valore di "trasparenza" per ciascun pixel (fra 0 e 1)

$$\mathbf{I}[y, x] = f(\mathbf{I}_1[y, x], \mathbf{I}_2[y, x], \mathbf{I}_3[y, x])$$

```
img1 = cv.imread('esempi/hill.jpg')
img2 = cv.imread('esempi/study.png')
img2_alpha = cv.imread('esempi/study-alpha.png')
h, w = img2.shape[:2]
a = img2_alpha / 255 # Trasforma il canale alpha da [0,255] a [0,1]

x, y = 200, 100
res = img1.copy()
res[y:y+h, x:x+w] = img1[y:y+h, x:x+w]*(1.0-a) + img2*a
```

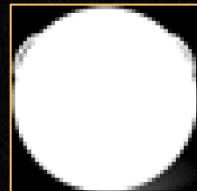
$$\mathbf{I}_1 = [p_1]$$



$$\mathbf{I}_2 = [p_2]$$



$$\mathbf{I}_3 = [a]$$



$$f(p_1, p_2, a) = p_1 \cdot (1 - a) + p_2 \cdot a$$

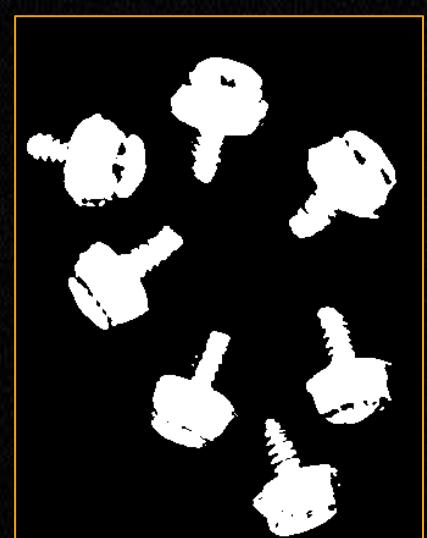
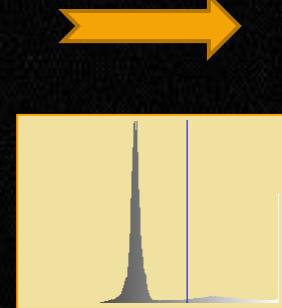
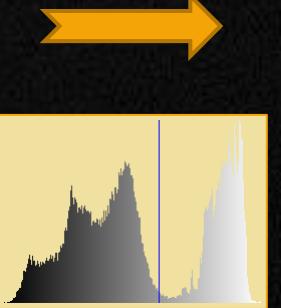


Binarizzare un'immagine grayscale: soglia globale

► Come scegliere la soglia?

- Manualmente
- Osservando l'istogramma
- Metodo di Otsu: in automatico cerca sull'istogramma la soglia che minimizza la varianza intra-classe dell'intensità dei pixel delle due classi (foreground e background) determinate dalla soglia stessa.

$$f(v, t) = \begin{cases} 0 & v < t \\ 255 & v \geq t \end{cases}$$

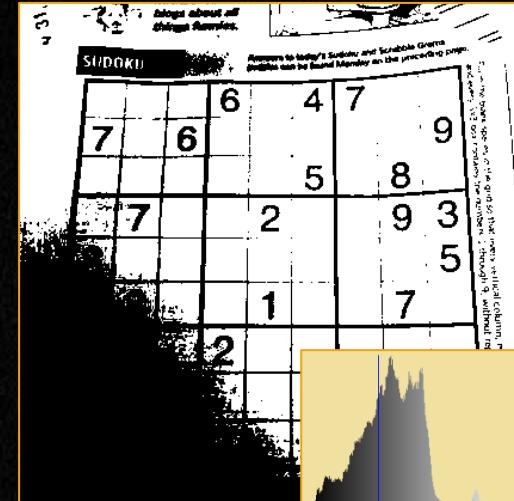


Binarizzare un'immagine grayscale: soglia locale

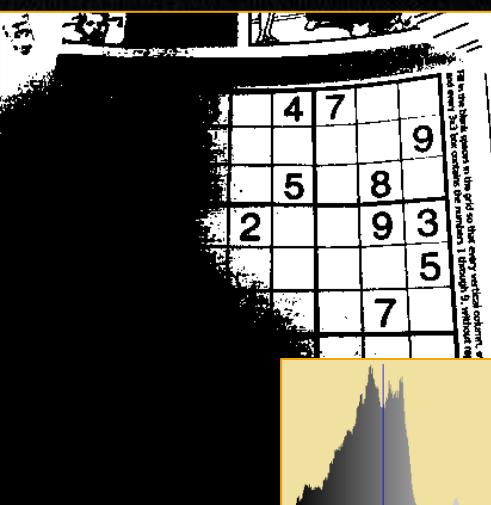
- ▶ Quando oggetti e sfondo non sono uniformi, spesso non è possibile determinare una soglia globale appropriata
 - Esempio: illuminazione non uniforme
- ▶ Soglia locale (o adattiva):
 - Determinata per ogni pixel considerando una piccola regione dell'immagine attorno ad esso
 - Il modo più semplice consiste nel determinare la soglia come media dei pixel nella regione meno un valore costante



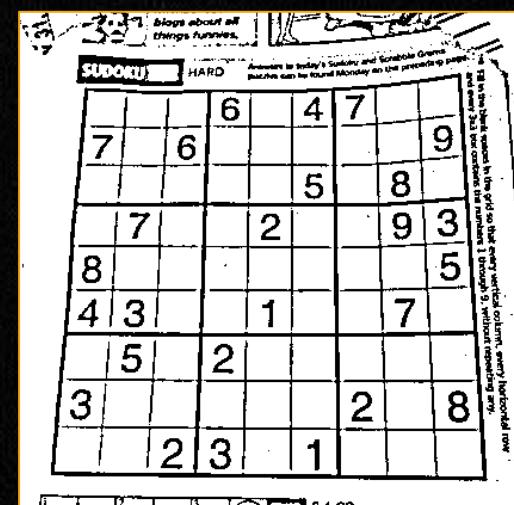
Originale grayscale



Soglia globale $t = 88$



Soglia globale $t = 118$

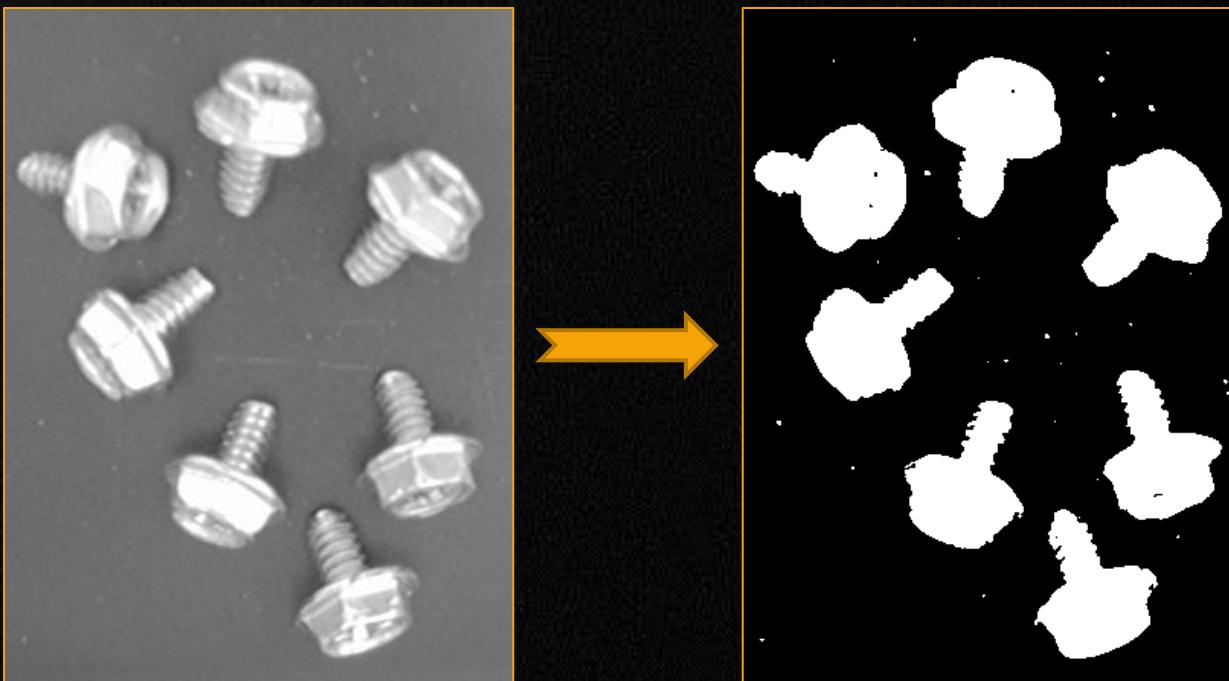


Soglia locale (media 11x11 meno 9)

Binarizzare un'immagine grayscale in OpenCV

```
# Soglia globale (128)
img = cv.imread('esempi/bolts.png', cv.IMREAD_GRAYSCALE)

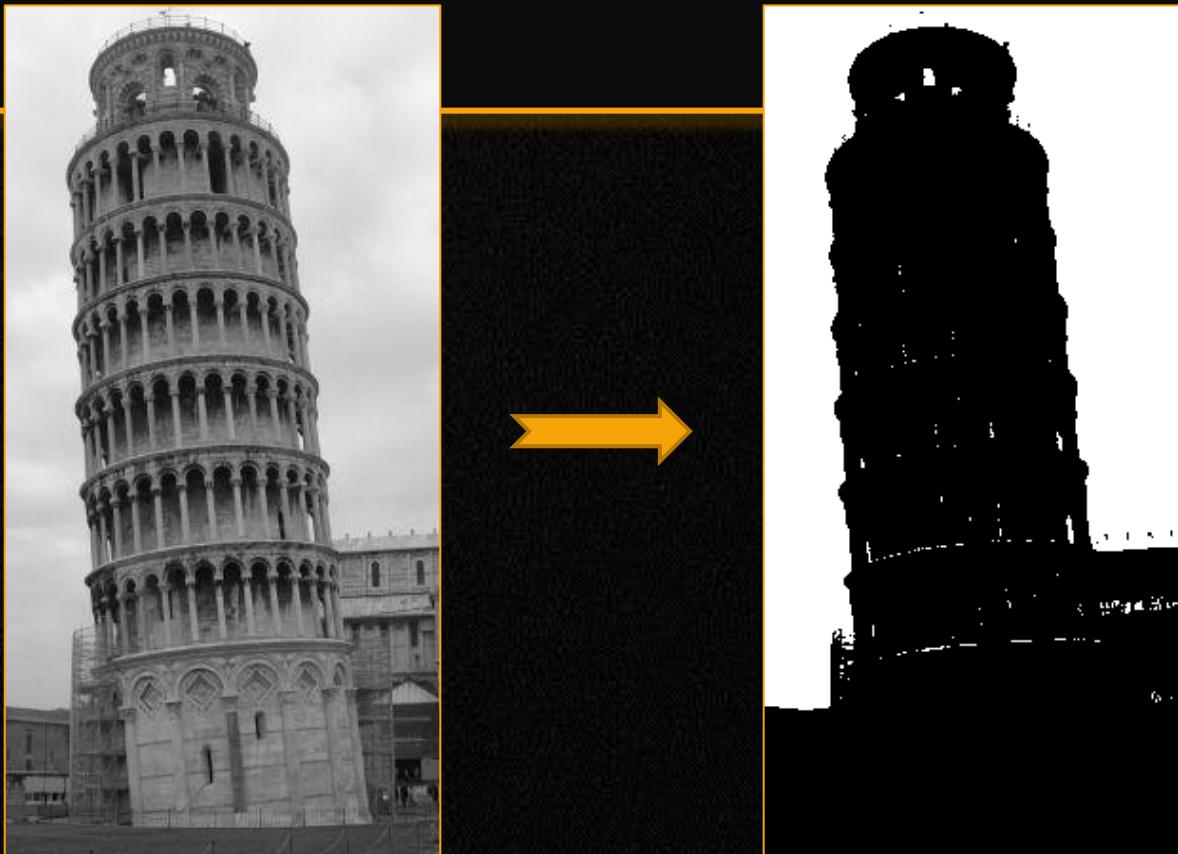
_, res = cv.threshold(img, 128, 255, cv.THRESH_BINARY)
```



Binarizzare un'immagine grayscale in OpenCV

45

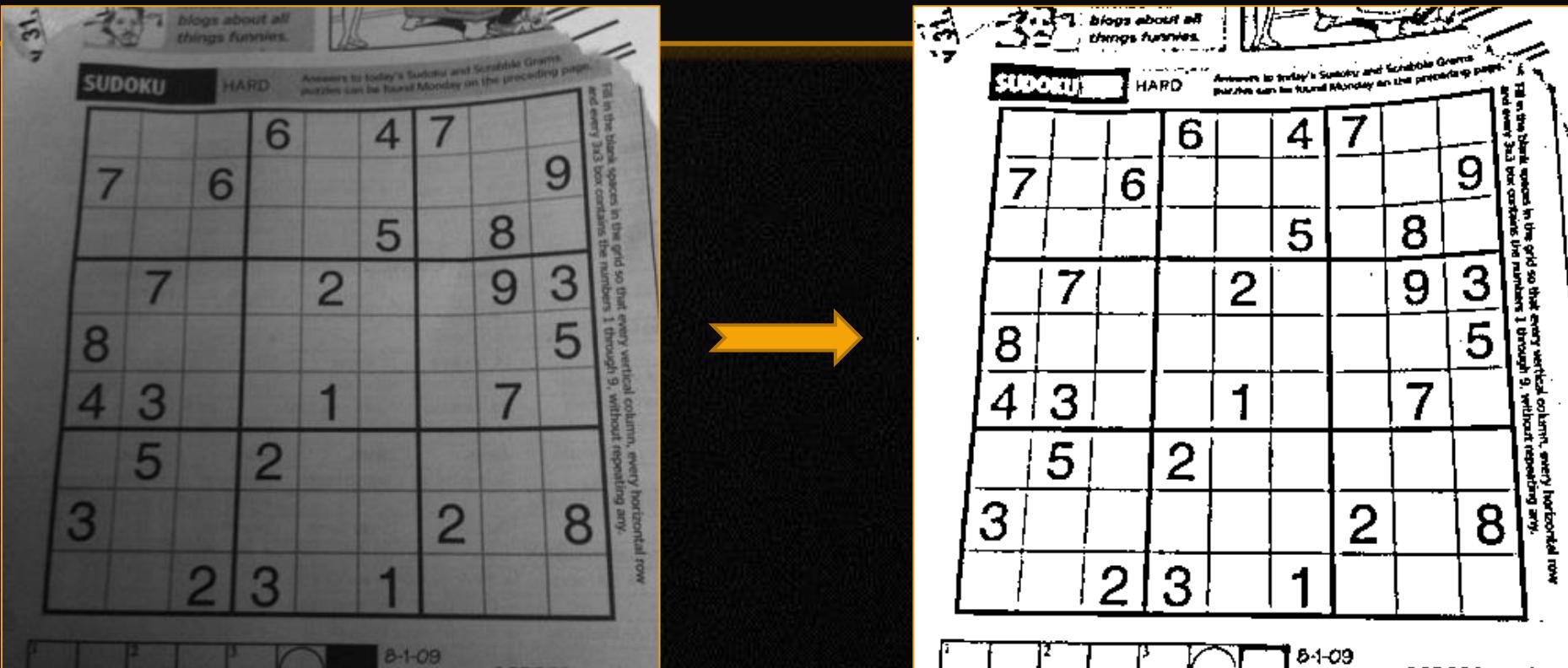
```
# Soglia globale determinata dall'algoritmo di Otsu  
img = cv.imread('immagini/torre.jpg', cv.IMREAD_GRAYSCALE)  
  
t, res = cv.threshold(img, -1, 255, cv.THRESH_OTSU)
```



Binarizzare un'immagine grayscale in OpenCV

```
# Soglia locale (media su intorno 11x11 meno il valore 10)
img = cv.imread('immagini/sudoku.jpg', cv.IMREAD_GRAYSCALE)

res = cv.adaptiveThreshold(img, 255, cv.ADAPTIVE_THRESH_MEAN_C,
                           cv.THRESH_BINARY, 11, 10)
```



Binarizzare un'immagine in OpenCV

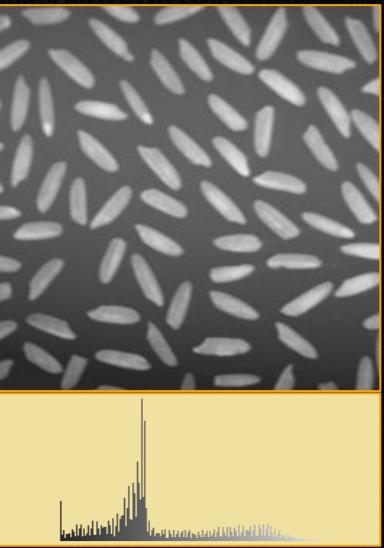
```
# Cosa succede binarizzando il canale saturazione di un'immagine HSL?  
img = cv.imread('esempi/ferrari.jpg')  
hls = cv.cvtColor(img, cv.COLOR_BGR2HLS)  
  
,hls[...,2] = cv.threshold(hls[...,2], 150, 255, cv.THRESH_BINARY)  
  
res = cv.cvtColor(hls, cv.COLOR_HLS2BGR)
```



Contrast stretching

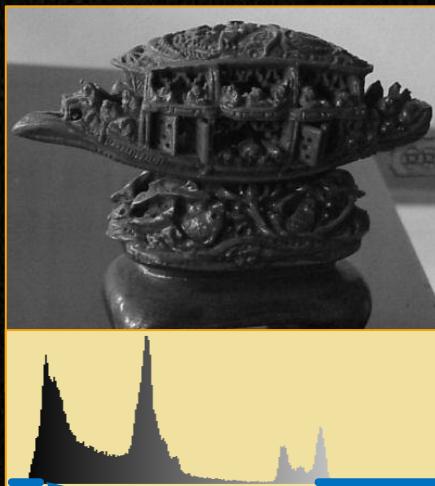
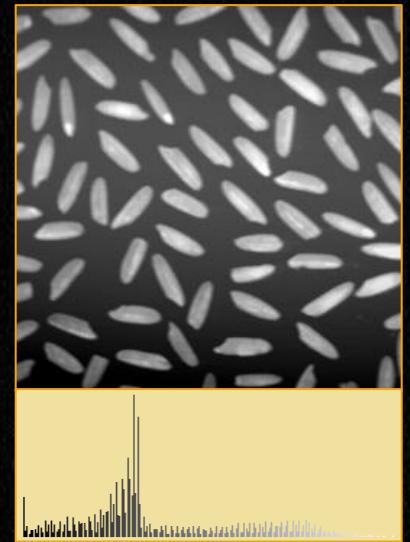
- ▶ Espansione dei livelli di grigio per aumentare il contrasto
 - Si può ottenere con un semplice mapping lineare: $f(\mathbf{I}[y, x]) = 255 \cdot \frac{\mathbf{I}[y, x] - \alpha}{\beta - \alpha}$
 - I due parametri α e β possono essere il minimo e massimo livello di grigio dell'immagine

- ▶ Problema: un solo pixel pari a 0 e uno pari a 255 rende inutile l'operazione
 - Idea: scegliere α e β sull'istogramma (ad esempio in corrispondenza del 5° e 95° percentile)



$$\alpha = \min(\mathbf{I}) = 40$$

$$\beta = \max(\mathbf{I}) = 227$$



$$\min(\mathbf{I}) = 0$$

$$\max(\mathbf{I}) = 255$$



$$\alpha = P_5(\mathbf{I}) = 17$$

$$\beta = P_{95}(\mathbf{I}) = 185$$



5% pixel più scuri

$P_5(\mathbf{I})=17$

$P_{95}(\mathbf{I})=185$

5% pixel più chiari



Contrast stretching in Python/OpenCV

49

```
def contrast_stretching(img, a, b):

    # Converte in floating point e applica la funzione di mapping
    d = b-a if b!=a else 1
    n = 255 * (img.astype(float)-a) / d

    # Forza il range [0,255] e converte in byte
    return np.clip(n, 0, 255).astype(np.uint8)

img = cv.imread('immagini/rice.png', cv.IMREAD_GRAYSCALE)

# Contrast stretching con  $\alpha=\min(I)$  e  $\beta=\max(I)$ 
res = contrast_stretching(img, img.min(), img.max())

img = cv.imread('esempi/kernel.png', cv.IMREAD_GRAYSCALE)

# Contrast stretching con  $\alpha=P_5(I)$  e  $\beta=P_{95}(I)$ 
res = contrast_stretching(img, np.percentile(img, 5), np.percentile(img, 95))
```



Equalizzazione dell'istogramma

► Metodo molto usato per:

- migliorare il contrasto
- rendere confrontabili immagini catturate in condizioni diverse di illuminazione

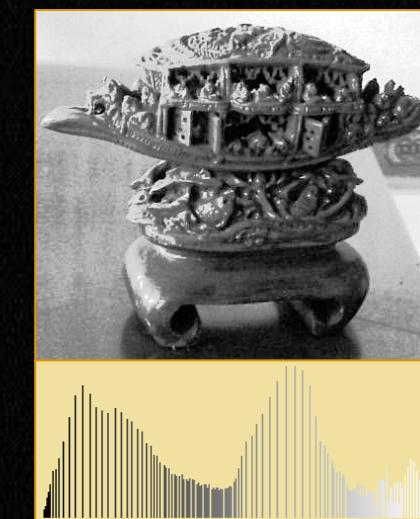
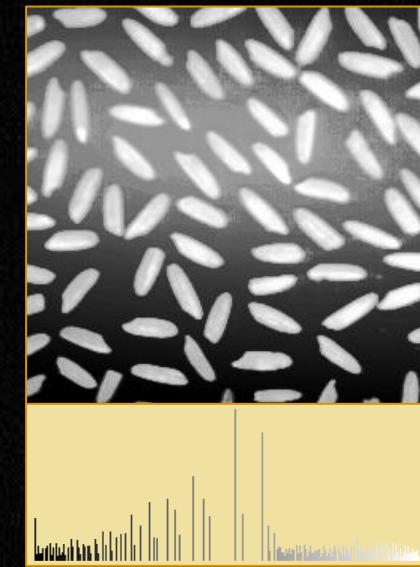
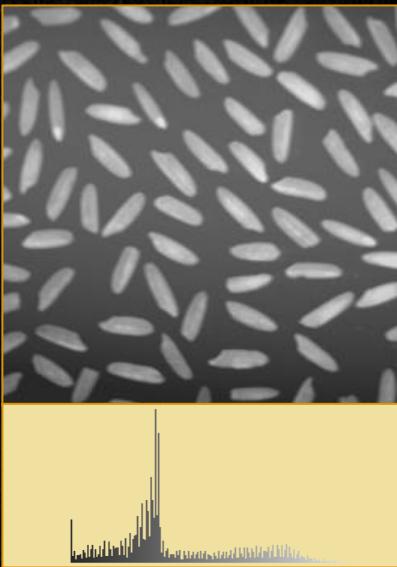
► Obiettivo (ideale):

- Distribuire l'istogramma uniformemente sui livelli di grigio
- Zone dell'istogramma "scarsamente popolate" vengono "comprese"
- Zone "con molti pixel" vengono "allargate"

$$\text{► } f(v) = \sum_{i=0}^v H[i]$$

con H istogramma dell'immagine normalizzato:

$$H[i] = \frac{255 \cdot h[i]}{\sum h[i]} \Rightarrow \sum_{i=0}^{255} H[i] = 255$$



Equalizzazione immagini a colori

51

- ▶ Non è corretto applicare separatamente l'equalizzazione a ciascun canale RGB
- ▶ Si può convertire in HSL per applicare l'equalizzazione al solo canale L



Immagine originale



Equalizzazione di ciascun canale RGB



Conversione HSL ed equalizzazione di L



Equalizzazione in Python/OpenCV

```
#Equalizzazione di un'immagine grayscale
img = cv.imread('esempi/kernel.png', cv.IMREAD_GRAYSCALE)
res = cv.equalizeHist(img)

# Equalizzazione di un'immagine a colori
# Esempio di come NON fare: equalizzazione di ciascun canale RGB
bgr = cv.imread('immagini/tbbt.jpg')
b, g, r = cv.split(bgr)
b_eq, g_eq, r_eq = [cv.equalizeHist(x) for x in (b, g, r)]
res_wrong = cv.merge((b_eq, g_eq, r_eq))

# Esempio di come procedere convertendo in HSL ed equalizzando L
hls = cv.cvtColor(bgr, cv.COLOR_BGR2HLS)
h, l, s = cv.split(hls)
l_eq = cv.equalizeHist(l)
hls_eq = cv.merge((h, l_eq, s))
res_ok = cv.cvtColor(hls_eq, cv.COLOR_HLS2BGR)
```

Numeri reali come coordinate delle immagini

- ▶ Un'immagine può essere vista come una funzione:

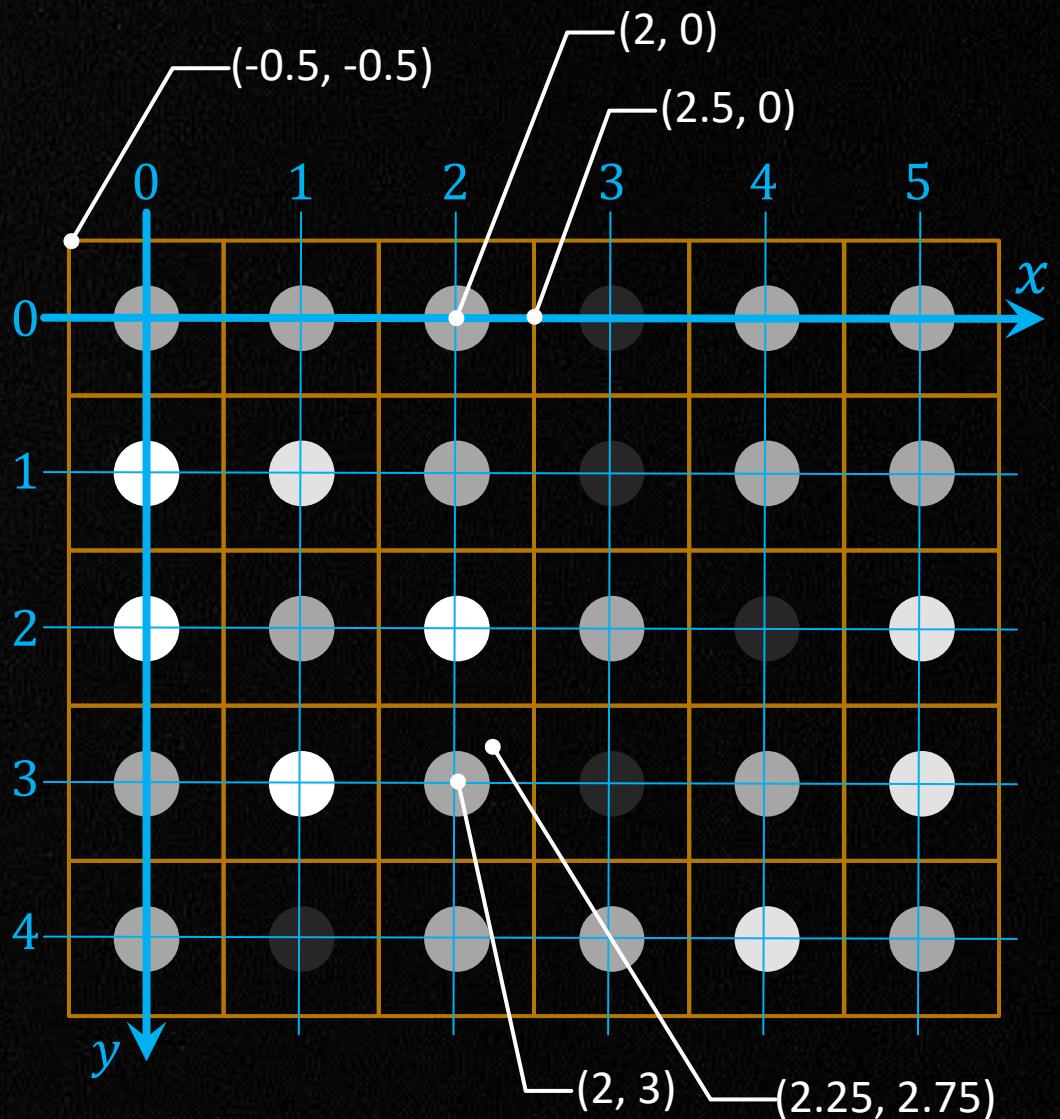
- $I: \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$
- Ad esempio, se il pixel alle coordinate $(2,0)$ del canale 1 vale 192, si ha: $I(2,0,1) = 192$

- ▶ Possiamo in generale considerare come numeri reali sia le coordinate (x, y) che i valori dei pixel:

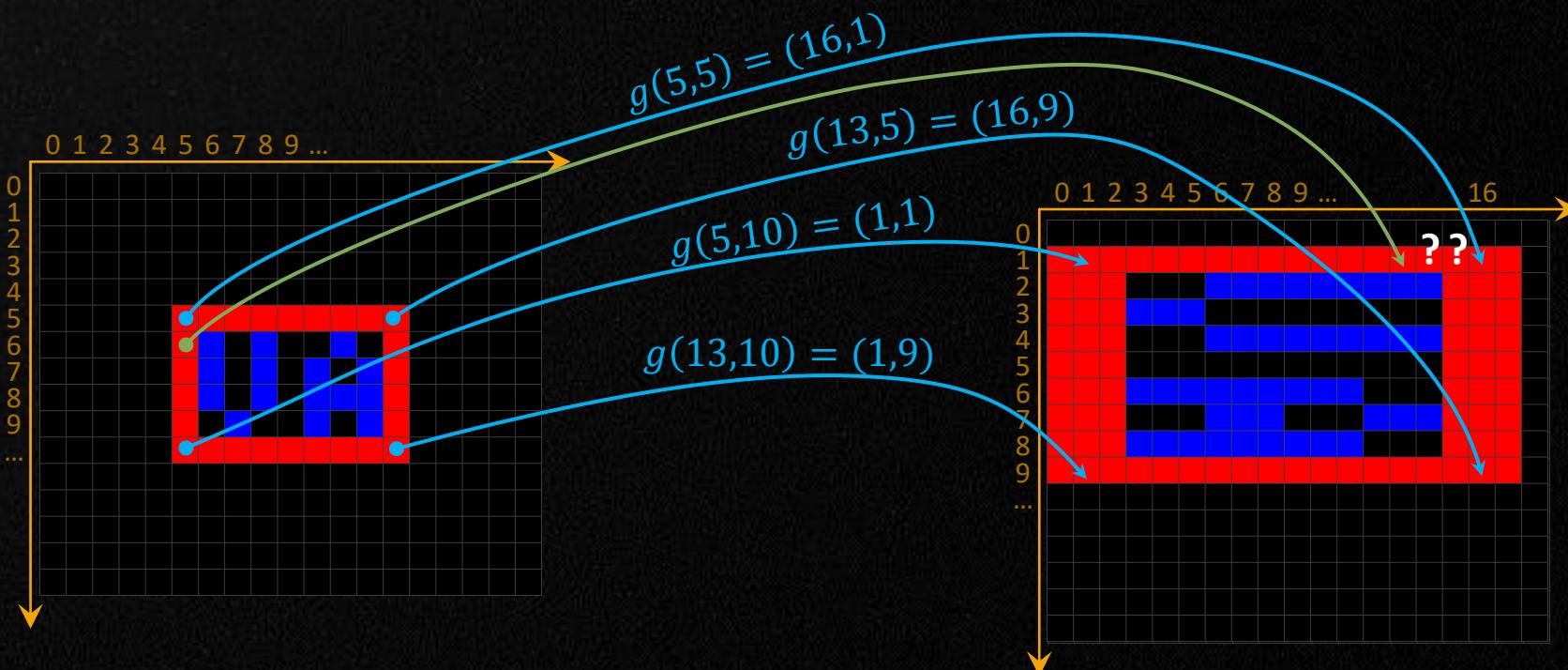
- $I: \mathbb{R} \times \mathbb{R} \times \mathbb{N} \rightarrow \mathbb{R}$

- ▶ Attenzione a:

- Origine in alto a sinistra
- Verso dell'asse y
- Coordinate negative



- ▶ Si tratta di trasformazioni che si applicano alle coordinate e non ai valori dei pixel
 - La griglia dei pixel viene deformata e mappata sull'immagine di destinazione
- ▶ Funzione di mapping: $g: \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} \times \mathbb{R}$



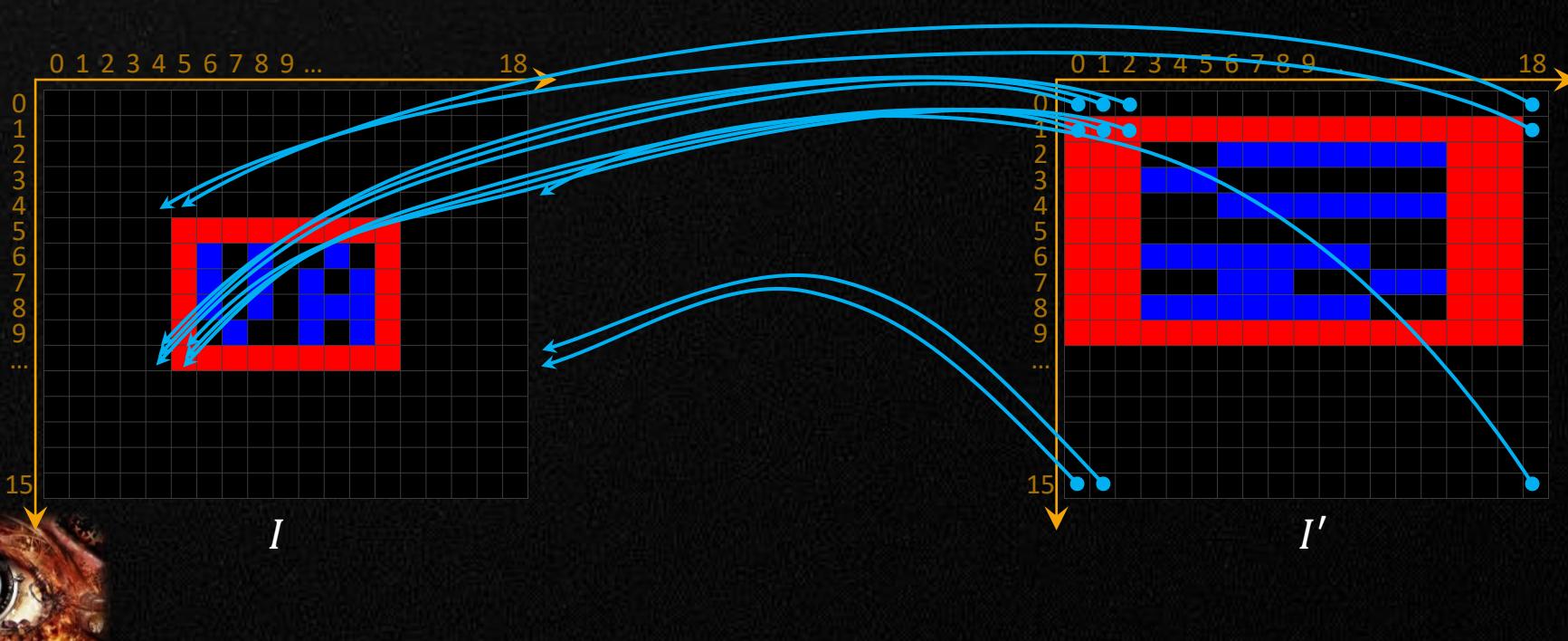
Trasformazioni geometriche: mapping inverso

► Mapping inverso: dalla destinazione alla sorgente: $f: \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} \times \mathbb{R}$, $f = g^{-1}$

- Per ogni pixel (x, y) dell'immagine destinazione I' , si calcolano le coordinate corrispondenti nell'immagine di partenza I e si copia il colore del pixel corrispondente

► Problemi da risolvere:

- Quale valore per i pixel non esistenti (fuori dall'immagine sorgente)?
- Quale valore del pixel quando $f(x, y)$ non restituisce coordinate intere?



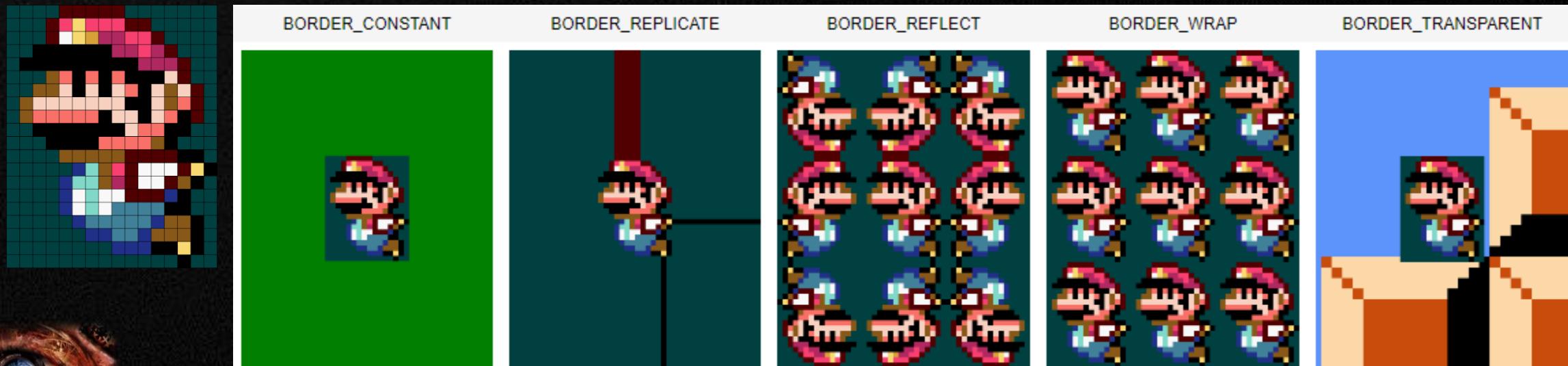
(x, y)	$f(x, y)$
(0, 0)	(4, 10.33)
(1, 0)	(4, 10)
(2, 0)	(4, 9.67)
...	...
(18, 0)	(4, 4.33)
(0, 1)	(5, 10.33)
(1, 1)	(5, 10)
(1, 2)	(5, 9.67)
...	...
(18, 1)	(5, 4.33)
...	...
(0, 15)	(19, 10.33)
(1, 15)	(19, 10)
...	...
(18, 15)	(19, 4.33)

Gestire le coordinate fuori dall'immagine

56

- ▶ Gli approcci più comuni per estrapolare il valore (con denominazione OpenCV):

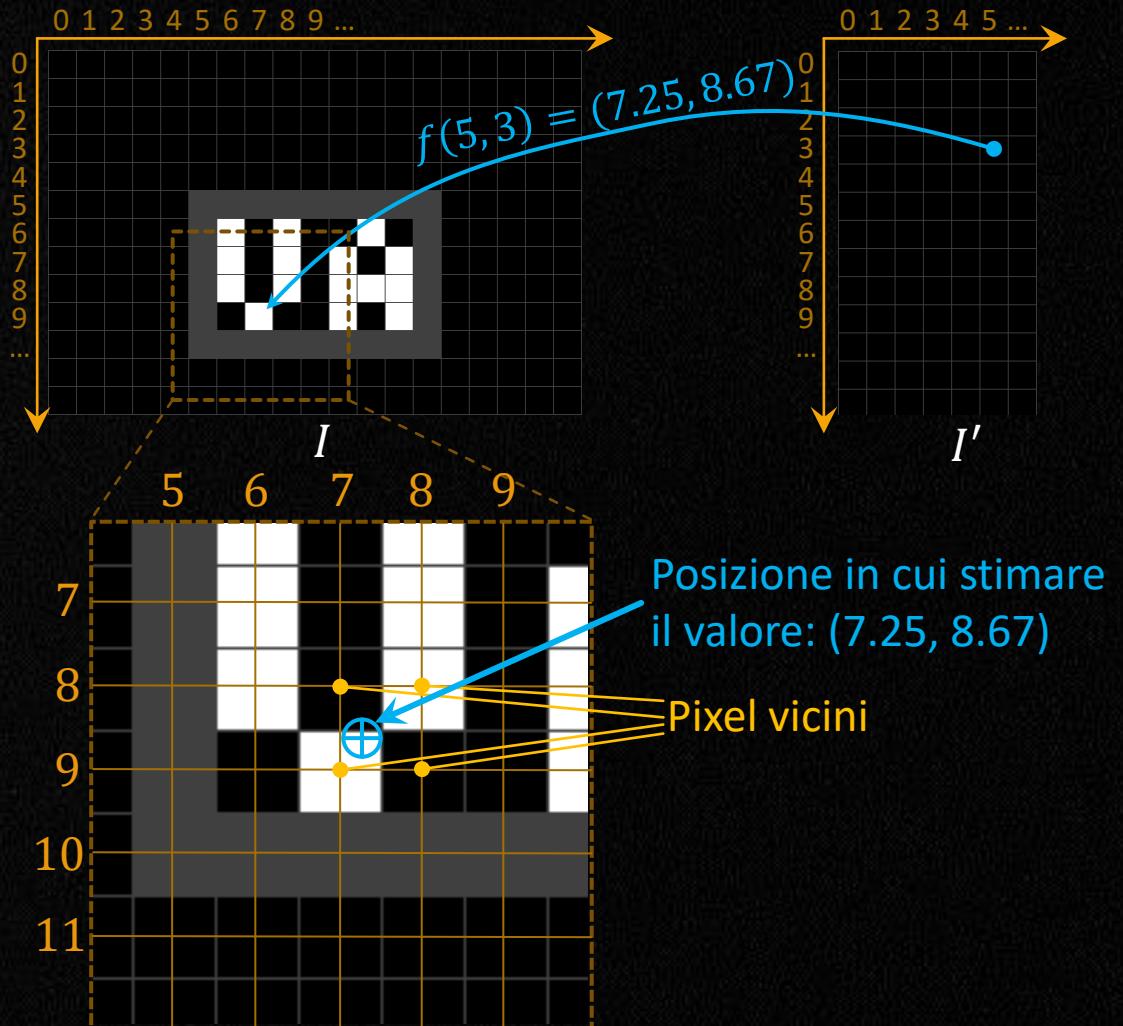
- Colore di background costante (cv.BORDER_CONSTANT)
- Copia del pixel più vicino (cv.BORDER_REPLICATE)
- Riflessione dei pixel dell'immagine (cv.BORDER_REFLECT)
- Replica dell'immagine (cv.BORDER_WRAP)
- Nessuna modifica: lascia il valore già presente nell'immagine destinazione (cv.BORDER_TRANSPARENT)



Stimare un valore per il pixel da coordinate non intere

57

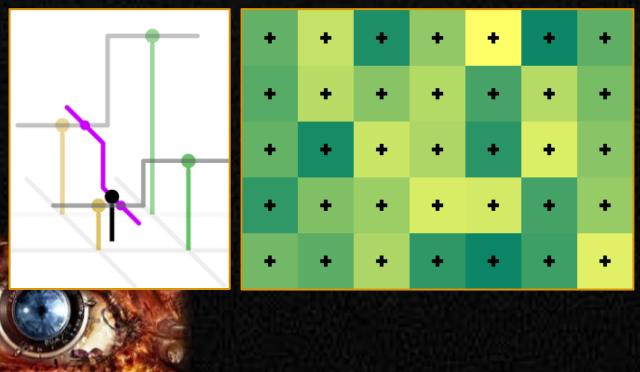
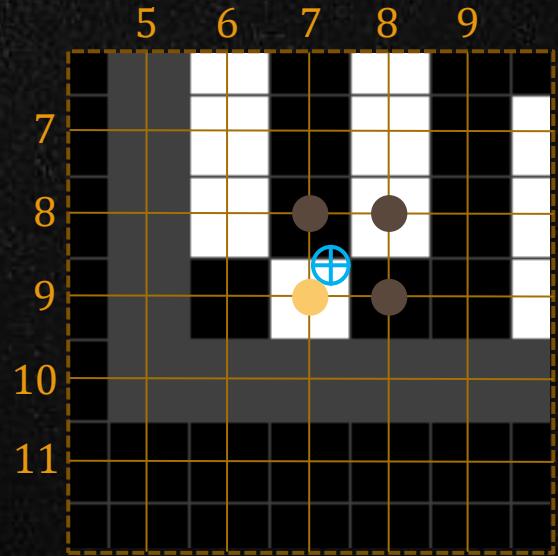
- ▶ In quasi tutte le trasformazioni geometriche, la maggior parte delle coordinate $f(x, y)$ dei valori da recuperare nell'immagine sorgente sono numeri con la virgola.
- ▶ Si procede stimando il valore a partire dai pixel (con coordinate intere) più vicini.
- ▶ Il metodo più semplice è scegliere il pixel più vicino (nearest-neighbor).
- ▶ Migliori risultati si ottengono con tecniche di **interpolazione** che adattano una funzione polinomiale ai pixel in un intorno ed eseguono la stima in base al valore di tale funzione alle coordinate $f(x, y)$.



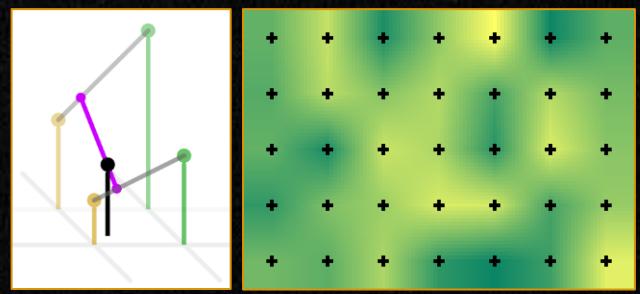
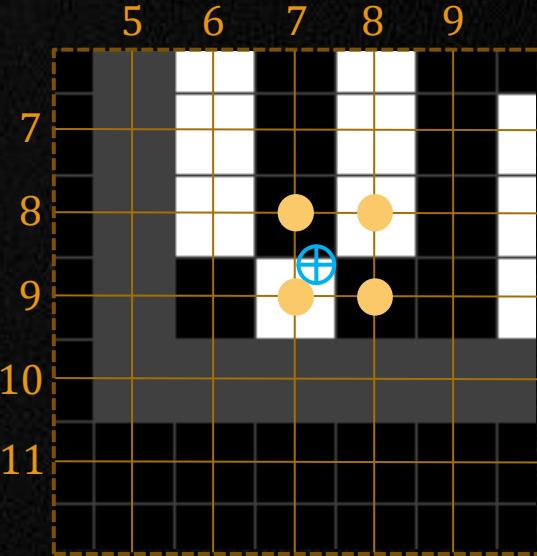
Interpolazione

58

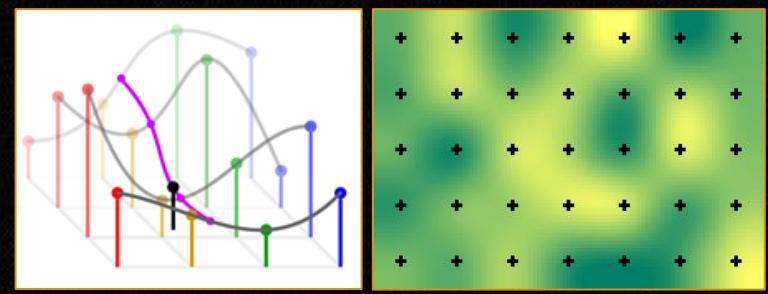
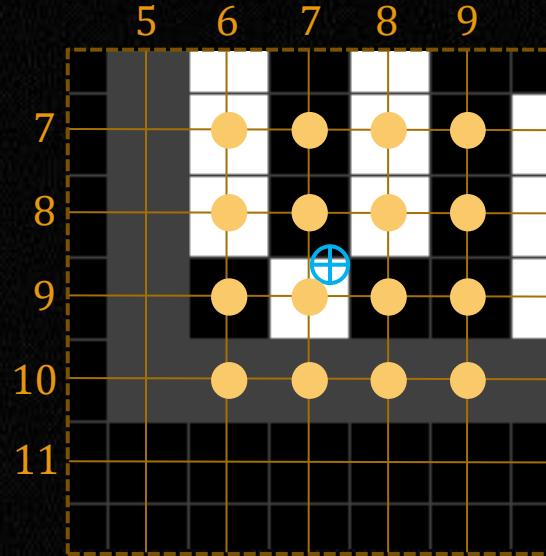
Nearest-neighbor



Bilineare



Bicubica



Ingrandimento di un'immagine

59

- ▶ Il ridimensionamento è il caso più semplice e probabilmente più utilizzato di trasformazione geometrica
 - Caso particolare di trasformazione affine (vedi lucidi seguenti)
- ▶ OpenCV dispone di una funzione specifica: `resize()`
- ▶ Interpolazione lineare:
 - Buon compromesso fra efficienza e qualità del risultato
- ▶ Interpolazione bicubica:
 - Di solito produce risultati migliori ma è molto più lenta

```
img = cv.imread('esempi/study.png')
h, w = img.shape[:2]
scale = 7.5
size = (int(w*scale),int(h*scale))
img_nn = cv.resize(img, size,
                   interpolation=cv.INTER_NEAREST)
img_bl = cv.resize(img, size,
                   interpolation=cv.INTER_LINEAR)
img_bc = cv.resize(img, size,
                   interpolation=cv.INTER_CUBIC)
```



Nearest-neighbor



0.176 ms

Bilineare



0.192 ms

Bicubica



1.08 ms

Confronto ingrandimento con interpolazione bilineare e bicubica

60



Bilineare



Confronto ingrandimento con interpolazione bilineare e bicubica

61

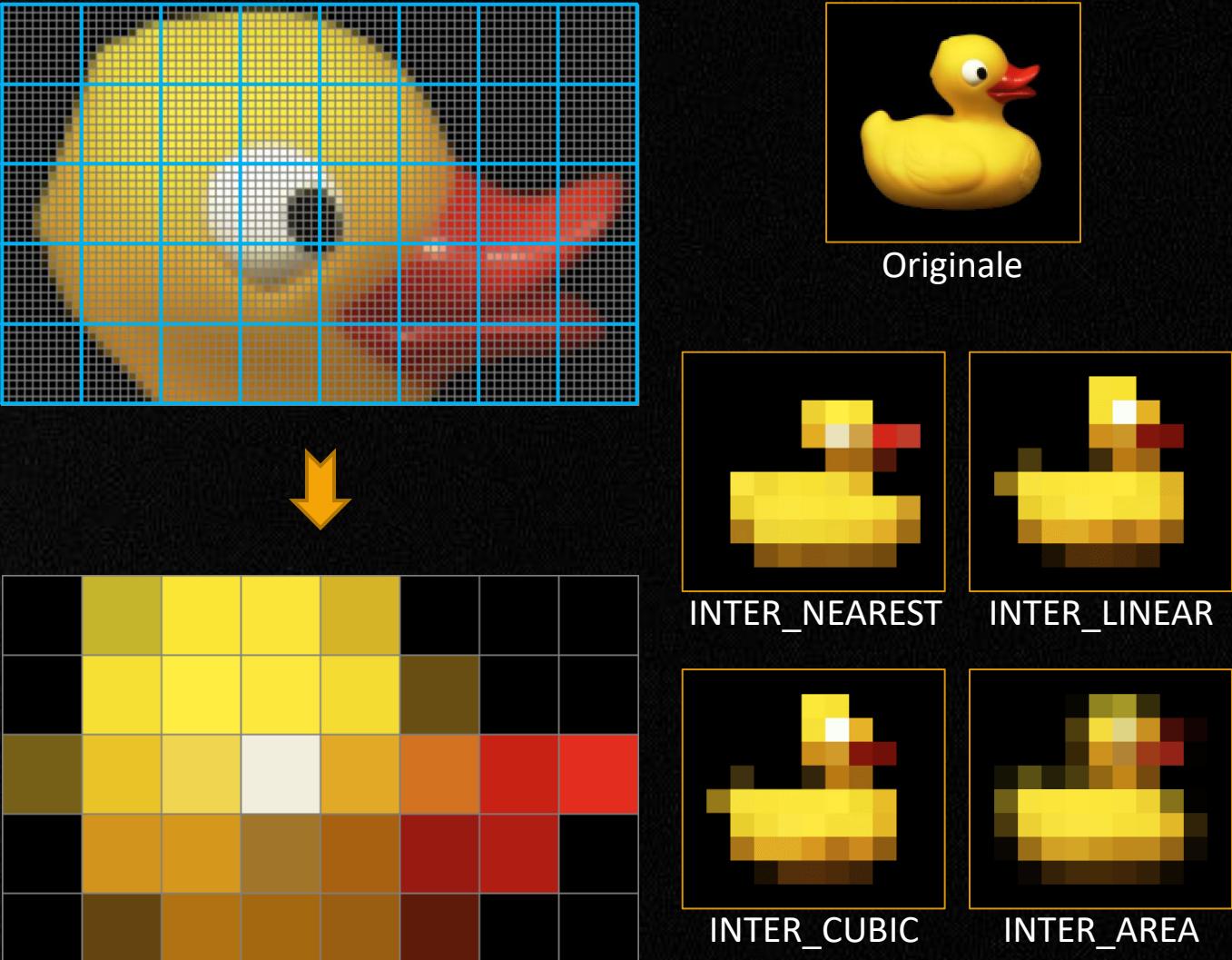


Bicubica



Rimpicciolire un'immagine

- In caso di grande riduzione delle dimensioni, l'interpolazione bilineare e bicubica possono considerare un numero di pixel troppo limitato rispetto alle informazioni presenti nell'immagine di partenza.
 - Nell'esempio a destra l'immagine è stata rimpicciolita 10 volte: ogni pixel dell'immagine destinazione corrisponde a 100 pixel della sorgente, mentre l'interpolazione bilineare ne considera solo 4 e la bicubica solo 16.
- In questi casi in OpenCV si può utilizzare **INTER_AREA** che considera tutti i pixel coinvolti.



► La funzione

$$g(x, y) = \begin{bmatrix} s_x \cos \theta & -s_y \sin \theta \\ s_x \sin \theta & s_y \cos \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

- ruota di un angolo θ rispetto all'origine
- applica un fattore di scala s_x lungo l'asse x e s_y lungo l'asse y
- esegue una traslazione t_x pixel lungo l'asse x e t_y lungo l'asse y

► Più in generale, una trasformazione affine 2D può essere rappresentata come moltiplicazione

per una matrice $\mathbf{A} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix}$ e somma (traslazione) di un vettore $\mathbf{t} = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$



- Si può riscrivere $\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$ come $\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & t_x \\ a_{10} & a_{11} & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$, aggiungendo una terza coordinata (sempre pari a 1) a tutti i vettori e aumentando la matrice con una nuova riga di zeri e una nuova colonna contenente il vettore traslazione.
- La traslazione nello spazio 2D può essere espressa come moltiplicazione in uno spazio 3D in cui la terza dimensione è pari a 1: le coordinate $(x, y, 1)$ sono dette *coordinate omogenee*.
- Vantaggio: si possono combinare più trasformazioni affini semplicemente moltiplicando fra loro le corrispondenti matrici.
 - Ad esempio se **A** e **B** sono le matrici di due trasformazioni affini T_A e T_B , **C** = **BA** è la matrice della trasformazione che corrisponde a eseguire T_A seguita da T_B .



- ▶ La funzione `warpAffine()` applica una trasformazione affine a un'immagine
- ▶ Richiede di specificare le prime due righe della matrice di trasformazione
$$\begin{bmatrix} a_{00} & a_{01} & t_x \\ a_{10} & a_{11} & t_y \\ 0 & 0 & 1 \end{bmatrix}$$
 - L'argomento `M` della funzione è quindi costituito da una matrice 2x3: $\mathbf{M} = \begin{bmatrix} a_{00} & a_{01} & t_x \\ a_{10} & a_{11} & t_y \end{bmatrix}$
- ▶ `warpAffine()` provvede ad invertire la trasformazione, ottenendo $f = g^{-1}$, che applica alle coordinate dell'immagine destinazione con il tipo di interpolazione e la gestione dei bordi specificati.
 - In alternativa è possibile passare direttamente la matrice della trasformazione f (già invertita), specificando il flag `WARP_INVERSE_MAP`.



Trasformazioni affini in OpenCV

```

import math
img = cv.imread('esempi/mario-c.png'); b = img.shape[0]//2;
bc = img[0,0].tolist()
img = cv.copyMakeBorder(img, b, b, b, b, cv.BORDER_CONSTANT, value = bc);
h, w, _ = img.shape

trasf = lambda M: cv.warpAffine(img,M,(w,h),None,cv.INTER_CUBIC,cv.BORDER_CONSTANT,bc)
# Traslazione di 9 pixel verso destra e 9 verso l'alto
M_T = np.array([ [1., 0., 9],
                  [0., 1., -9] ])
img_T = trasf(M_T)
# Riduzione di scala x e aumento y
M_S = np.array([ [0.7, 0.0, 0],
                  [0.0, 1.3, 0] ])
img_S = trasf(M_S)
# Rotazione di - π/8
θ = -math.pi/8; cos_θ, sin_θ = math.cos(θ), math.sin(θ)
M_R = np.array([ [cos_θ,-sin_θ, 0],
                  [sin_θ, cos_θ, 0] ])
img_R = trasf(M_R)

```



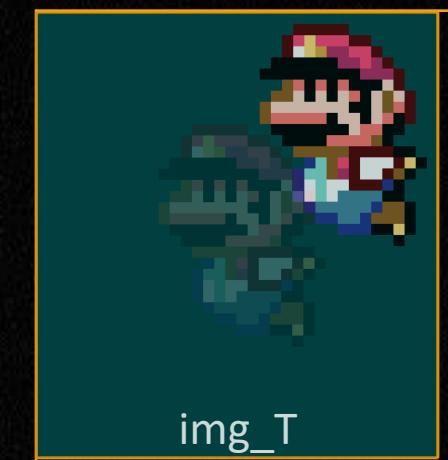
Trasformazioni affini in OpenCV

- Le trasformazioni avvengono rispetto all'origine dell'immagine (in alto a sinistra)



img

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

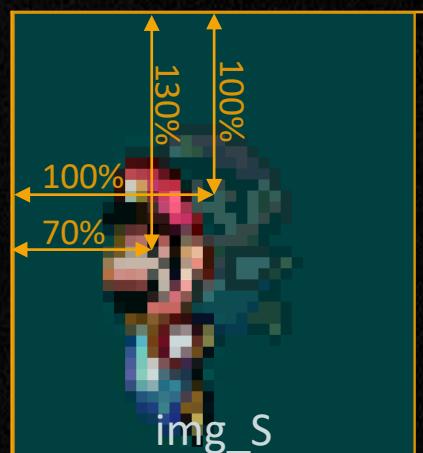


img_T

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 9 \\ 0 & 1 & -9 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

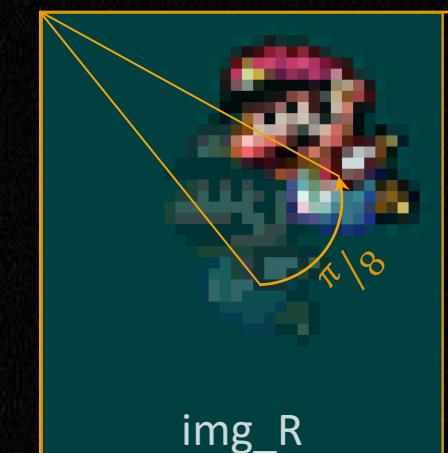
- Verso della rotazione: in un normale piano cartesiano un angolo positivo ruota in senso anti-orario, ma nell'immagine l'asse y è diretto verso il basso, quindi:

- angolo positivo: verso orario
- angolo negativo: verso anti-orario



img_S

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 0.7 & 0 & 0 \\ 0 & 1.3 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$



img_R

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(-\pi/8) & -\sin(-\pi/8) & 0 \\ \sin(-\pi/8) & \cos(-\pi/8) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$



Rotazione rispetto a un punto diverso dall'origine

68

- ▶ Traslare l'origine nel punto desiderato, eseguire la rotazione e infine la traslazione inversa.
- ▶ Il codice qui a fianco mostra come è possibile combinare le tre trasformazioni in un'unica matrice usando NumPy.
 - N.B. È disponibile la funzione OpenCV `getRotationMatrix2D()` che restituisce direttamente la matrice a partire dalle coordinate del punto, dall'angolo (verso antiorario) e da un eventuale fattore di scala.



```
# Rotazione rispetto al centro dell'immagine cx,cy
cx, cy = w//2, h//2
# Trasla il centro dell'immagine nell'origine
M_T1 = np.array([
    [1., 0., -cx],
    [0., 1., -cy],
    [0., 0., 1]
])
img_T1 = trasf(M_T1[:2])
# Rotazione di π/4
θ = math.pi/4
cos_θ, sin_θ = math.cos(θ), math.sin(θ)
M_R = np.array([
    [cos_θ, -sin_θ, 0],
    [sin_θ, cos_θ, 0],
    [0., 0., 1]
])
img_R = trasf(M_R[:2])
# Traslazione inversa
M_T2 = np.array([
    [1., 0., cx],
    [0., 1., cy],
    [0., 0., 1]
])
img_T2 = trasf(M_T2[:2])
# Compone le tre trasformazioni in una singola
M_Rc = M_T2 @ M_R @ M_T1
img_Rc = trasf(M_Rc[:2])
```

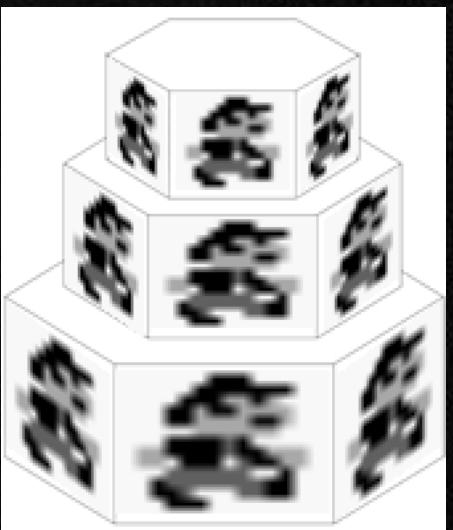
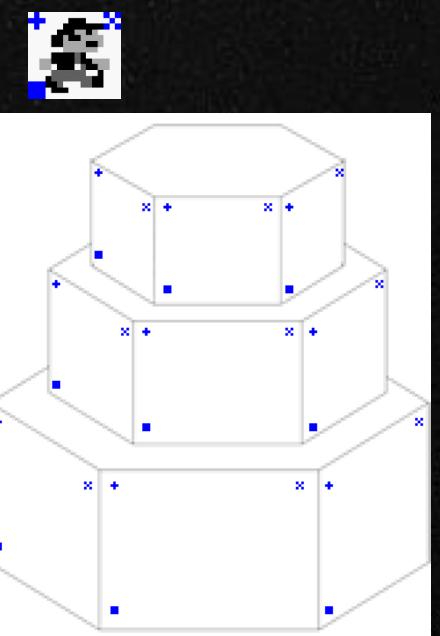
Trasformazione affine da coppie di punti corrispondenti

69

► Funzione OpenCV `getAffineTransform()`:

- Date tre coppie di punti corrispondenti, calcola la matrice di trasformazione affine

$$\mathbf{M} = \begin{bmatrix} a_{00} & a_{01} & t_x \\ a_{10} & a_{11} & t_y \end{bmatrix}$$
 che mappa ciascun punto nel suo corrispondente.



```
back = cv.imread('esempi/cake.png')
m = cv.imread('esempi/mario.png')
h, w = m.shape[:2]
m_pts = np.float32([[1,1],[w-2,1],[1,h-2]])
c_pts = np.float32([
    [[ 68, 35],[106, 35],[ 68, 66]],
    # ... 3 punti per ogni quadrato nell'immagine
    [[ 4,116],[ 38,140],[ 4,163]] ])
def drawPoints(img, lp):
    for pts in lp:
        for i in range(3):
            p = tuple(pts[i].round().astype(int))
            cv.drawMarker(img, p, (255,0,0), i, 3)
    return img
back_points = drawPoints(back.copy(), c_pts)
m_points = drawPoints(m.copy(), [m_pts])
for pts in c_pts:
    M = cv.getAffineTransform(m_pts,pts)
    cv.warpAffine(m, M, back.shape[1::-1], back,
                  cv.INTER_LINEAR, cv.BORDER_TRANSPARENT)
```

- ▶ A differenza delle trasformazioni affini, non preserva il parallelismo fra rette
- ▶ Definita da una matrice 3x3:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x_c/w \\ y_c/w \end{bmatrix}, \text{ con } w = \begin{cases} z_c & \text{se } z_c \neq 0 \\ \infty & \text{altrimenti} \end{cases} \text{ e } \begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} = \mathbf{M}_P \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}, \mathbf{M}_P \in \mathbb{R}^{3 \times 3}$$

- ▶ In OpenCV sono disponibili le funzioni:
 - `getPerspectiveTransform()` che calcola la matrice \mathbf{M}_P a partire da quattro coppie di punti corrispondenti
 - `warpPerspective()` che applica la trasformazione definita da una matrice \mathbf{M}_P a un'immagine, dopo aver invertito la trasformazione ottenendo $f = g^{-1}$, con il tipo di interpolazione e la gestione dei bordi specificati
 - Analogamente a `warpAffine()`, è possibile passare direttamente la matrice della trasformazione f (già invertita), specificando il flag `WARP_INVERSE_MAP`



Trasformazioni proiettive in OpenCV

71

```
img = cv.imread('esempi/building.jpg')
pts1 = np.float32([[195,16], [466,183], [485,269], [181,293]])
w, h = 400, 120 # Dimensioni dell'immagine destinazione
pts2 = np.float32([[0,0], [w-1,0], [w-1, h-1], [0, h-1]])
Mp = cv.getPerspectiveTransform(pts1, pts2)
res = cv.warpPerspective(img, Mp, (w,h), flags = cv.INTER_CUBIC)

m = cv.imread('esempi/sprite.png')
h, w = m.shape[:2] # Dimensioni di m
pts_m = np.float32([[0,0], [w-1,0], [w-1, h-1], [0, h-1]])
pts_b = np.float32([[243,111], [269,122], [273,220], [244,217]])
Mp = cv.getPerspectiveTransform(pts_m, pts_b)
res2 = cv.warpPerspective(m, Mp, img.shape[1::-1], img.copy(),
                           cv.INTER_LINEAR, cv.BORDER_TRANSPARENT)
```



► Immagini grayscale

- Coordinate cartesiane e matriciali, organizzazione dei pixel in memoria, immagini come array NumPy
- Istogramma: definizione, calcolo, analisi

► Immagini a colori

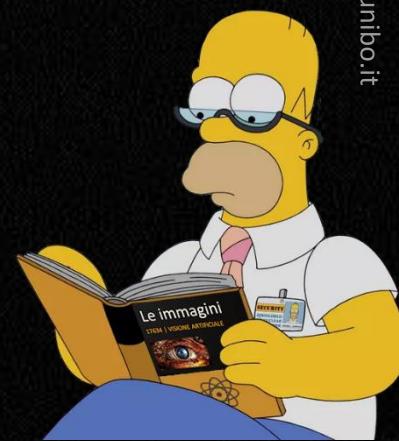
- Canali RGB, coordinate, organizzazione in memoria, immagini a colori come array NumPy
- HSV e HSL, conversioni in Python

► Operazioni sui pixel su singola immagine e su più immagini

- Variazione luminosità e contrasto, gamma, operazioni aritmetiche, binarizzazione, LUT
- Contrast stretching ed equalizzazione dell'istogramma

► Trasformazioni geometriche, mapping inverso, interpolazione

- Cambiamenti di scala, trasformazioni affini, trasformazioni proiettive



- ▶ Nel libro [Szeliski, Computer Vision: Algorithms and Applications, 2022]:
 - Sezione 2.3: The digital camera
 - Sezione 3.1: Point operators
 - Sezione 3.6: Geometric transformations
- ▶ Documentazione OpenCV del modulo «Image processing», in particolare:
 - Geometric Image Transformations
https://docs.opencv.org/master/da/d54/group__imgproc__transform.html
 - Miscellaneous Image Transformations
https://docs.opencv.org/master/d7/d1b/group__imgproc__misc.html
 - Color Space Conversions
https://docs.opencv.org/master/d8/d01/group__imgproc__color__conversions.html
 - Histograms
https://docs.opencv.org/master/d6/dc7/group__imgproc__hist.html

