A decorative graphic in the top-left corner consisting of a grid of squares in shades of purple, blue, and green, arranged in a stepped pattern.

Fondamenti di Elaborazione di Immagini

Operazioni sulle immagini

Raffaele Cappelli
raffaele.cappelli@unibo.it

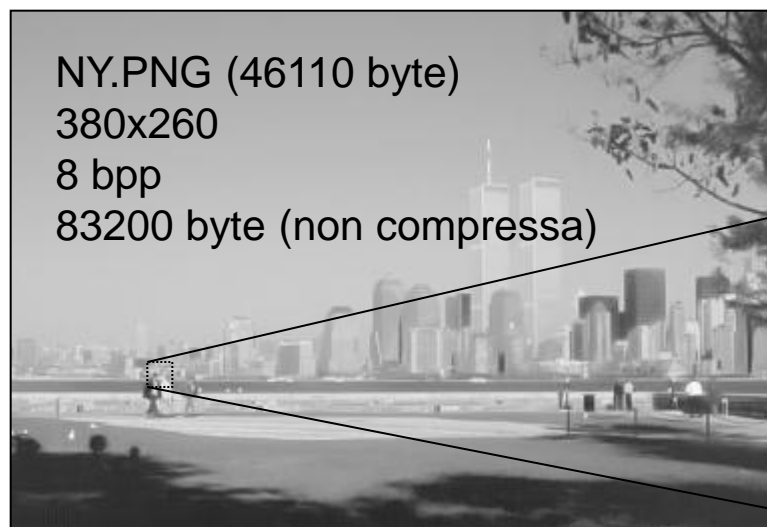
Contenuti

- Concetti di base
 - Le immagini digitali
 - Immagini a colori: modello RGB e HSL
- Operazioni sui pixel
 - Binarizzazione e operazioni aritmetiche su immagini
 - Operazioni sull'istogramma
- Operazioni locali
 - Filtri digitali e convoluzione
- Operazioni globali
 - Ruotare e ridimensionare un'immagine

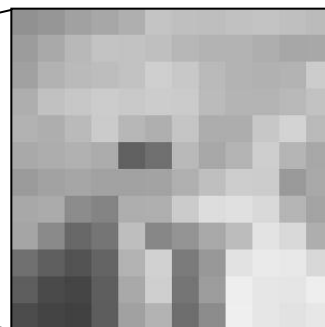
Immagini digitali

■ Immagine raster

- Una matrice di valori (pixel - picture element): ognuno rappresenta il dato (campionato e quantizzato) misurato da un sensore
- Alcune caratteristiche rilevanti:
 - 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): WxHxD (Depth = bit per pixel)



[12x12]



145	150	161	167	175	196	191	190	193	195	191	187
156	169	184	195	194	191	183	179	179	174	167	168
161	178	186	189	194	207	200	186	179	177	179	204
174	193	199	204	201	204	203	187	180	180	185	194
178	174	186	203	183	175	197	173	173	199	211	186
169	173	177	172	97	112	185	168	180	206	181	168
155	162	166	162	162	163	176	191	205	204	154	169
168	171	140	132	174	175	203	221	224	217	181	164
165	138	104	117	190	135	148	169	188	227	217	174
112	95	83	100	179	209	123	152	226	234	232	216
94	74	70	94	169	207	117	157	237	230	229	238
77	68	65	93	161	179	110	140	239	230	226	231

Immagini digitali (2)

■ Immagini vettoriali

- Costituite da un insieme di primitive geometriche (linee, archi, ...)
- Ampiamente utilizzate in CAD, GIS, Computer grafica
- Talvolta utilizzate anche in image analysis
- Visualizzate su schermo solo a seguito di conversione in immagini raster



Immagini grayscale in C#

- 8bpp: un byte per ogni pixel
- Immagine allocata come array bi-dimensionale
- Immagine allocata come array mono-dimensionale
 - Più efficiente
 - In genere è l'approccio preferibile

```
int w = 380;
int h = 260;

byte[,] img = new byte[h, w];

for (int y = 0; y < h; y++)
    for (int x = 0; x < w; x++)
        img[y, x] /= 2;
```

```
int w = 380;
int h = 260;
int n = w * h;
byte[] img = new byte[n];
for (int i = 0; i < n; i++)
    img[i] /= 2;
```

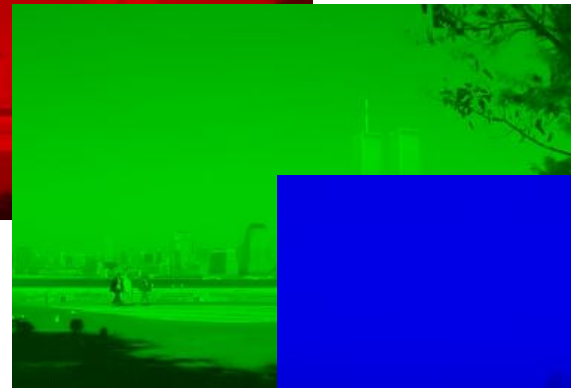
Immagini a colori

■ Palette

- I valori dei pixel sono indici all'interno di una tavolozza (palette) di colori
- Immagini solitamente a 16 o 256 colori

■ Formato RGB

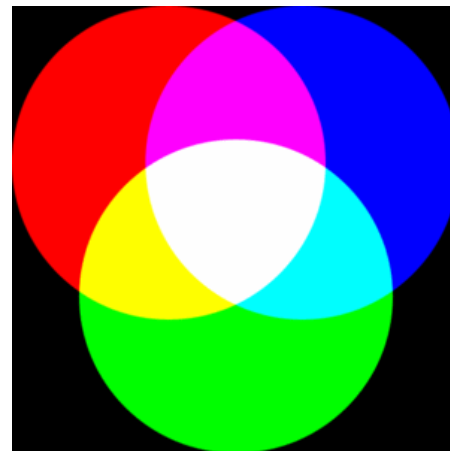
- Ogni pixel contiene un valore per ciascuna delle 3 componenti
 - generalmente 24 bpp



Il modello RGB

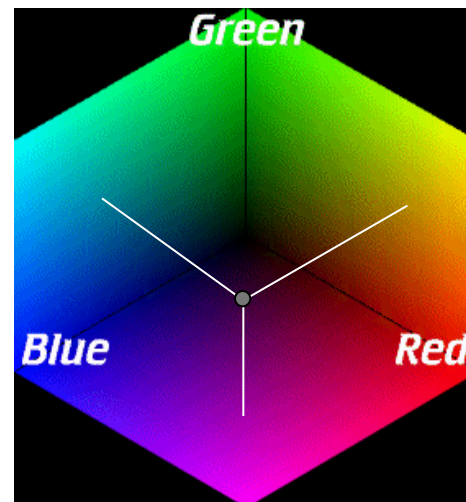
■ Modello additivo

- 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

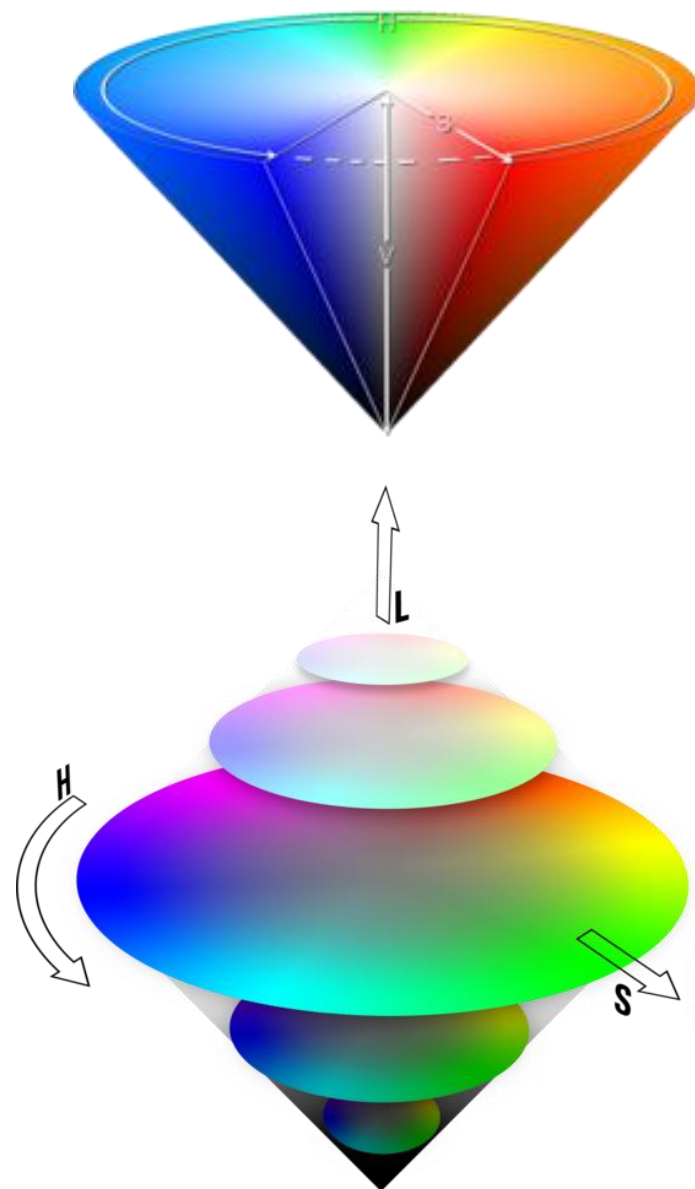


I modelli HS*

- Modelli basati sulle caratteristiche con cui un essere umano usualmente definisce un colore:
 - Tinta (Hue)
 - Saturazione
 - Luminosità
- Vantaggi
 - Possibilità di specificare i colori in modo intuitivo
 - Possono essere utilizzati più efficacemente per localizzazione e riconoscimento di pattern
- Due modelli principali:
 - HSV (o HSB)
 - HSL

I modelli HS* (2)

- HSV
 - Può essere rappresentato come un cono in cui l'asse verticale codifica V
- HSL
 - Può essere rappresentato come un doppio cono in cui l'asse verticale codifica L
- HSL meglio rappresenta i concetti di saturazione e luminosità
 - Variando S ci si muove sempre da un tono di grigio ($S=0$) al colore completamente saturo ($S=1$)
 - Variando L ci si muove sempre dal nero ($L=0$) al bianco ($L=1$)

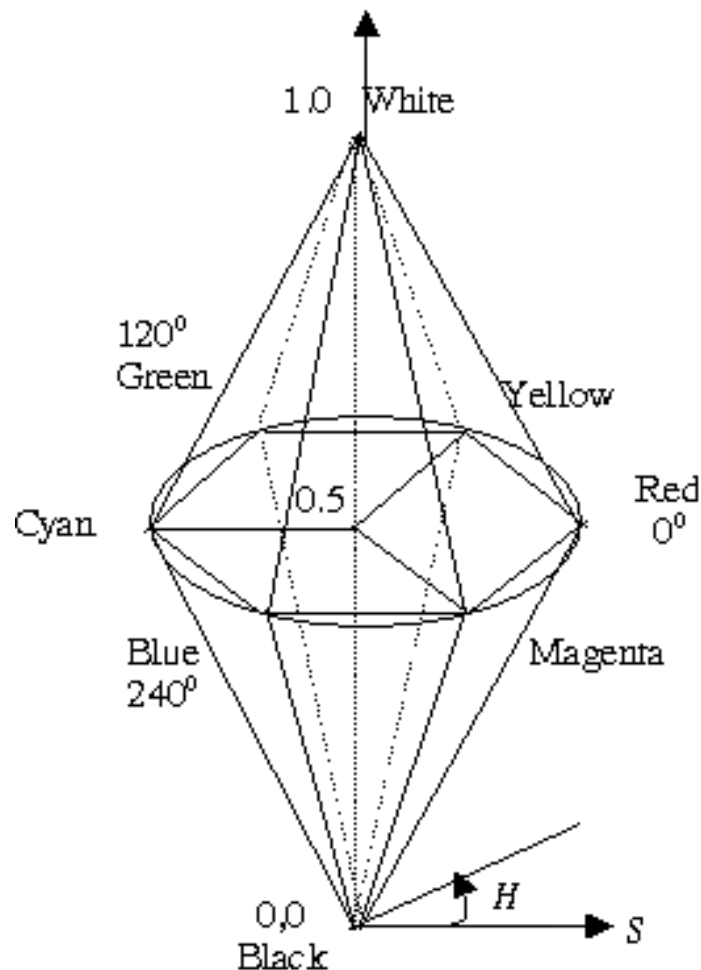


Il modello HSL

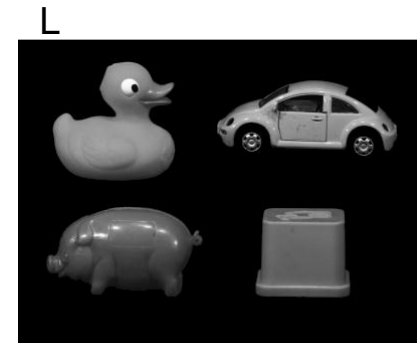
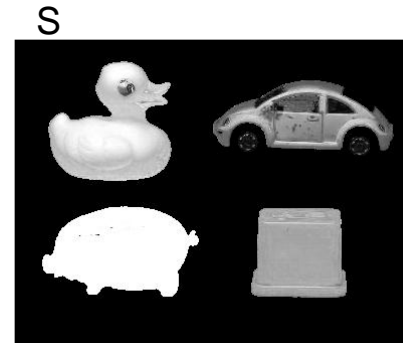
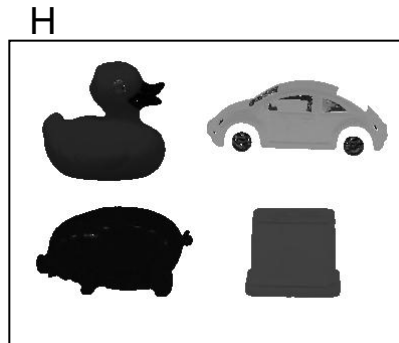
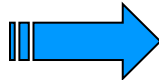
■ Intervallo valori

- Hue: $[0..2\pi]$
- Saturation: $[0..1]$
- Value: $[0..1]$
- Spesso si utilizzano 3 byte, discretizzando i valori nell'intervallo $[0..255]$

- Generalmente, anziché rappresentare lo spazio colore mediante due coni, si utilizzano due piramidi a base esagonale (il calcolo risulta più semplice ed efficiente)



HSL – Esempi



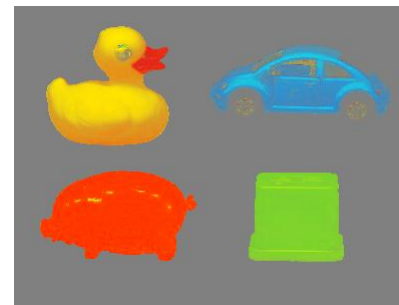
$H = \pi$



$H = \pi/2$



$L = 0.5$



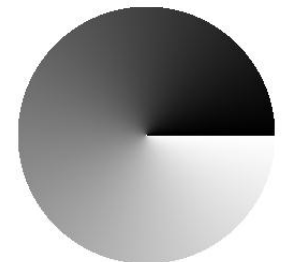
$S = 0.25$



$H += \pi/2$



$H += \pi$



Conversione RGB \rightarrow HSL

$$R, G, B \in [0,1]$$

$$C_{\max} = \max\{R, G, B\}$$

$$C_{\min} = \min\{R, G, B\}$$

$$H = \begin{cases} \text{undefined} & \text{if } C_{\max} = C_{\min} \\ \frac{\pi}{3} \left(\frac{G - B}{C_{\max} - C_{\min}} \right) & \text{if } C_{\max} = R \\ \frac{\pi}{3} \left(2 + \frac{B - R}{C_{\max} - C_{\min}} \right) & \text{if } C_{\max} = G \\ \frac{\pi}{3} \left(4 + \frac{R - G}{C_{\max} - C_{\min}} \right) & \text{if } C_{\max} = B \end{cases}$$

caso acromatico \swarrow

$$H \in [0, 2\pi]$$

$$S, L \in [0, 1]$$

$$\text{if } H < 0 \rightarrow H = H + 2\pi$$

$$L = \frac{1}{2} (C_{\max} + C_{\min})$$

$$S = \begin{cases} 0 & \text{if } C_{\max} = C_{\min} \\ \frac{C_{\max} - C_{\min}}{2L} & \text{if } 0 < L \leq \frac{1}{2} \\ \frac{C_{\max} - C_{\min}}{2 - 2L} & \text{if } L > \frac{1}{2} \end{cases}$$

Conversione HSL \rightarrow RGB

$$H \in [0, 2\pi]$$

$$S, L \in [0, 1]$$

if $S = 0$

$$\begin{aligned} \forall c \in \{R, G, B\} \\ c = L \end{aligned}$$

caso acromatico

$$R, G, B \in [0, 1]$$

otherwise

$$t_2 = \begin{cases} L \cdot (1 + S) & \text{if } L < \frac{1}{2} \\ L + S - L \cdot S & \text{otherwise} \end{cases}$$

$$t_1 = 2 \cdot L - t_2$$

$$H_1 = \frac{H}{2\pi}$$

$$t_R = H_1 + \frac{1}{3}; \text{ if } t_R > 1 \rightarrow t_R = t_R - 1$$

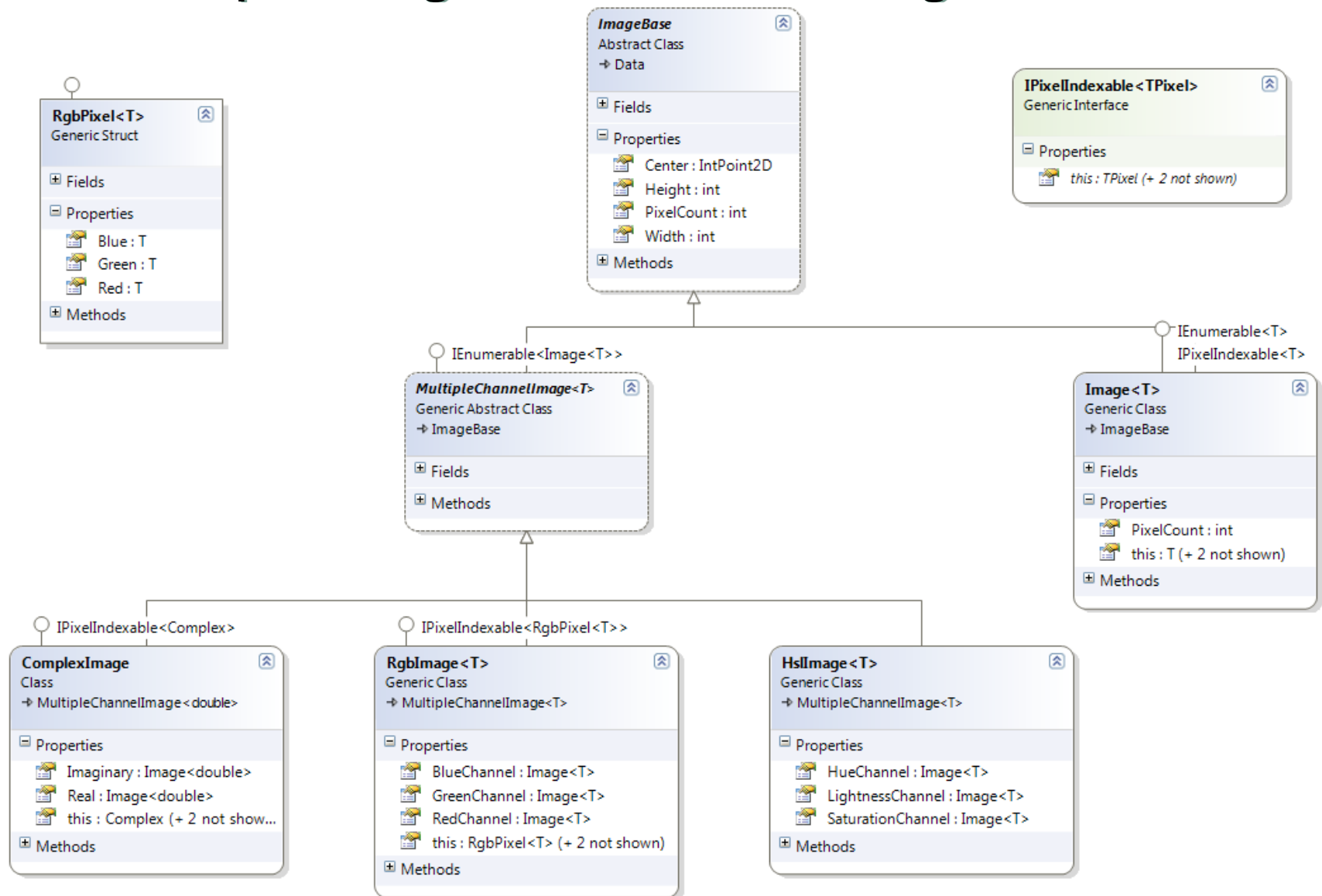
$$t_G = H_1$$

$$t_B = H_1 - \frac{1}{3}; \text{ if } t_R < 0 \rightarrow t_R = t_R + 1$$

$$\forall c \in \{R, G, B\}$$

$$c = \begin{cases} t_1 + 6 \cdot (t_2 - t_1) \cdot t_c & \text{if } t_c < \frac{1}{6} \\ t_2 & \text{if } \frac{1}{6} \leq t_c < \frac{1}{2} \\ t_1 + 6 \cdot (t_2 - t_1) \cdot \left(\frac{2}{3} - t_c\right) & \text{if } \frac{1}{2} \leq t_c < \frac{2}{3} \\ t_1 & \text{otherwise} \end{cases}$$

Classi per la gestione di immagini



Operazioni sui pixel

■ Su una singola immagine

- Ogni pixel dell'immagine di uscita è funzione solo del corrispondente pixel dell'immagine di input

$$\mathbf{I}'[i, j] = f(\mathbf{I}[i, j])$$

- Esempi principali:

- Variazione della luminosità
- Variazione di alcuni dei livelli di grigio: lista dei valori da modificare
- Conversione da livelli di grigio a (pseudo)colori
- Binarizzazione con soglia globale

■ Su più immagini

- Ogni pixel dell'immagine di uscita è funzione solo dei corrispondenti pixel delle immagini di input

$$\mathbf{I}[i, j] = f(\mathbf{I}_1[i, j], \mathbf{I}_2[i, j], \dots)$$

- Caso più comune:

- Operazioni aritmetiche fra due immagini: somma, sottrazione, AND, OR, XOR, ...

Operazioni sui pixel (2)

■ 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 LUT per eseguire l'operazione su tutti i pixel

```
[AlgorithmInfo("Look Up Table Transform", Category = "Basic operations")]
public class LookupTableTransform<TOutputPixel> : ImageOperation<Image<byte>, Image<TOutputPixel>>
    where TOutputPixel : struct, IEquatable<TOutputPixel>
{
    private const int lookUpTableLength = 256;
    private TOutputPixel[] lookupTable;

    [AlgorithmParameter]
    public TOutputPixel[] LookupTable
    {
        get
        { return lookupTable; }

        set
        {
            if (value.Length != lookUpTableLength)
                throw new ArgumentException("The lookup table must contain " +
                    lookUpTableLength + " elements", "value");
            lookupTable = value;
        }
    }
}
```

LookupTableTransform<TOutputPixel>
Generic Class
→ ImageOperation<Image<byte>, Image<TOutputPixel>>

Fields

- lookupTable : TOutputPixel[]
- lookUpTableLength : int

Properties

- LookupTable { get; set; } : TOutputPixel[]

Methods

- LookupTableTransform()
- LookupTableTransform(Image<byte> image, IEnumerable<TOutputPixel> table)
- LookupTableTransform(Image<byte> image, PixelMapping<byte, TOutputPixel> function)
- LookupTableTransform(Image<byte> image, TOutputPixel[] table)
- Run() : void

Creazione della Lookup Table

```
public LookupTableTransform(Image<byte> image, TOutputPixel[] table)
    : base(image)
{
    LookupTable = table;
}

public LookupTableTransform(Image<byte> image, PixelMapping<byte, TOutputPixel> function)
    : base(image)
{
    TOutputPixel[] table = new TOutputPixel[lookUpTableLength];
    for (int i = 0; i < lookUpTableLength; i++)
    { table[i] = function((byte)i); }
    LookupTable = table;
}

public override void Run()
{
    Result = new Image<TOutputPixel>(InputImage.Width, InputImage.Height);
    for (int i = 0; i < InputImage.PixelCount; i++)
    {
        Result[i] = lookupTable[InputImage[i]];
    }
}
```

Costruttore a cui si può passare l'array LUT già inizializzato

Costruttore a cui si può passare la funzione di mapping (delegate C#)

Applicazione della LUT

Operazioni sui pixel – Esempi

■ Variazione della luminosità

```
byte[] lut = new byte[256];  
for (int p = 0; p < 256; p++)  
    lut[p] = (p + var * 255 / 100).ClipToByte();  
Result = new LookupTableTransform<byte>(img, lut).Execute();
```

Implementazione 1

```
PixelMapping<byte, byte> f = delegate(byte p)  
{ return (p + var * 255 / 100).ClipToByte(); };  
Result = new LookupTableTransform<byte>(img, f).Execute();
```

Implementazione 2
(con delegate)

```
PixelMapping<byte, byte> f = p=>(p+var*255/100).ClipToByte();  
Result = new LookupTableTransform<byte>(img, f).Execute();
```

Implementazione 3
(con lambda expr.)

Var = -20



Var = 0



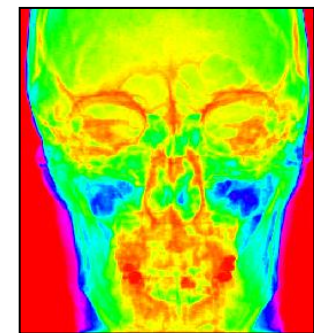
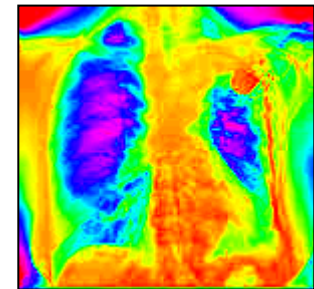
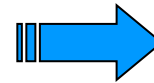
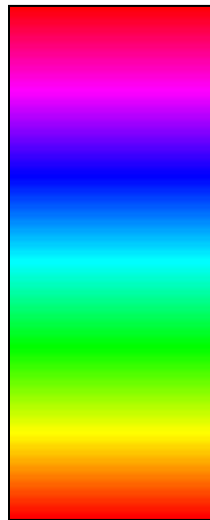
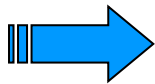
Var = +20



Operazioni sui pixel – Esempi (2)

- Esempio di Lookup Table da livelli di grigio a RGB

```
var lut = new RgbPixel<byte>[256] { ... };  
var res = new RgbImage<byte>(img.Width, img.Height);  
for (int i = 0; i < res.PixelCount; i++)  
    res[i] = lut[img[i]];
```



Operazioni sui pixel – Esempi (3)

■ Binarizzazione con soglia globale

```
Result = img.Clone();  
for (int i = 0; i < Result.PixelCount; i++)  
    Result[i] = (byte)(Result[i] < thr ? 0 : 255);
```

Implementazione 1
(senza LUT)

```
var op = new LookupTableTransform<byte>(img, p=>(byte)(p<thr ? 0 : 255));  
Result = op.Execute();
```

Implementazione 2
(con LUT e λ -expr.)

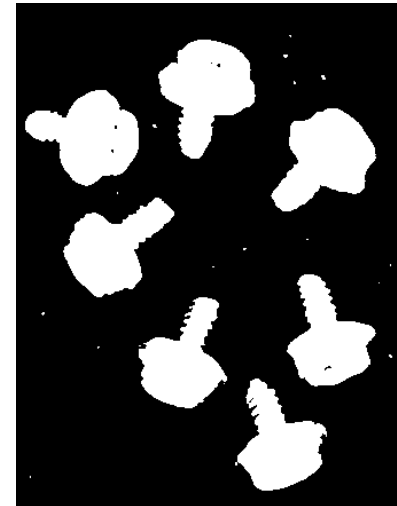
Originale



Thr = 110



Thr = 132

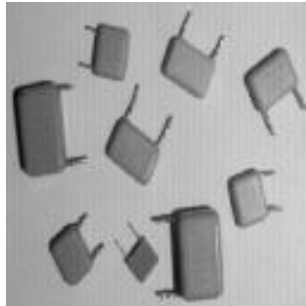


Operazioni aritmetiche fra immagini - Esempi

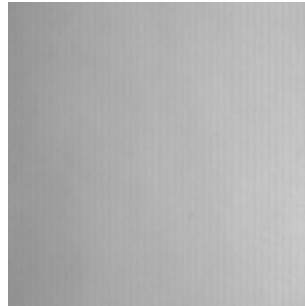
```
Result = new Image<byte>(img1.Width, img1.Height);
```

```
for (int i = 0; i < img1.PixelCount; i++)  
    Result[i] = (byte)Math.Abs(img1[i] - img2[i]);
```

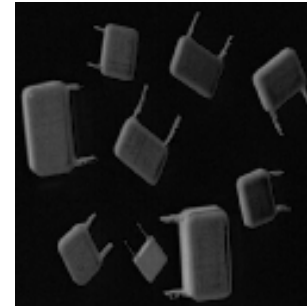
Img1



Img2



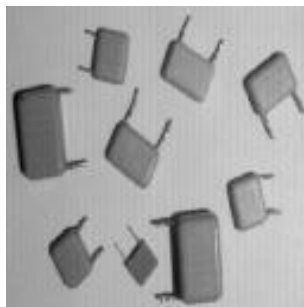
|Img1-Img2|



...

```
Result[i] = (byte)(img1[i] & img2[i]);
```

Img1



Img2



Img1 AND Img2



Istogramma di un'immagine grayscale

- 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 sono “condensati” in una zona, ciò significa che l'immagine ha un scarso contrasto
 - se nell'istogramma sono predominanti le basse intensità, l'immagine è molto scura e viceversa



Istogramma – Esempi



Immagine scura

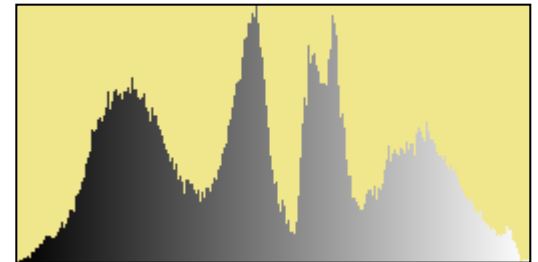
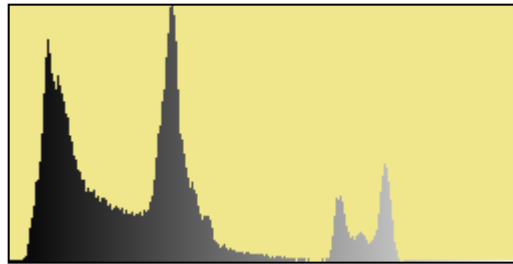
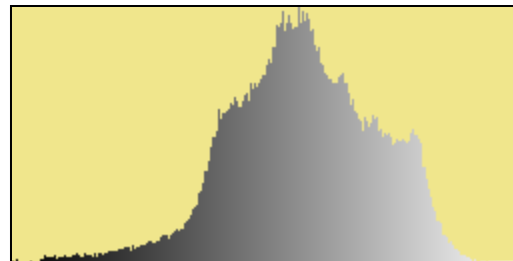


Immagine bilanciata

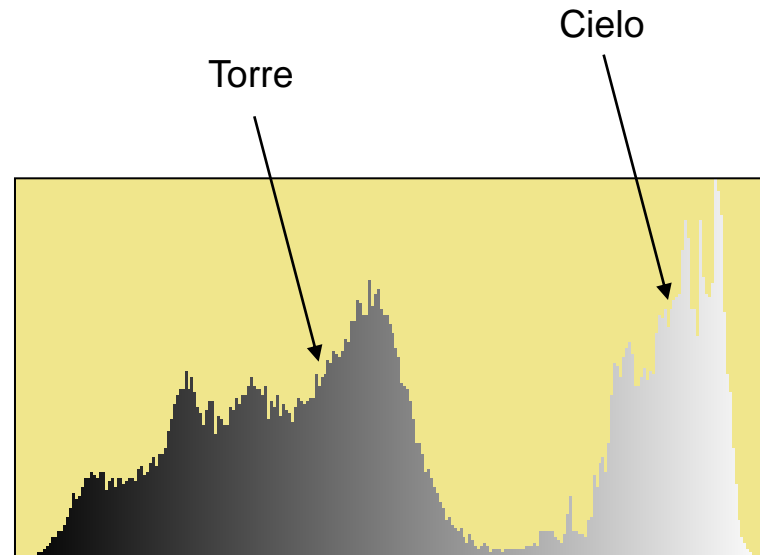
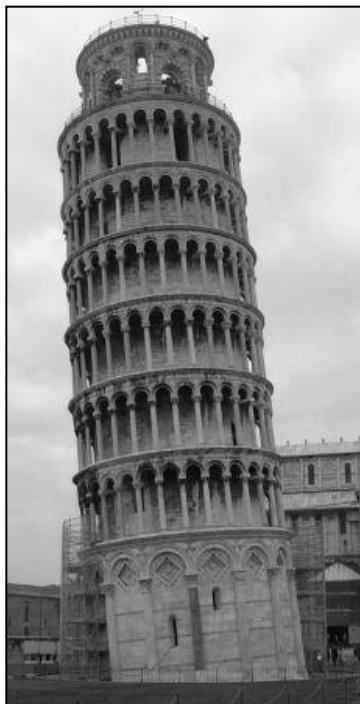


Immagine con poco contrasto



Analisi dell'istogramma

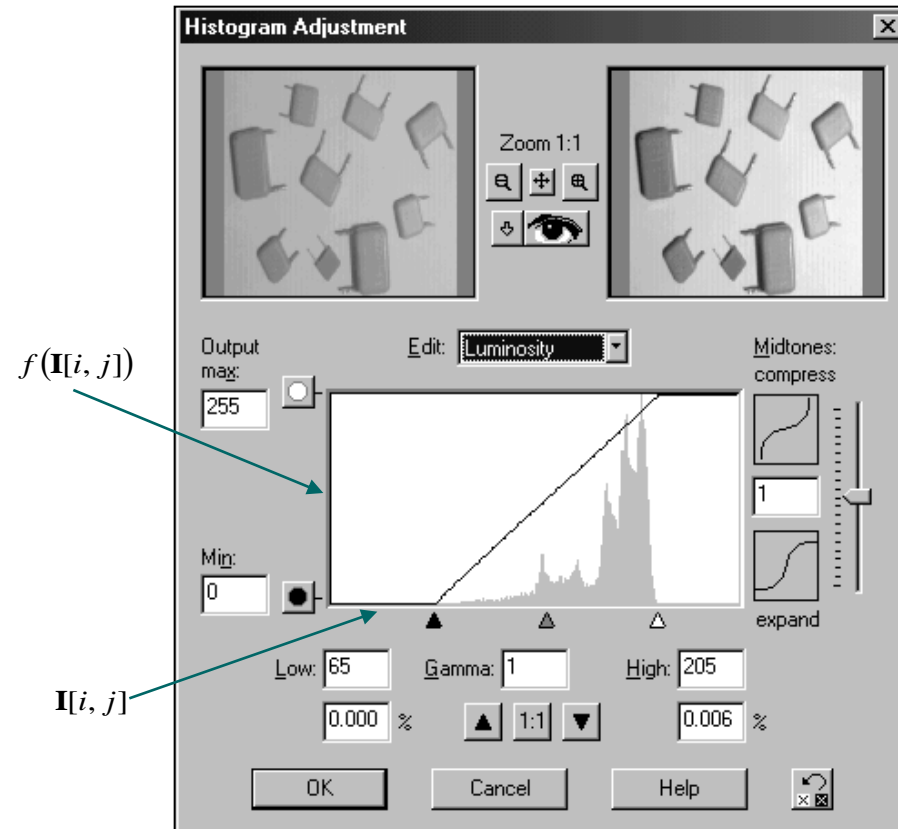
- Se i diversi oggetti in un'immagine hanno livelli di grigio differenti, l'istogramma può fornire un primo semplice meccanismo di classificazione
 - Esempio: un istogramma bimodale denota spesso la presenza di un oggetto abbastanza omogeneo su uno sfondo di luminosità pressoché costante.



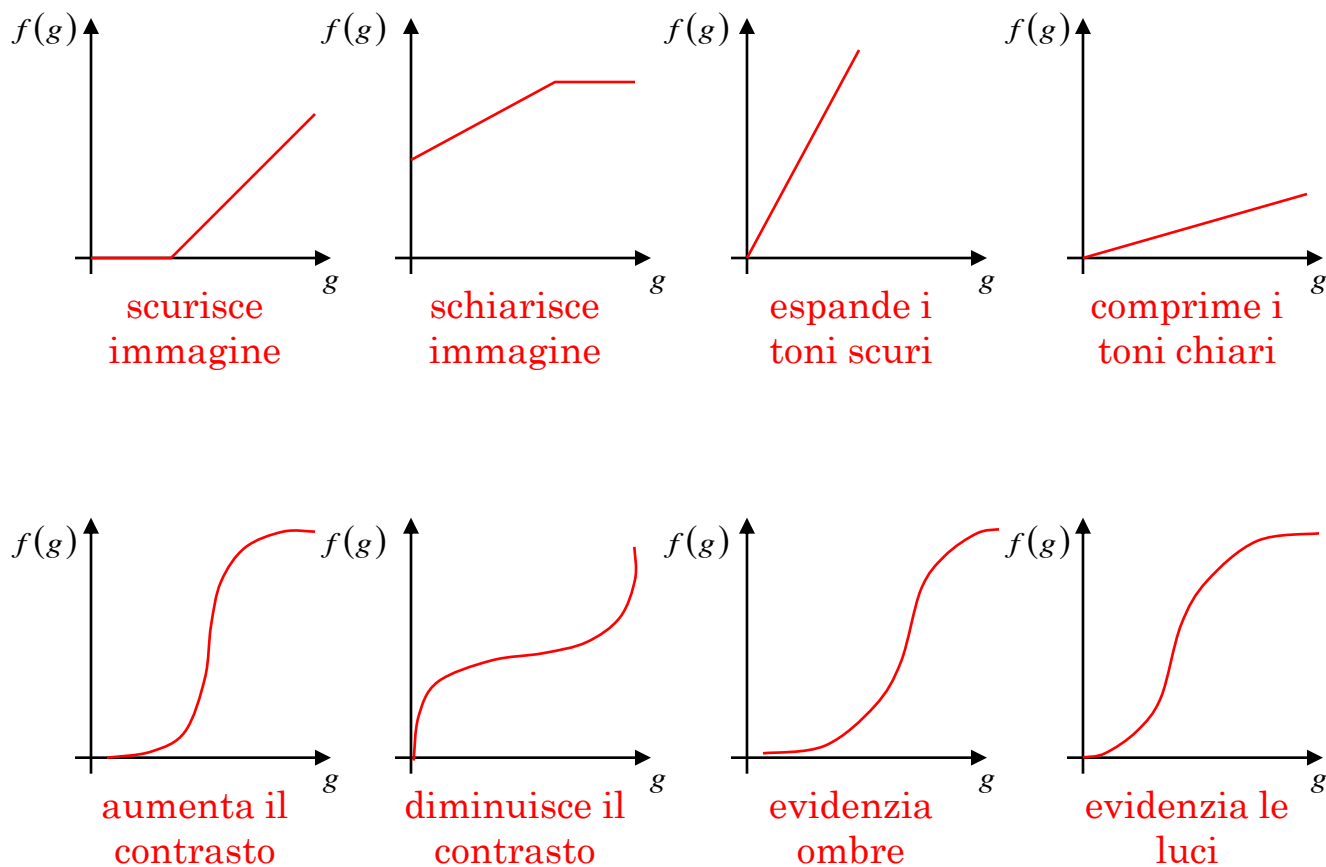
Istogramma e operazioni sui pixel

- L'istogramma fornisce informazioni utili a varie operazioni sui pixel
- Definendo opportune funzioni di mapping f è possibile:
 - aumentare il contrasto (espansione range dinamico)
 - scurire/schiarire l'immagine
 - evidenziare/nascondere dettagli
 - equalizzare l'istogramma
 - ridurlo a istogramma prefissato

$$\mathbf{I}'[i, j] = f(\mathbf{I}[i, j])$$



Istogramma e operazioni sui pixel – Esempi



Contrast stretching

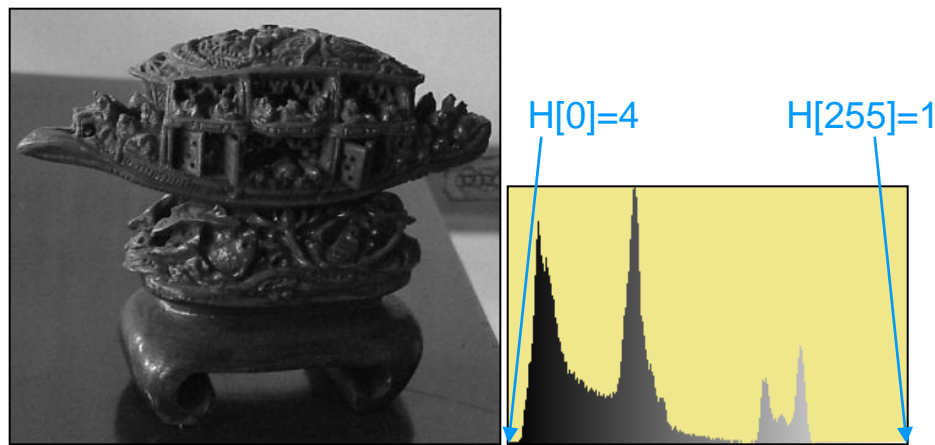
- Espansione dei livelli di grigio per aumentare il contrasto
- Si può ottenere con un semplice mapping lineare: $f(g) = 255 \cdot \frac{g - \min\{g_i\}}{\max\{g_i\} - \min\{g_i\}}$

```
byte min = InputImage[0], max = InputImage[0];
for (int i = 1; i < InputImage.PixelCount; i++)
{
    if (InputImage[i] < min) min = InputImage[i];
    else if (InputImage[i] > max) max = InputImage[i];
}

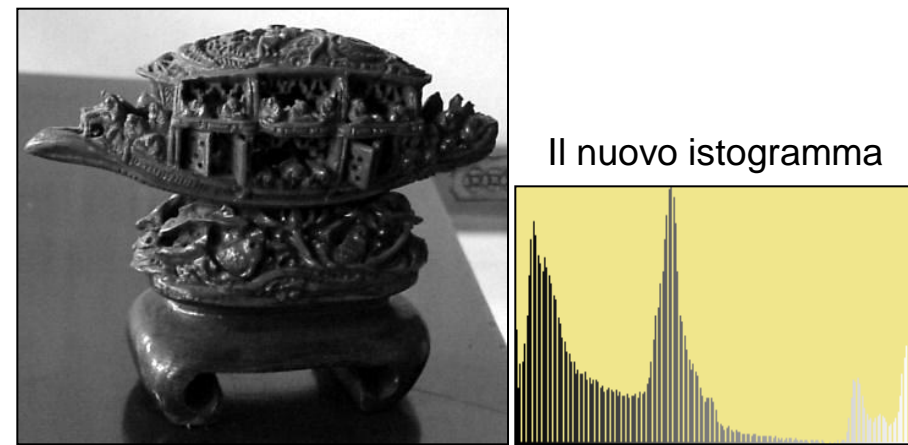
int diff = max - min;
if (diff > 0)
{
    var op = new LookupTableTransform<byte>(InputImage,
        p => (255 * (p - min) / diff).ClipToByte());
    Result = op.Execute();
}
else Result = InputImage.Clone();
```

Contrast stretching (2)

- L'implementazione nel lucido precedente non è molto robusta:
 - È sufficiente un pixel per cambiare la stima del minimo e il massimo
 - Pochi outliers (ad esempio dovuti a rumore nell'immagine) possono compromettere il risultato
- Una tecnica migliore consiste nel scartare una piccola percentuale dei pixel più chiari e più scuri prima di cercare il minimo e il massimo
 - A tale fine si può semplicemente utilizzare l'istogramma



min=0, max=255 :
Nessun contrast stretching!



Ignorando l'**1%** dei pixel ad ogni estremo, si ottiene:
min=11, max=190 : contrasto migliorato

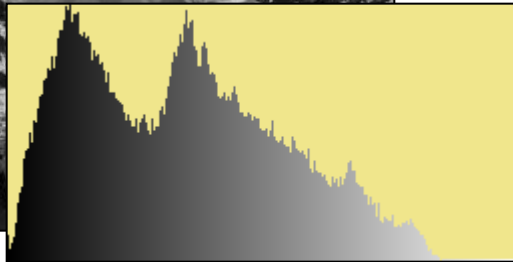
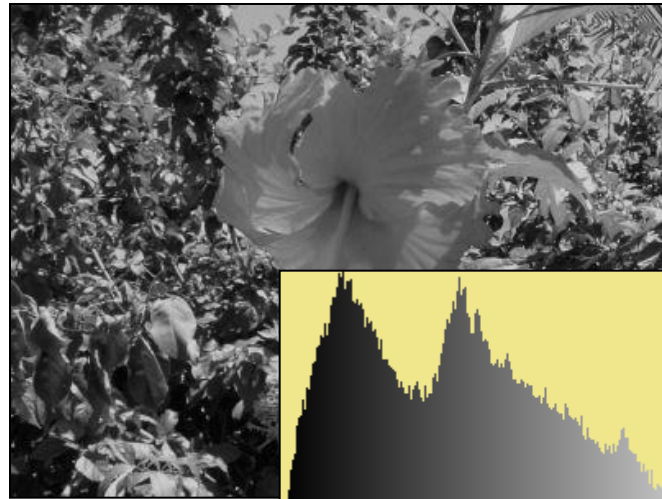
Equalizzazione dell'istogramma

- Una elaborazione molto importante
 - Spesso utilizzata per rendere **confrontabili** immagini catturate in **diverse condizioni di illuminazione**
- Obiettivo (ideale):
 - Produrre un'immagine con l'**istogramma uniformemente distribuito** su tutti i livelli di grigio, ossia distribuire equamente i pixel alle diverse intensità

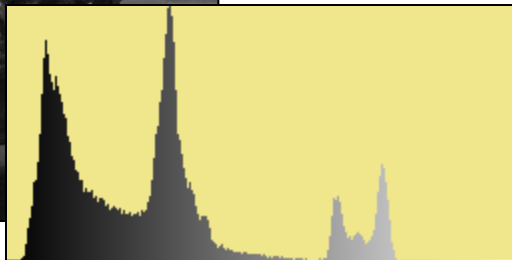
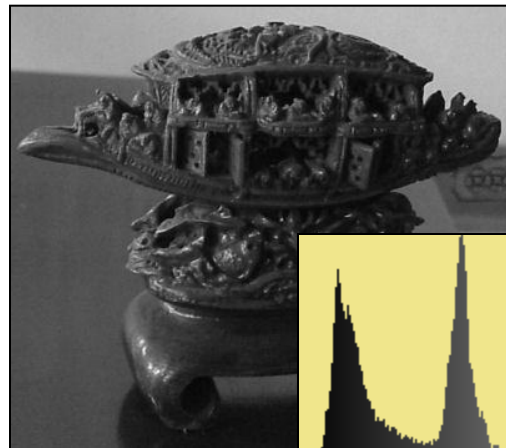
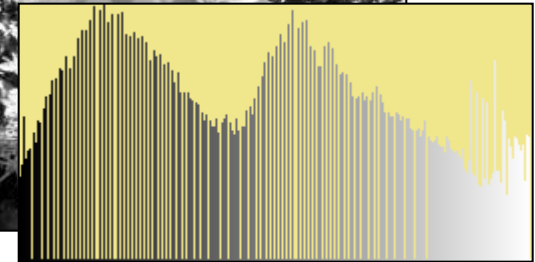
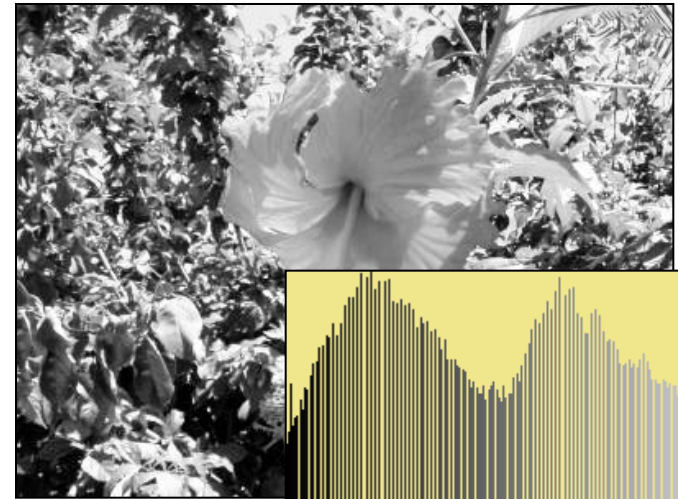
$$f(g) = \frac{255}{\#Pixel} \cdot \int_0^g H(w) \cdot dw \quad \Rightarrow \quad \text{Discretiz.} \quad f(g_i) = \frac{255}{\#Pixel} \cdot \sum_{j=0}^i H[g_j]$$

```
// Calcola l'istogramma
var hist = new HistogramBuilder(InputImage).Execute();
// Ricalcola ogni elemento dell'istogramma come somma dei precedenti
for (int i = 1; i < 256; i++)
    hist[i] += hist[i - 1];
// Definisce la funzione di mapping e applica la LUT
var op = new LookupTableTransform<byte>(InputImage,
    p => (byte)(255 * hist[p] / InputImage.PixelCount));
Result = op.Execute();
```

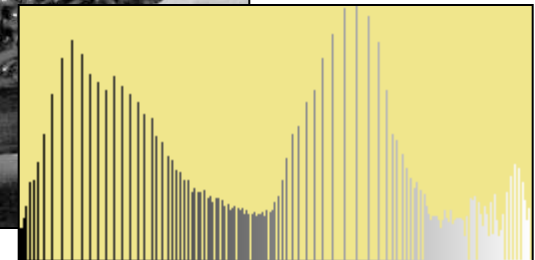
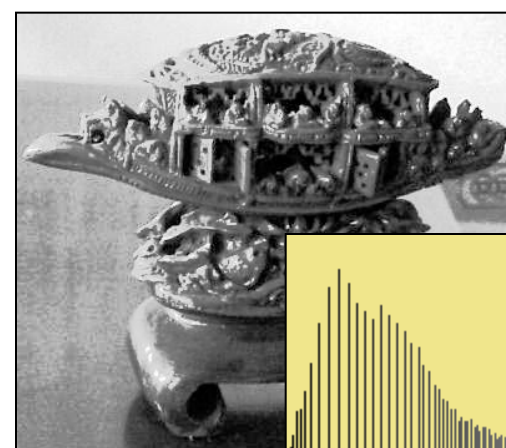
Equalizzazione – Esempi



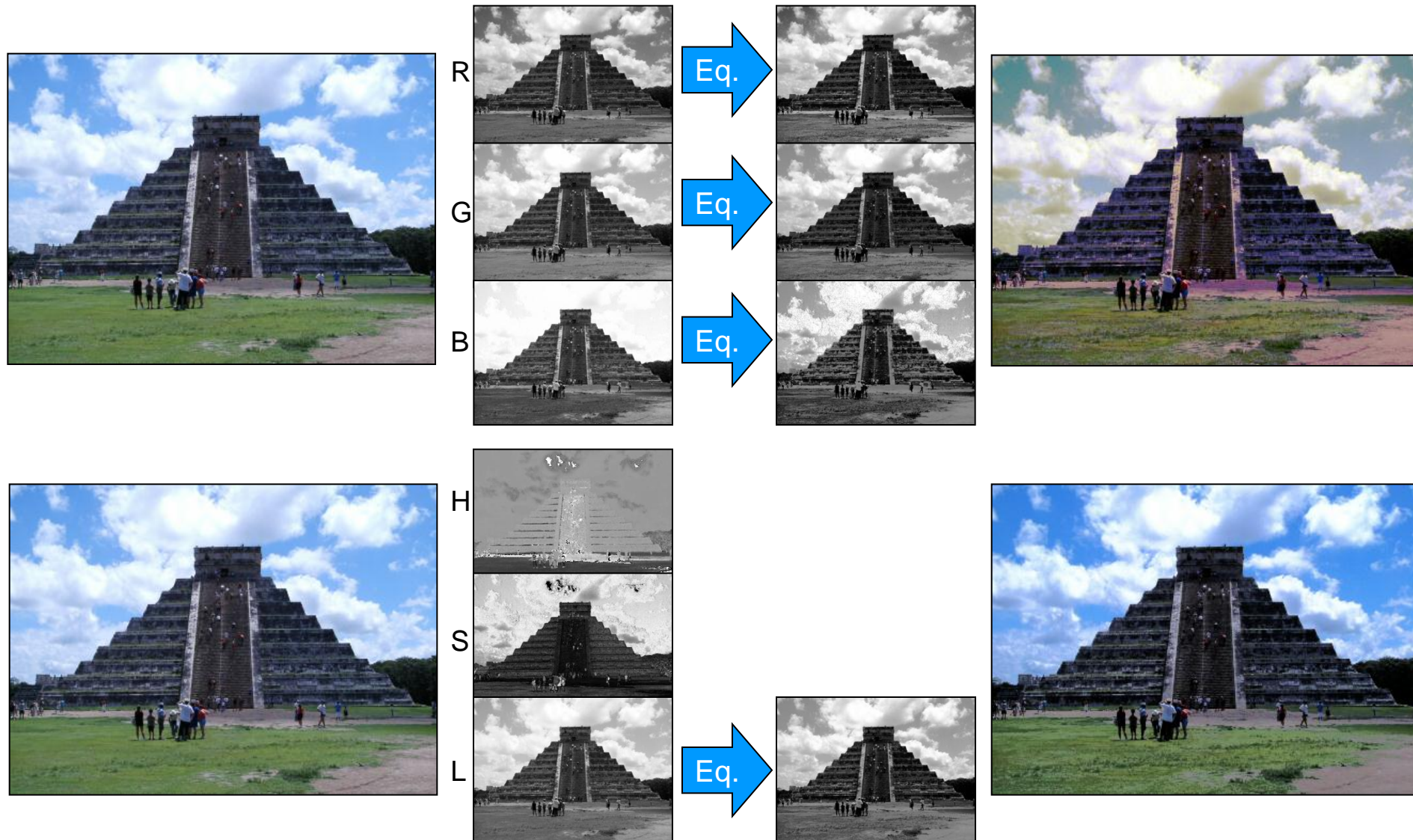
Equalizzazione



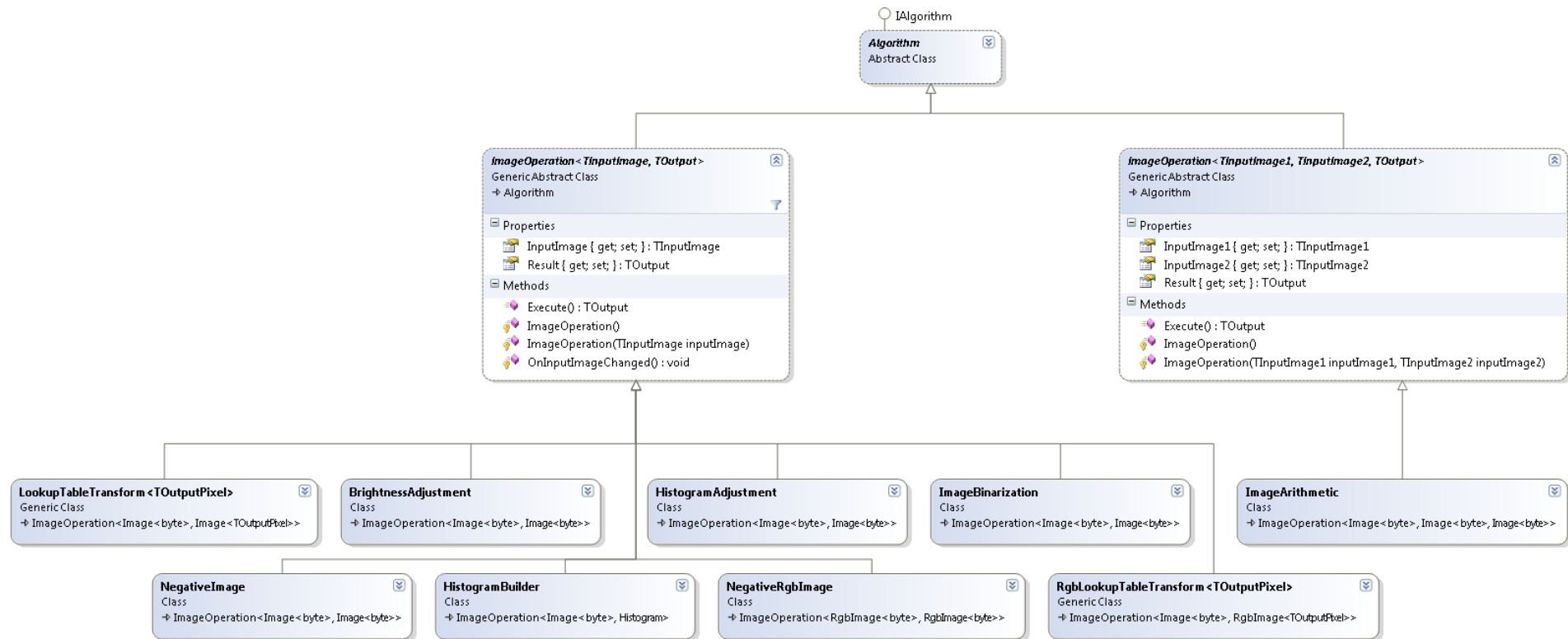
Equalizzazione



Equalizzazione di immagini a colori



ImageOperation e alcune classi derivate



Filtri digitali e convoluzione

■ Filtro digitale:

- Una maschera discreta di pesi che indicano come ogni elemento dell'immagine debba essere modificato sulla base del valore dei pixel vicini

■ Definizioni:

- Sia F un filtro definito su una griglia $m \times m$ (m dispari);
- L'applicazione di F a un'immagine I nel punto $[i, j]$ modifica il pixel $I[i, j]$ come segue:

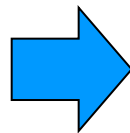
$$I'[i, j] = \sum_{y=1}^m \sum_{x=1}^m (I[i + \lceil \frac{m}{2} \rceil - y, j + \lceil \frac{m}{2} \rceil - x] \cdot F[y, x])$$

4	-2	1
-1	5	-3
-6	0	4

F

30	28	32
27	26	10
29	22	18

I



$$I'[i, j] =$$

$$18*4 - 22*2 + 29*1 +$$

$$- 10*1 + 26*5 - 27*3 +$$

$$- 32*6 + 28*0 + 30*4$$

- Tale operazione di media pesata locale è detta **convoluzione** nel punto $[i, j]$

Filtri digitali e convoluzione (2)

■ Osservazioni

- Il filtro è ribaltato sui due assi (si notino i due “-” nella formula)
- Spesso nella pratica la convoluzione viene **erroneamente** calcolata senza effettuare tale ribaltamento, utilizzando la formula della *correlazione* (con il “+”).
 - Ciò è corretto solo per filtri simmetrici rispetto all’origine.
 - Solo utilizzando il segno “-” la convoluzione è: commutativa, associativa e distributiva.
- Se il risultato della convoluzione deve essere un valore di intensità (ad es. compreso tra 0 e 255), normalmente si esegue una **normalizzazione**, dividendo il valore risultante per la somma dei pesi del filtro.
 - Altrimenti, in generale, il risultato della convoluzione è un numero con segno

Filtri digitali e convoluzione (3)

■ Complessità computazionale

- Piuttosto elevata: data un'immagine di $n \times n$ pixel e un filtro di $m \times m$ elementi, la convoluzione richiede $m^2 n^2$ moltiplicazioni e altrettante somme.

■ Aritmetica intera

- I calcolatori odierni, benché molto più efficienti che in passato nelle operazioni in virgola mobile, eseguono comunque più velocemente le operazioni in aritmetica intera.
- Pertanto il calcolo della convoluzione dovrebbe sempre essere eseguito in aritmetica intera (eventualmente discretizzando i pesi del filtro dopo averli moltiplicati per una costante opportuna).

Filtri digitali e convoluzione – Esempi

- Filtri di **Smoothing** (regolarizzazione):
 - Producono una sfocatura più o meno evidente, in grado di nascondere piccole imperfezioni e brusche variazioni di luminosità
 - Possono essere utili come passo iniziale prima di ulteriori elaborazioni



$$\frac{1}{9} \cdot \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$



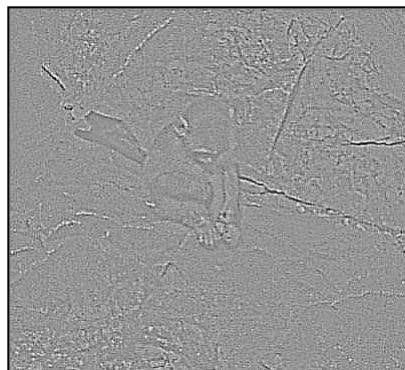
$$\frac{1}{8} \cdot \begin{bmatrix} 0 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$



$$\frac{1}{25} \cdot \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Filtri digitali e convoluzione – Esempi (2)

- Filtri di **Sharpening** (affilamento):
 - Evidenziano dettagli fini dell'immagine e le brusche variazioni di luminosità (contorni)
 - L'effetto desiderato si ottiene sommando la risposta del filtro all'immagine originale
 - Possono avere effetti indesiderati in presenza di rumore nell'immagine

Immagine originale I Risultato convoluzione R  $I' = I + k \cdot R$

$$F = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Controlla la forza
dell'effetto.
 $k=0.25$ nell'esempio

Convoluzione: aspetti implementativi

■ Calcolo del risultato

- In generale è necessario lavorare su un buffer di appoggio
- L'approccio più semplice consiste nel produrre il risultato in una nuova immagine

■ Problematiche relative ai bordi

- L'intorno di un pixel non è sempre disponibile: i pixel di bordo non possono produrre risultati corretti
- Diverse possibilità:
 - Ignorare $m/2$ pixel di bordo su ogni lato (ad esempio ponendoli a 0 nell'immagine risultante)
 - Supporre che i pixel non disponibili abbiano intensità zero
 - Prolungare i pixel di bordo supponendo intensità costante nei pixel non disponibili
 - ...

Convoluzione: implementazione di base

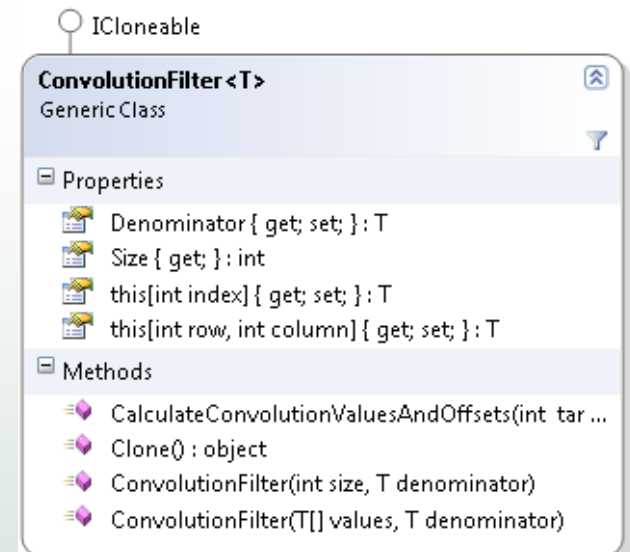
```

Result = new Image<int>(InputImage.Width, InputImage.Height);
int w = InputImage.Width;
int h = InputImage.Height;
int m = Filter.Size;
int m2 = m / 2;
int i1 = m2;
int i2 = h - m2 - 1;
int j1 = m2;
int j2 = w - m2 - 1;

// I bordi in Result restano a 0
for (int i = i1; i <= i2; i++)
    for (int j = j1; j <= j2; j++)
    {
        int val = 0;
        for (int y = 0; y < m; y++)
            for (int x = 0; x < m; x++)
                val += InputImage[i + m2 - y, j + m2 - x] * Filter[y, x];

        Result[i, j] = val / Filter.Denominator;
    }

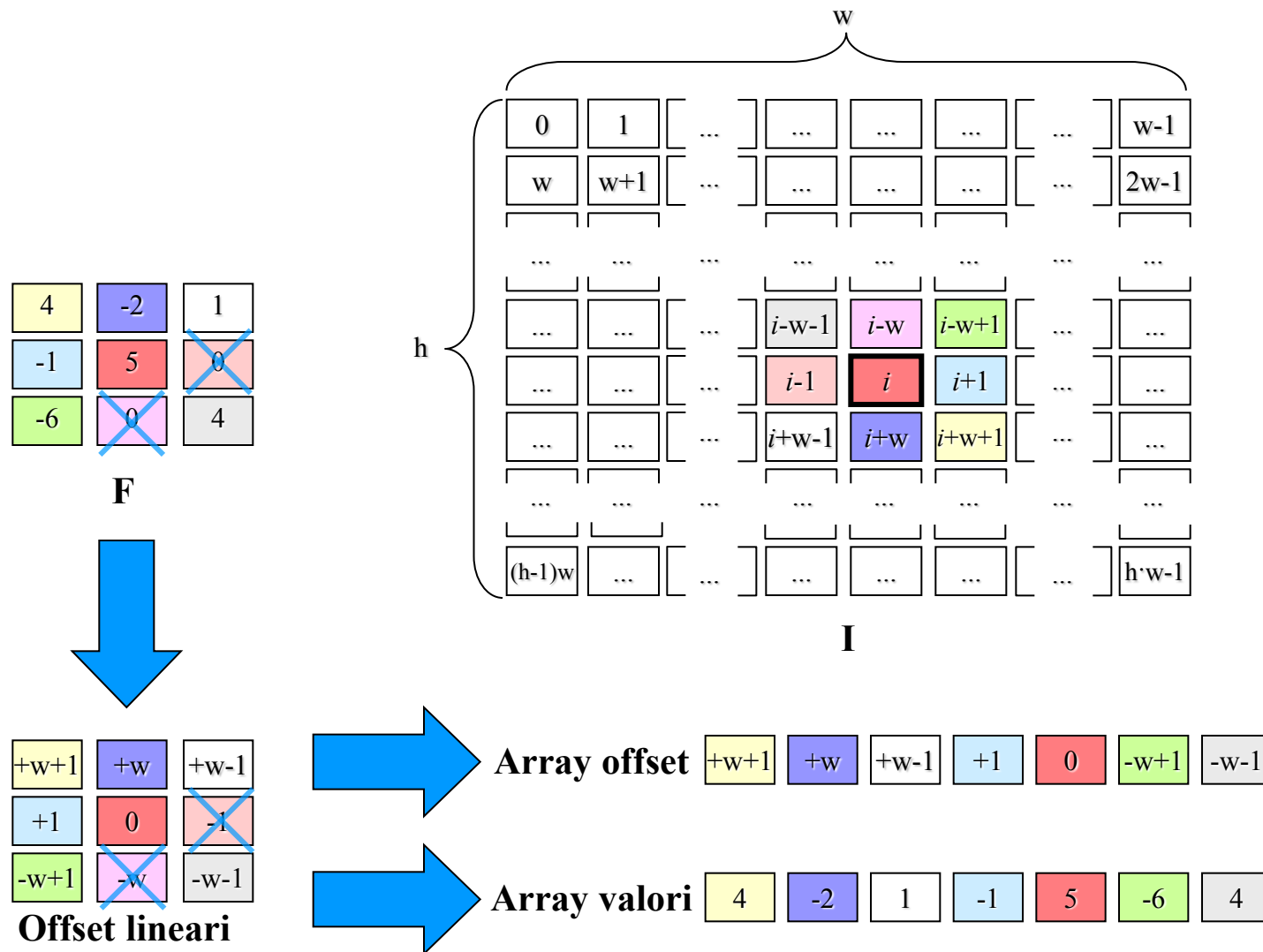
```



Calcolo efficiente della convoluzione

- L'implementazione nel lucido precedente è piuttosto inefficiente
 - L'accesso ai pixel e ai valori del filtro mediante due indici richiede ogni volta un prodotto e una somma
 - Alcune operazioni (somma/sottrazione di m^2 e y) sono inutilmente ripetute nel ciclo più interno
- Un'implementazione più efficiente può essere facilmente ottenuta attraverso la “linearizzazione” del filtro:
 - Si calcolano in anticipo gli offset dei soli elementi diversi da zero rispetto alla posizione del pixel su cui applicare il filtro
 - Permette di operare su vettori monodimensionali (sia per l'immagine che per il filtro)
 - Particolarmente vantaggiosa se molti elementi del filtro sono 0

Filtro “linearizzato” e offset precalcolati



Filtro “linearizzato” e offset precalcolati (2)

```
...
int nM = Filter.Size * Filter.Size;
int[] FOff = new int[nM];
int[] FVal = new int[nM];
int maskLen = 0;
for (int y = 0; y < Filter.Size; y++)
    for (int x = 0; x < Filter.Size; x++)
        if (Filter[y, x] != 0)
        {
            FOff[maskLen] = (m2 - y) * w + (m2 - x);
            FVal[maskLen] = Filter[y, x]; maskLen++;
        }
int index = m2 * (w + 1); // indice lineare all'interno dell'immagine
int indexStepRow = m2 * 2; // aggiustamento indice a fine riga (salta bordi)
for (int y = y1; y <= y2; y++, index += indexStepRow)
    for (int x = x1; x <= x2; x++)
    {
        int val = 0;
        for (int k = 0; k < maskLen; k++)
            val += InputImage[index + FOff[k]] * FVal[k];
        Result[index++] = val / Filter.Denominator;
    }
```

Convoluzione: confronto di alcune implementazioni

■ Caso di prova:

- Convoluzione con filtro 15x15 su immagine grayscale 3072x2304
- Filtro senza valori nulli
- Pentium IV 3,4GHz con hyperthreading

Implementazione	Tempo di esecuzione (sec)	
	Debug	Release
C# - Impl. di base con array bidimensionali	45.0	16.7
C# - Impl. con offset precalcolati e array monodimensionali	22.8	6.7
C# - Impl. con offset precalcolati e array monodimensionali + codice unsafe (puntatori)	11.1	5.4
C/C++ - Impl. con offset precalcolati e puntatori	14.3	3.1

Calcolo efficiente della convoluzione (2)

■ Filtri separabili

- Un filtro è *separabile* se può essere espresso come prodotto di un vettore colonna per un vettore riga
- Utilizzando un filtro separabile ($m \times m$) è possibile calcolare la convoluzione **applicando** all'immagine **in sequenza due filtri mono-dimensionali** anziché un filtro bi-dimensionale
- La complessità si riduce da $O(m^2n^2)$ a $O(2mn^2)$
- Nel caso di prova visto in precedenza, il tempo di esecuzione passa da 6.7 sec a 1.1 sec

$$\mathbf{F} = \mathbf{F}_x \cdot \mathbf{F}_y^t$$

$$\begin{array}{|c|c|c|} \hline \mathbf{F} \\ \hline 1 & 2 & 1 \\ \hline 2 & 4 & 2 \\ \hline 1 & 2 & 1 \\ \hline \end{array} = \begin{array}{|c|} \hline \mathbf{F}_x \\ \hline 1 \\ \hline 2 \\ \hline 1 \\ \hline \end{array} \cdot \begin{array}{|c|c|c|} \hline \mathbf{F}_y^t \\ \hline 1 & 2 & 1 \\ \hline \end{array}$$

$$\mathbf{T}[i, j] = \sum_{x=1}^m (\mathbf{I}[i, j + \lceil \frac{m}{2} \rceil - x] \cdot \mathbf{F}_x[x])$$

$$\mathbf{I}'[i, j] = \sum_{y=1}^m (\mathbf{T}[i + \lceil \frac{m}{2} \rceil - y, j] \cdot \mathbf{F}_y[y])$$

■ Trasformata di Fourier

- Calcolare la convoluzione nel dominio delle frequenze può essere in generale più efficiente, in particolare per filtri a elevata dimensione [→VA2]

Ruotare e ridimensionare un'immagine

■ Mapping diretto

- Sia $f: R \times R \rightarrow R \times R$ una funzione che mappa ogni pixel della vecchia immagine nella nuova; ad esempio, nel caso di trasformazioni affini (traslazione $[t_x, t_y]$ + rotazione θ + scala s), la funzione è:

$$\begin{bmatrix} x_{new} \\ y_{new} \end{bmatrix} = \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \cdot \begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix} \cdot \begin{bmatrix} x_{old} \\ y_{old} \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

- Eseguendo tale trasformazione a partire da una scansione dell'immagine di partenza, si hanno i seguenti problemi:
 - Valori dei nuovi pixel non necessariamente interi (approssimazione)
 - Alcuni pixel vengono mappati al di fuori della nuova immagine
 - Alcuni pixel della nuova immagine non sono coperti ("buchi")

Ruotare e ridimensionare un'immagine (2)

■ Mapping inverso

- Un modo efficace di risolvere tali problemi consiste nell'eseguire la scansione della nuova immagine e, per ogni pixel $[x_{new}, y_{new}]$, determinare il punto di riferimento $[x_{old}, y_{old}]$ nella vecchia immagine attraverso la funzione inversa f^{-1} .

$$\begin{bmatrix} x_{old} \\ y_{old} \end{bmatrix} = \begin{bmatrix} 1/s & 0 \\ 0 & 1/s \end{bmatrix} \cdot \begin{bmatrix} \cos(-\theta) & \sin(-\theta) \\ -\sin(-\theta) & \cos(-\theta) \end{bmatrix} \cdot \left(\begin{bmatrix} x_{new} \\ y_{new} \end{bmatrix} - \begin{bmatrix} t_x \\ t_y \end{bmatrix} \right)$$

- Il punto di riferimento $[x_{old}, y_{old}]$, che è in coordinate continue (floating point), potrebbe cadere:
 - 1) fuori dalla vecchia immagine: tipicamente si utilizza un valore fisso (colore dello sfondo se noto, oppure nero)
 - 2) su di un pixel della vecchia immagine: se ne copia l'intensità
 - 3) in una posizione intermedia tra 4 pixel della vecchia immagine: si può applicare una tecnica di interpolazione (vedi lucido seguente)

Ruotare e ridimensionare un'immagine (3)

■ Interpolazione di Lagrange

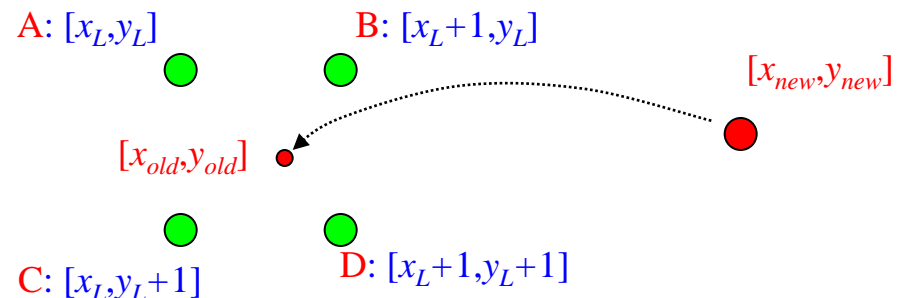
- Il valore di intensità del nuovo pixel $[x_{new}, y_{new}]$ viene calcolato cercando il piano che meglio approssima i 4 pixel (ai minimi quadrati)
- $I'(x,y)$ e $I(x,y)$ indicano l'intensità dei pixel nella nuova e vecchia immagine rispettivamente, e w_A, w_B, w_C, w_D i pesi

$$w_A = (x_L + 1 - x_{old}) \cdot (y_L + 1 - y_{old})$$

$$w_B = (x_{old} - x_L) \cdot (y_L + 1 - y_{old})$$

$$w_C = (x_L + 1 - x_{old}) \cdot (y_{old} - y_L)$$

$$w_D = (x_{old} - x_L) \cdot (y_{old} - y_L)$$



$$I(x_{new}, y_{new}) = \frac{I(A) \cdot w_A + I(B) \cdot w_B + I(C) \cdot w_C + I(D) \cdot w_D}{\underbrace{w_A + w_B + w_C + w_D}_{=1}}$$