

Elim Ferone

Sommario

Introduzione ad OpenCV (Lez.1).....	3
Caricamento e visualizzazione di un'immagine (primo programma).....	4
Funzioni utili OpenCV (Lez.2).....	5
Funzione imread.....	5
Funzione imwrite.....	6
Strutture dati.....	6
Basic data types.....	6
Struttura Mat.....	7
Allocazione.....	8
Costruttori.....	8
Costruttori di copia.....	9
Altre funzioni.....	10
Accesso agli elementi.....	10
Operazioni algebriche.....	12
Utilities.....	13
Padding.....	13
Filtraggio spaziale (Lez.3-4).....	15
Filtraggio lineare.....	15
Correlazione e convoluzione.....	16
Rappresentazione vettoriale.....	19
Correlazione e convoluzione OpenCV.....	19
Specifiche del filtro.....	20
Filtri di smoothing.....	20
Filtri non lineari (basati su statistiche d'ordine).....	22
Thresholding.....	23
Sharpening.....	25
Derivata prima di un'immagine.....	25
Derivata seconda di un'immagine.....	26
Esempio di derivata discreta.....	26
Operatore Laplaciano.....	27

Unsharp masking.....	29
Gradiente e operatori di derivazione.....	30
Colore (Lez.5).....	32
Caratterizzazione del colore.....	32
Diagramma di cromaticità.....	34
Modello (spazio) colore.....	34
Modello RGB (Red Green Blue).....	35
Modello CMY-CMYK (Cyan Magenta Yellow).....	35
Modello HSI (Hue Saturation Intensity).....	36
Elaborazioni full-color.....	37
Smoothing RGB e HSI.....	38
Sharpening RGB e HSI.....	39
Funzione imread per immagini a colori e accesso ai canali.....	39
Funzione cvtColor.....	40
Segmentazione (Lez.6).....	40
Adiacenza, connettività e regione.....	41
Individuazione delle caratteristiche di un'immagine.....	42
Proprietà delle derivate.....	43
Punti isolati.....	43
Linee.....	44
Modelli di edge.....	45
Algoritmo di Marr-Hildreth e LoG.....	48
Canny e Harris (Lez.7).....	50
Algoritmo di Canny (edge detector).....	50
Algoritmo di Harris (corner detector).....	54
Trasformata di Hough (Lez.8).....	59
Trasformata di Hough per rette.....	60
Trasformata di Hough per cerchi.....	63
Caratteristiche della trasformata di Hough.....	65
Trasformata di Hough in OpenCV.....	66
Sogliatura (Lez.9).....	66
Sogliatura globale.....	68
Metodo di Otsu (soglia singola).....	69
Smoothing per migliorare la sogliatura.....	72

Utilizzo degli edge per migliorare la sogliatura.....	72
Metodo di Otsu (soglie multiple).....	74
Sogliatura variabile.....	75
Thresholding OpenCV.....	76
Region Growing e Split and Merge (Lez.10).....	77
Region Growing.....	77
Split and Merge.....	78
Clustering (Lez.11).....	80
Algoritmo K-means.....	80
K-means++.....	83
Algoritmo mean-shift.....	84
Morfologia matematica (Lez.12).....	86
Elementi strutturanti.....	87
Operazioni morfologiche.....	88
Erosione.....	88
Dilatazione.....	89
Dualità.....	89
Apertura.....	90
Chiusura.....	91
Esempio di apertura e chiusura.....	91
Trasformazioni hit or miss.....	92
Morfologia in scala di grigio.....	92
Erosione e dilatazione attraverso ES flat.....	93
Apertura e chiusura attraverso ES flat.....	93
Smoothing e gradiente morfologico.....	94

Introduzione ad OpenCV (Lez.1)

OpenCV è una libreria open source per la computer vision e l'elaborazione delle immagini.

La computer vision consiste nella trasformazione di immagini o video in una decisione o in una nuova rappresentazione (es. decisione “c’è una persona o una macchina nella scena”, es. nuova rappresentazione “rimuovere imperfezioni dall’immagine”).

Per un essere umano la vista è uno dei sensi più sviluppati, ma ottenere lo stesso risultato tramite computer vision risulta molto più difficile. In particolare **un computer vede le immagini come matrici di numeri**, e l'obiettivo della computer vision è quello di trasformare queste matrici in “percezioni”. **OpenCV fornisce quindi gli strumenti di base per risolvere i problemi nell’ambito della computer vision.**

OpenCV ha una struttura a livelli:

- al di sotto del SO troviamo le **interfacce dei linguaggi supportati e le applicazioni**;
- nel livello successivo ci sono le **funzionalità di alto livello**;
- seguite dalle **funzioni di basso livello**;
- infine abbiamo le **ottimizzazioni hardware**, infatti questo livello prende il nome di **Hardware Acceleration Layer**.

Per accedere alle funzioni messe a disposizione da OpenCV è necessario includere gli **header opportuni**:

- `#include "opencv2/opencv.hpp"`, che consente di accedere a tutte le funzioni della libreria;
- `#include "opencv2/imgproc/imgproc.hpp"`, che permette di accedere alle funzioni specifiche per l’elaborazione delle immagini.

Caricamento e visualizzazione di un’immagine (primo programma)

```
#include <opencv2/opencv.hpp>

using namespace cv;

int main(int argc, char** argv) {
    Mat img = imread( filename: argv[1], flags: -1);

    if(img.empty())
        return -1;

    namedWindow( winname: "Example1", flags: WINDOW_AUTOSIZE);
    imshow( winname: "Example1", mat: img);
    waitKey( delay: 0);
    destroyWindow( winname: "Example1");

    return 0;
}
```

Dopo aver incluso l’header opportuno, utilizziamo il namespace cv, in modo da evitare di dover richiamare il prefisso cv:: per le funzioni OpenCV.

La funzione **imread** si occupa di **leggere l’immagine** che passeremo come primo argomento, in questo caso da riga di comando (`argv[1]`), mantenendola invariata,

se il parametro è -1, ossia “**unchanged**”, rappresentandola a colori, se il parametro è 0, ossia “**color**”, o rappresentandola in scala di grigio, se il parametro è 1, ossia “**grayscale**”. La funzione, inoltre, determina automaticamente il tipo di file (JPEG, PNG, ecc.) e alloca la memoria necessaria per la struttura dati (Mat) che conterrà i dati dell’immagine.

Successivamente viene richiamato il **metodo empty()** per verificare che l’immagine sia stata letta correttamente.

Per visualizzare l’immagine bisogna utilizzare la funzione **namedWindow**, la quale creerà una finestra che avrà come **nome** il primo parametro passato alla funzione e come **dimensione** quella passata come secondo parametro (in questo caso **WINDOW_AUTOSIZE**, quindi la finestra avrà la dimensione adatta a contenere l’immagine letta nella imread nella sua dimensione originale).

Per mostrare l’immagine utilizziamo la **funzione imshow**, che prende in input il nome di una finestra precedentemente creata e l’immagine da visualizzare all’interno di essa, a partire dalla struttura dati che contiene i dati di un’immagine.

Si utilizza inoltre la **funzione waitKey** per mantenere la finestra visualizzata per il numero di millisecondi passati come parametro. Se passiamo 0, però, la finestra rimarrà aperta fino a quando non verrà premuto un tasto qualsiasi.

Una volta premuto un tasto, o scaduto il tempo, si passa alla funzione **destroyWindow** che distrugge la finestra che ha come nome quello passato come parametro della funzione. La distruzione della finestra può avvenire solo se la finestra è stata precedentemente registrata tramite una **namedWindow** e se la finestra non è stata chiusa forzatamente (tramite “x” in alto a destra della finestra) durante la visualizzazione dovuta alla **waitKey**.

Le strutture che contengono i dati delle immagini di tipo **Mat** vengono automaticamente deallocate al termine del programma.

Funzioni utili OpenCV (Lez.2)

Funzione imread

La funzione **imread()** si occupa di leggere l’immagine che passeremo come primo argomento, mantenendola invariata, se il parametro è -1, ossia “**unchanged**”, rappresentandola a colori, se il parametro è 0, ossia “**color**”, o rappresentandola in scala di grigio, se il parametro è 1, ossia “**grayscale**”. La funzione, inoltre, determina automaticamente il tipo di file (JPEG, PNG, ecc.) e alloca la memoria necessaria per la struttura dati (Mat) che conterrà i dati dell’immagine.

```

cv::Mat cv::imread(
    const string& filename,           // Input filename
    int         flags   = cv::IMREAD_COLOR // Flags set how to interpret file
);

```

Parameter ID	Meaning	Default
cv::IMREAD_COLOR	Always load to three-channel array.	yes
cv::IMREAD_GRAYSCALE	Always load to single-channel array.	no
cv::IMREAD_ANYCOLOR	Channels as indicated by file (up to three).	no
cv::IMREAD_ANYDEPTH	Allow loading of more than 8-bit depth.	no
cv::IMREAD_UNCHANGED	Equivalent to combining: cv::IMREAD_ANYCOLOR cv::IMREAD_ANYDEPTH ^a	no

Funzione imwrite

La funzione imwrite() si occupa di **salvare l'immagine** che ha come **nome** quello passato come **primo argomento**, dal quale la funzione determinerà il tipo di file, come **struttura dati** quella passata come **secondo argomento** e un **vettore di parametri opzionali dipendenti dal formato dell'immagine**.

```

bool cv::imwrite(
    const string&     filename,           // Input filename
    cv::InputArray     image,              // Image to write to file
    const vector<int>& params = vector<int>() // (Optional) for parameterized fmts
);

```

Parameter ID	Meaning	Range	Default
cv::IMWRITE_JPG_QUALITY	JPEG quality	0-100	95
cv::IMWRITE_PNG_COMPRESSION	PNG compression (higher values mean more compression)	0-9	3
cv::IMWRITE_PXM_BINARY	Use binary format for PPM, PGM, or PBM files	0 or 1	1

Strutture dati

Le strutture dati utilizzate da OpenCV possono essere divise in **3 categorie**:

- **basic data types**, che sono **creati a partire dalle primitive del C++** e contengono le **definizioni di vettori e matrici di piccole dimensioni**, ma anche semplici oggetti geometrici come **punti** e **rettangoli**;
- **helper objects**, che rappresentano concetti più astratti, come i **range objects**;
- **large array types**, che contengono **array e altri contenitori di primitive**, come ad esempio la **struttura Mat** che contiene i dati delle immagini.

Basic data types

Una tipologia di basic data types è la **classe template Vec<>**, che rappresenta un **container di primitive**, noto come **fixed vector class**, utilizzato per **array di piccole dimensioni**.

In generale ogni combinazione della seguente forma è valida

`cv::Vec{2,3,4,6}{b,w,s,i,f,d},`

con i numeri nella prima parentesi che rappresentano la **dimensione**, mentre i caratteri nella seconda rappresentano il **tipo degli elementi** del vettore.

Esempi possono essere **Vec2i**, che è un vettore di 2 elementi interi, o anche **Vec4d**, che è un vettore di 4 elementi double.

Di questo tipo di basic data types esiste anche la versione per le matrici, chiamata **Matx<>**, ossia **fixed matrix class**. Anch'essa viene utilizzata per **matrici di piccole dimensioni** ed ogni combinazione della seguente forma è valida

`cv::Matx{1,2,3,4,6}{1,2,3,4,6}{f,d},`

con i numeri nelle prime due parentesi che rappresentano **numero di righe e numero di colonne**, mentre i caratteri nell'ultima parentesi rappresentano il **tipo**.

Esempi possono essere **Matx22f**, che è una matrice 2x2 di elementi float, o anche **Matx33d**, che è una matrice 3x3 di elementi double.

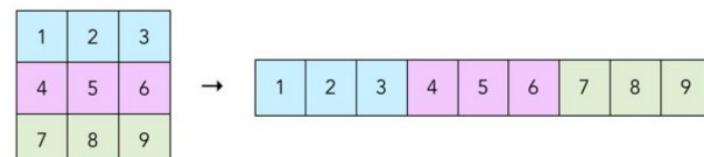
Struttura Mat

La classe Mat rappresenta la **classe centrale dell'intera implementazione C++** della libreria OpenCV, infatti la **maggior parte delle funzioni della libreria sono membri di Mat, ricevono oggetti Mat come argomento o restituiscono oggetti Mat.**

Questa classe viene **utilizzata per rappresentare array densi** di un qualsiasi numero di dimensioni, con l'aggettivo **denso** che indica che per ogni posizione dell'array c'è un valore memorizzato nella corrispondente area di memoria.

All'interno di questa struttura i dati sono memorizzati per riga, in maniera **sequenziale**, ed in pratica per muoversi tra i diversi pixel viene utilizzato un array, detto **step**, che comporta ad esempio uno shift di 1 posizione, nel caso di immagini in bianco e nero, o di 3 posizioni, nel caso di immagini a colori.

Le **immagini a colori** sono rappresentate da **3 valori per ogni pixel**, che corrispondono ai canali RGB, che in OpenCV vanno nell'**ordine BGR**.



	Column 0	Column 1	Column ...	Column m									
Row 0	0,0 1,0	0,0 1,0	0,0 1,0	0,1 1,1	0,1 1,1	0,1 1,1	0, m 1, m	0, m 1, m	0, m 1, m
Row 1	0,0 1,0	0,0 1,0	0,0 1,0	0,1 1,1	0,1 1,1	0,1 1,1	0, m 1, m	0, m 1, m	0, m 1, m
	0,0 1,0	0,0 1,0	0,0 1,0	0,1 1,1	0,1 1,1	0,1 1,1	0, m 1, m	0, m 1, m	0, m 1, m

Ogni matrice ha diversi attributi:

- **flags**, che sono campi di bit che ne specificano il contenuto (della matrice);
- **dims**, che indica il numero di dimensioni;
- **rows e cols**, che indicano il numero di righe e colonne nel caso di matrice bidimensionale;
- **data**, che è un puntatore all'area di memoria in cui sono memorizzati i dati.

Allocazione

E' possibile creare un array semplicemente istanziando una variabile di tipo **Mat**, con l'array che in questo caso non ha né tipo né dimensione, quindi se si vuole allocare memoria bisogna utilizzare il metodo **create()** passando come argomenti il **numero di righe**, il **numero di colonne** e il **tipo**.

Per quanto riguarda il tipo bisogna specificare il tipo fondamentale ed il numero di canali

`CV_{8U,16S,16U,32S,32F,64F}C{1,2,3},`

quindi ad esempio `CV_32F3` corrisponde ad un array che contiene elementi di tipo float a 32 bit con tre canali.

Costruttori

Si può evitare di eseguire i passi di istanziazione e allocazione di memoria separatamente, specificando gli argomenti all'interno del costruttore.

```
cv::Mat m( 3, 10, CV_32FC3, cv::Scalar( 1.0f, 0.0f, 1.0f ) );  
  
cv::Mat m;  
  
// Create data area for 3 rows and 10 columns of 3-channel 32-bit floats  
m.create( 3, 10, CV_32FC3 );
```

Esistono diversi tipi di costruttori, ad esempio quello riportato nell'immagine soprastante prende in input, oltre ai tre argomenti già visti, anche un oggetto di tipo **Scalar**, che permette di inizializzare ogni posizione di un oggetto di tipo **Mat** ai valori BGR passati in input al costruttore di **Scalar**.

Abbiamo poi un costruttore che permette la creazione di un'immagine a partire da dati preesistenti, sfruttando un puntatore **void** a questi (i dati), e definendo anche lo step con diverse macro (come ad es. **AUTO_STEP**, che permette di determinare automaticamente lo spostamento tra i pixel in base al numero di canali e al tipo di dati della matrice).

Infine è anche possibile creare matrici a partire da oggetti di tipo **Size**, che semplicemente contengono le dimensioni di righe e colonne.

Constructor	Description
<code>cv::Mat;</code>	Default constructor
<code>cv::Mat(int rows, int cols, int type);</code>	Two-dimensional arrays by type
<code>cv::Mat(int rows, int cols, int type, const Scalar& s);</code>	Two-dimensional arrays by type with initialization value
<code>cv::Mat(int rows, int cols, int type, void* data, size_t step=AUTO_STEP);</code>	Two-dimensional arrays by type with preexisting data
<code>cv::Mat(cv::Size sz, int type);</code>	Two-dimensional arrays by type (size in sz)
<code>cv::Mat(cv::Size sz, int type, const Scalar& s);</code>	Two-dimensional arrays by type with initialization value (size in sz)

Si possono creare anche **matrici multidimensionali**, ossia con vari piani separati, dichiarando il numero di dimensioni.

Constructor	Description
<code>cv::Mat(cv::Size sz, int type, void* data, size_t step=AUTO_STEP);</code>	Two-dimensional arrays by type with preexisting data (size in sz)
<code>cv::Mat(int ndims, const int* sizes, int type);</code>	Multidimensional arrays by type
<code>cv::Mat(int ndims, const int* sizes, int type, const Scalar& s);</code>	Multidimensional arrays by type with initialization value
<code>cv::Mat(int ndims, const int* sizes, int type, void* data, size_t step=AUTO_STEP);</code>	Multidimensional arrays by type with preexisting data

Costruttori di copia

Un'importante possibilità risiede nell'utilizzo dei costruttori di copia, che permettono di creare una matrice a partire da un'altra. In questo caso è possibile anche creare una copia a partire da un sottoinsieme della matrice, che si può ottenere tramite range, specificando righe e colonne, oppure tramite un oggetto Rect (Rectangle), che prevede la definizione delle coordinate del pixel in alto a sinistra e delle due dimensioni width e height.

Ulteriori possibilità per quanto riguarda i costruttori di copia sono la possibilità di definire un array di range per array multidimensionali, o anche la possibilità di creare una matrice da espressioni logico/matematiche, le quali sono di tipo MatExpr.

Constructor	Description
<code>cv::Mat(const Mat& mat);</code>	Copy constructor
<code>cv::Mat(const Mat& mat, const cv::Range& rows, const cv::Range& cols);</code>	Copy constructor that copies only a subset of rows and columns
<code>cv::Mat(const Mat& mat, const cv::Rect& roi);</code>	Copy constructor that copies only a subset of rows and columns specified by a region of interest
<code>cv::Mat(const Mat& mat, const cv::Range* ranges);</code>	Generalized region of interest copy constructor that uses an array of ranges to select from an n -dimensional array
<code>cv::Mat(const cv::MatExpr& expr);</code>	Copy constructor that initializes <code>m</code> with the result of an algebraic expression of other matrices

Altre funzioni

La classe Mat fornisce anche dei **metodi statici** per creare degli array di uso comune, come ad esempio quelli **inizializzati a 0**, **ad 1**, o alla **matrice identica**, ossia formata da valori 1 solo sulla diagonale principale, con tutti gli altri elementi a 0.

Function	Description
<code>cv::Mat::zeros(rows, cols, type);</code>	Create a <code>cv::Mat</code> of size <code>rows × cols</code> , which is full of zeros, with type <code>type</code> (CV_32F, etc.)
<code>cv::Mat::ones(rows, cols, type);</code>	Create a <code>cv::Mat</code> of size <code>rows × cols</code> , which is full of ones, with type <code>type</code> (CV_32F, etc.)
<code>cv::Mat::eye(rows, cols, type);</code>	Create a <code>cv::Mat</code> of size <code>rows × cols</code> , which is an identity matrix, with type <code>type</code> (CV_32F, etc.)

Accesso agli elementi

Esistono diversi modi per accedere agli elementi di una matrice, ma le opzioni principali sono per **posizione** o per **iterazione**.

Per quanto riguarda l'**accesso diretto** l'approccio principale è il **metodo template at<>()**, che richiede il **tipo di elementi** a cui si dovrà accedere e, all'interno delle parentesi, gli **indici di riga e colonna** per accedere al dato.

- Es. matrice 10x10 single channel

```
cv::Mat m = cv::Mat::eye( 10, 10, 32FC1 );
printf(
    "Element (3,3) is %f\n",
    m.at<float>(3,3)
);
```

- Es. matrice 10x10 multichannel

```
cv::Mat m = cv::Mat::eye( 10, 10, 32FC2 );
printf(
    "Element (3,3) is (%f,%f)\n",
    m.at<cv::Vec2f>(3,3)[0],
    m.at<cv::Vec2f>(3,3)[1]
);
```

E' possibile accedere ad una posizione anche passando un oggetto di tipo **Point**, che avrà **coordinate x e y**, esattamente come gli indici di riga e colonna.

Example	Description
M.at<int>(i);	Element i from integer array M
M.at<float>(i, j);	Element (i, j) from float array M
M.at<int>(pt);	Element at location (pt.x, pt.y) in integer matrix M
M.at<float>(i, j, k);	Element at location (i, j, k) in three-dimensional float array M
M.at<uchar>(idx);	Element at n-dimensional location indicated by idx[] in array M of unsigned characters

Per accedere invece **ad un'intera riga** si può utilizzare il **metodo template ptr<>()**, che prende come argomento l'indice della riga e restituisce un puntatore al primo elemento.

Come detto in precedenza si può accedere agli elementi della matrice anche attraverso l'uso di iteratori, che possono essere di tipo **MatIterator<>** o **MatConstIterator<>** in base a se sono costanti o meno, considerando che **quegli costanti non permettono di modificare i valori della struttura che rappresentano**.

Ovviamente gli iteratori devono essere dichiarati dello stesso tipo degli oggetti contenuti nell'array.

Esistono inoltre dei metodi della classe Mat, ossia **begin()** ed **end()**, che **restituiscono oggetti iteratori che puntano rispettivamente al primo elemento e alla posizione successiva all'ultimo elemento della matrice.**

Di seguito abbiamo un esempio di utilizzo degli iteratori per calcolare il massimo valore tra le norme 2 di tutti i vettori presenti all'interno della matrice M, che è tridimensionale.

```
int sz[3] = { 4, 4, 4 };
cv::Mat m( 3, sz, CV_32FC3 ); // A three-dimensional array of size 4-by-4-by-4
cv::randu( m, -1.0f, 1.0f ); // fill with random numbers from -1.0 to 1.0

float max = 0.0f;           // minimum possible value of L2 norm
cv::MatConstIterator<cv::Vec3f> it = m.begin();
while( it != m.end() ) {

    len2 = (*it)[0]*(*it)[0]+(*it)[1]*(*it)[1]+(*it)[2]*(*it)[2];
    if( len2 > max ) max = len2;
    it++;
}
```

Un'altra situazione tipica consiste nell'**accedere ad un sottoinsieme dell'array**, che può avvenire tramite i metodi **row()** e **col()** che ricevono come argomento un intero, restituendo la riga o la colonna corrispondente al valore passato in input.

Esistono poi delle varianti, **rowRange()** e **colRange()**, che consentono di **estrarre un array di righe o colonne contigue**. Questi metodi possono prendere in input

due interi, che rappresentano l'**inizio e la fine** della porzione, con il primo incluso e l'ultimo escluso, oppure un oggetto Range.

Si può inoltre utilizzare direttamente l'**operatore “()” per estrarre sottomatrici**, passando come argomento due Range, uno per le righe ed uno per le colonne, oppure un oggetto di tipo Rect.

Example	Description
<code>m.row(i);</code>	Array corresponding to row <code>i</code> of <code>m</code>
<code>m.col(j);</code>	Array corresponding to column <code>j</code> of <code>m</code>
<code>m.rowRange(i0, i1);</code>	Array corresponding to rows <code>i0</code> through <code>i1-1</code> of matrix <code>m</code>
<code>m.rowRange(cv::Range(i0, i1));</code>	Array corresponding to rows <code>i0</code> through <code>i1-1</code> of matrix <code>m</code>
<code>m.colRange(j0, j1);</code>	Array corresponding to columns <code>j0</code> through <code>j1-1</code> of matrix <code>m</code>
<code>m.colRange(cv::Range(j0, j1));</code>	Array corresponding to columns <code>j0</code> through <code>j1-1</code> of matrix <code>m</code>
<code>m.diag(d);</code>	Array corresponding to the <code>d</code> -offset diagonal of matrix <code>m</code>
<code>m(cv::Range(i0,i1), cv::Range(j0,j1));</code>	Array corresponding to the subrectangle of matrix <code>m</code> with one corner at <code>i0, j0</code> and the opposite corner at <code>(i1-1, j1-1)</code>
<code>m(cv::Rect(i0,i1,w,h));</code>	Array corresponding to the subrectangle of matrix <code>m</code> with one corner at <code>i0, j0</code> and the opposite corner at <code>(i0+w-1, j0+h-1)</code>
<code>m(ranges);</code>	Array extracted from <code>m</code> corresponding to the subvolume that is the intersection of the ranges given by <code>ranges[0]</code> - <code>ranges[nDim-1]</code>

E' importante sapere, inoltre, che i metodi che accedono a sottoinsiemi di dati non creano una copia dei dati originali ma accedono direttamente ad essi (dati originali), dunque modificare tali valori equivale a modificare i valori originali.

Operazioni algebriche

Utilizzando la tecnica dell'overload, OpenCV permette di effettuare operazioni algebriche sulle matrici.

Un'operazione algebrica restituisce un oggetto di tipo MatExpr, che può però essere assegnato ad un oggetto di tipo Mat.

Di seguito operazioni algebriche e di vario tipo sulle matrici:

Example	Description
<code>m0 + m1, m0 - m1;</code>	Addition or subtraction of matrices
<code>m0 + s; m0 - s; s + m0, s - m1;</code>	Addition or subtraction between a matrix and a singleton
<code>-m0;</code>	Negation of a matrix
<code>s * m0; m0 * s;</code>	Scaling of a matrix by a singleton
<code>m0.mul(m1); m0/m1;</code>	Per element multiplication of <code>m0</code> and <code>m1</code> , per-element division of <code>m0</code> by <code>m1</code>
<code>m0 * m1;</code>	Matrix multiplication of <code>m0</code> and <code>m1</code>
<code>m0.inv(method);</code>	Matrix inversion of <code>m0</code> (default value of <code>method</code> is DECOMP_LU)
<code>m0.t();</code>	Matrix transpose of <code>m0</code> (no copy is done)
<code>m0>m1; m0>=m1; m0==m1; m0<=m1; m0<m1;</code>	Per element comparison, returns uchar matrix with elements 0 or 255

Example	Description
<code>m0&m1; m0 m1; m0^m1; ~m0; m0&s; s&m0; m0 s; s m0; m0^s; s^m0;</code>	Bitwise logical operators between matrices or matrix and a singleton
<code>min(m0,m1); max(m0,m1); min(m0,s); min(s,m0); max(m0,s); max(s,m0);</code>	Per element minimum and maximum between two matrices or a matrix and a singleton
<code>cv::abs(m0);</code>	Per element absolute value of m0
<code>m0.cross(m1); m0.dot(m1);</code>	Vector cross and dot product (vector cross product is defined only for 3×1 matrices)
<code>cv::Mat::eye(Nr, Nc, type); cv::Mat::zeros(Nr, Nc, type); cv::Mat::ones(Nr, Nc, type);</code>	Class static matrix initializers that return fixed $N_r \times N_c$ matrices of type type

Utilities

Abbiamo poi delle funzioni che possiamo definire “utilities”, come ad esempio **clone()** e **copyTo()** che permettono di effettuare una **copia vera e propria di una matrice**, assegnandola ad una nuova matrice. In questo caso le **due matrici non avranno alcun tipo di legame tra loro**, quindi modificare i valori in una non equivale a modificarli anche nell’altra.

Esiste anche una versione della **copyTo()** che permette di copiare un’immagine solo nei pixel specificati da una **maschera (mask)**, che avrà valori 1 in corrispondenza dei pixel appunto da copiare.

Molto utili sono i metodi **push_back()** e **pop_back()** che permettono di **inserire o rimuovere righe da una matrice**.

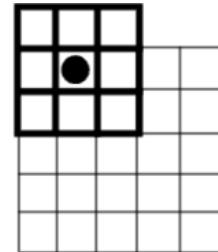
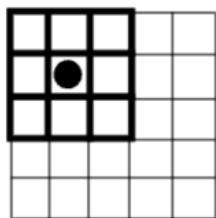
Example	Description
<code>m1 = m0.clone();</code>	Make a complete copy of m0, copying all data elements as well; cloned array will be continuous
<code>m0.copyTo(m1);</code>	Copy contents of m0 onto m1, reallocating m1 if necessary (equivalent to <code>m1=m0.clone()</code>)
<code>m0.copyTo(m1, mask);</code>	Same as <code>m0.copyTo(m1)</code> , except only entries indicated in the array mask are copied
<code>m0.convertTo(m1, type, scale, offset)</code>	Convert elements of m0 to type (e.g., CV_32F) and write to m1 after scaling by scale (default 1.0) and adding offset (default 0.0)
<code>m0.assignTo(m1, type);</code>	Internal use only (resembles convertTo)
<code>m0.setTo(s, mask);</code>	Set all entries in m0 to singleton value s; if mask is present, set only those values corresponding to nonzero elements in mask
<code>m0.reshape(chan, rows);</code>	Changes effective shape of a two-dimensional matrix; chan or rows may be zero, which implies “no change”; data is not copied
<code>m0.push_back(s);</code>	Extend an $m \times 1$ matrix and insert the singleton s at the end
<code>m0.push_back(m1);</code>	Extend an $m \times n$ by k rows and copy m1 into those rows; m1 must be $k \times n$
<code>m0.pop_back(n);</code>	Remove n rows from the end of an $m \times n$ (default value of n is 1) ^a

Padding

Il padding è un’operazione che **consiste nell’aggiungere ad un’immagine righe e colonne extra**.

Viene spesso utilizzato per **generalizzare alcune operazioni quando vengono applicate ai bordi dell'immagine**, come ad es. sostituire ad ogni pixel il valore medio calcolato nell'intorno 3x3.

Per saperne di più su bordo dell'immagine:



Image

0	0	0	0	0	0	0
0						0
0						0
0						0
0						0
0						0
0	0	0	0	0	0	0

Image

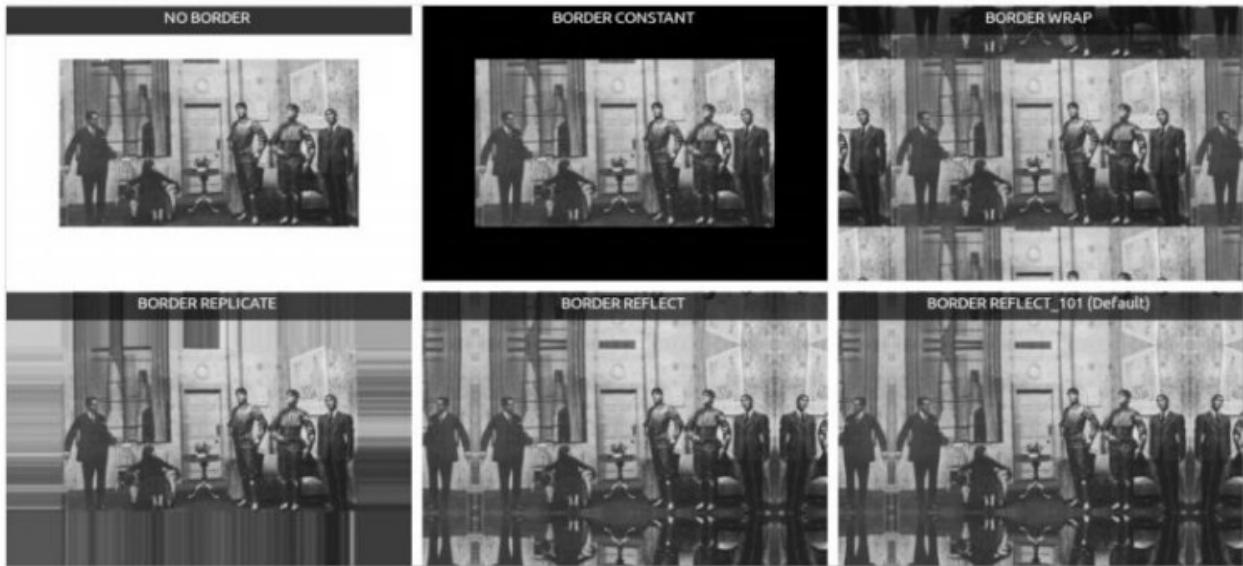
0	0	0	0	0	0	0
0						0
0						0
0						0
0						0
0						0
0	0	0	0	0	0	0

OpenCV fornisce, per effettuare il padding, la funzione **copyMakeBorder()**, la quale prende in input due immagini, la prima di **input** e la seconda che sarà la risultante dell'operazione di padding, 4 interi relativi al numero di pixel da aggiungere nelle varie direzioni, un intero rappresentato da una macro che rappresenta il tipo di bordo, in base al tipo di padding che si vuole effettuare, ed infine un oggetto di tipo **Scalar**, che permette di specificare il valore del bordo, utilizzato nel caso in cui si scelga come tipo di padding quello **BORDER_CONSTANT**.

```
void cv::copyMakeBorder(
    cv::InputArray src,                                // Input image
    cv::OutputArray dst,                               // Result image
    int top,                                         // Top side padding (pixels)
    int bottom,                                       // Bottom side padding (pixels)
    int left,                                         // Left side padding (pixels)
    int right,                                        // Right side padding (pixels)
    int borderType,                                   // Pixel extrapolation method
    const cv::Scalar& value = cv::Scalar()           // Used for constant borders
);
```

Come detto esistono **diversi tipi di padding**:

- **NO_BORDER**, nel caso in cui non si voglia aggiungere bordo;
- **BORDER_CONSTANT**, che aggiunge un bordo costante specificato dal parametro "value" di tipo **Scalar**;
- **BORDER_WRAP**, che in pratica "avvolge" il bordo, nel senso che ad es. i pixel aggiunti a sinistra vengono presi da quelli vicini al bordo a destra, o quelli aggiunti sopra vengono presi da quelli vicini al bordo sotto;
- **BORDER_REPLICATE**, che replica il bordo più vicino;
- **BORDER_REFLECT**, che riflette il bordo in maniera "simmetrica";
- **BORDER_REFLECT_101**, che riflette il bordo come il **BORDER_REFLECT**, ma escludendo il bordo stesso.



Filtraggio spaziale (Lez.3-4)

Il filtraggio è una **tecnica sviluppata per il dominio delle frequenze che permette di lasciar passare o bloccare alcuni elementi** (ossia alcune frequenze).

Tramite questa tecnica ci si può occupare di due tipi di **variazioni di intensità dei pixel dell'immagine**, ossia i **repentini**, quindi alte frequenze, o i **graduali**, quindi basse frequenze.

Gli **effetti del processo di filtraggio** sono l'**attenuazione**, che attenua le variazioni di intensità dei pixel, e il **miglioramento dei dettagli**, che permette di esaltare delle caratteristiche.

Le **tecniche di filtraggio spaziale operano sui pixel di un'immagine considerando il loro intorno** ed in particolare, per ogni pixel dell'immagine originale, si va a calcolare l'intensità del pixel corrispondente nell'immagine filtrata.

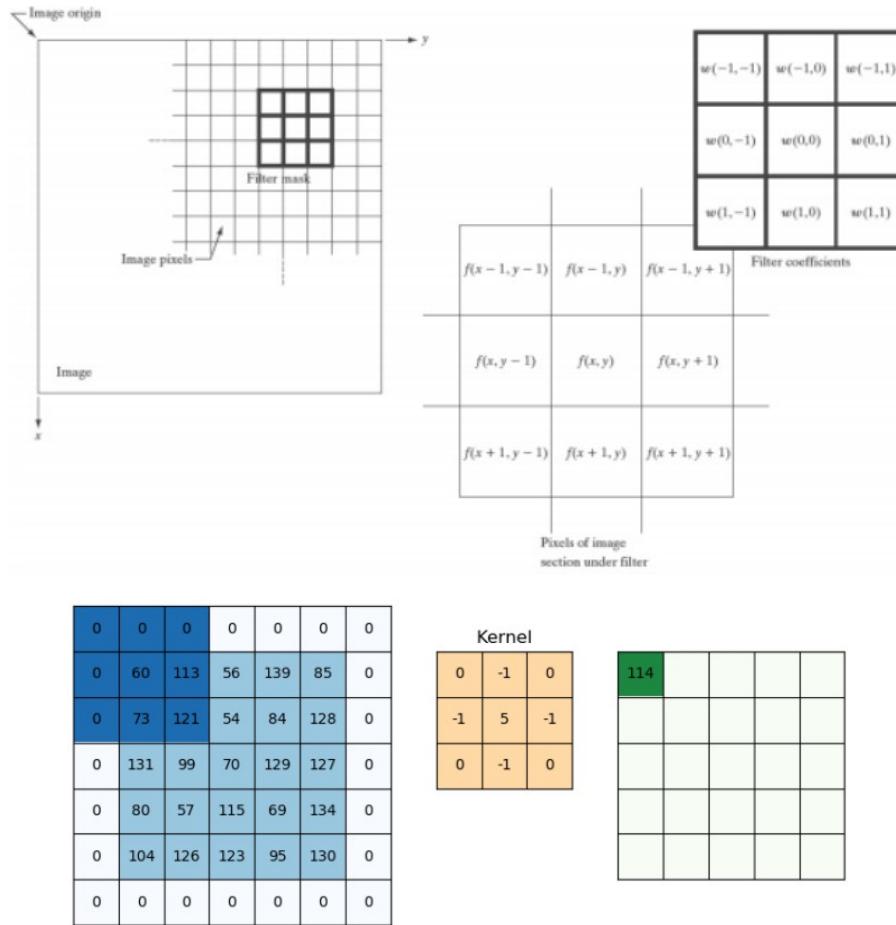
La **regola di trasformazione è spesso descritta da una matrice**, chiamata **filtro o maschera**, della **stessa dimensione dell'intorno**. Se la regola di trasformazione è una funzione lineare delle intensità nell'intorno, allora si parlerà di **filtraggio lineare spaziale**, altrimenti **non lineare**.

Filtraggio lineare

Nelle immagini filtrate, ogni pixel $g(x,y)$ è ottenuto come **combinazione lineare dei pixel nell'immagine originale f , in un intorno di (x,y)**

$$g(x, y) = \sum_{s=-a}^a \sum_{t=-b}^b w(s, t) f(x + s, y + t),$$

con $w(s,t)$ il **filtro**, all'interno del quale vi sono dei **pesi**, che verranno moltiplicati per i corrispondenti valori dell'immagine di input f . Per quanto riguarda le sommatorie, invece, **a e b sono le dimensioni del filtro** (visto che dobbiamo andare da $[-a,-b]$ ad $[a,b]$ significa che per una matrice 3×3 dobbiamo andare da $[-1,-1]$ ad $[1,1]$), che sarà rappresentato, solitamente, da una matrice con un numero dispari di righe e colonne (in modo da avere una sorta di simmetria, a partire dalla colonna centrale).



Correlazione e convoluzione

La formula vista nel paragrafo precedente, applicata su ogni pixel, prende il nome di **correlazione**, con cui appunto si intende il **progressivo scorrimento di una maschera sull'immagine ed il calcolo della somma dei prodotti in ogni posizione** (combinazione lineare).

Per spiegarne il **funzionamento** (della **correlazione 1D**) prendiamo in considerazione un **impulso unitario discreto** f , detto così poiché presenta un unico 1, e un filtro w . Per applicare il filtro bisognerà effettuare uno **zero padding**, sia a sinistra che a destra, del size del filtro meno 1 ($w-1$) e poi allineare l'ultimo elemento del filtro con il primo elemento dell'impulso.

Impulso unitario discreto

f $0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0$	w $1 \ 2 \ 3 \ 2 \ 8$
\downarrow f w $1 \ 2 \ 3 \ 2 \ 8$	
\uparrow starting position alignment	
$f \quad \overline{0 \ 0 \ 0 \ 0} \quad 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0$ $w \quad 1 \ 2 \ 3 \ 2 \ 8$	

zero padding

Ad ogni passo bisognerà quindi effettuare il **prodotto $w*f$** , per poi alla fine eseguire il **cropping**, ossia rimuovere gli 0 precedentemente aggiunti con il padding.

$$\begin{array}{r}
 f \quad 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\
 w \quad 1 \ 2 \ 3 \ 2 \ 8 \\
 w * f \quad 0
 \end{array}$$

$$\begin{array}{r}
 f \quad 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\
 w \quad 1 \ 2 \ 3 \ 2 \ 8 \\
 w * f \quad 0 \ 0
 \end{array}$$

$$\begin{array}{r}
 f \quad 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\
 w \quad 1 \ 2 \ 3 \ 2 \ 8 \\
 w * f \quad 0 \ 0 \ 0
 \end{array}$$

$$\begin{array}{r}
 f \quad 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\
 w \quad 1 \ 2 \ 3 \ 2 \ 8 \\
 w * f \quad 0 \ 0 \ 0 \ 8
 \end{array}$$

$$\begin{array}{r}
 f \quad 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\
 w \quad 1 \ 2 \ 3 \ 2 \ 8 \\
 w * f \quad 0 \ 0 \ 0 \ 8 \ 2 \ 3 \ 2 \ 1 \ 0 \ 0 \ 0 \ 0
 \end{array}$$

$$\cdots$$

$$\begin{array}{r}
 f \quad 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\
 w \quad 1 \ 2 \ 3 \ 2 \ 8 \\
 w * f \quad 0 \ 0 \ 0 \ 8 \ 2 \ 3 \ 2 \ 1 \ 0 \ 0 \ 0 \ 0
 \end{array}$$

$$\begin{array}{r}
 w * f \quad 0 \ 0 \ 0 \ 8 \ 2 \ 3 \ 2 \ 1 \ 0 \ 0 \ 0 \ 0 \\
 \text{Cropping} \quad \underline{\quad 0 \ 8 \ 2 \ 3 \ 2 \ 1 \ 0 \ 0 \quad} \quad w \text{ ruotato di } 180^\circ \\
 f \quad 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0
 \end{array}$$

Il risultato della correlazione sarà quindi il filtro stesso, ma ruotato di 180°.

La **convoluzione**, invece, prevede una **pre-rotazione del filtro di 180°**, con la successiva applicazione delle stesse operazioni viste nella correlazione.

$$\begin{array}{ccccccc}
 & & f & & & & \\
 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
 w & 8 & 2 & 3 & 2 & 1 & & \\
 \downarrow & & & & & & & \\
 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
 & & & & & & & \\
 \uparrow & \text{starting position alignment} & & & & & & \\
 & & & & \cdots & & & \\
 w * f & 0 & 0 & 0 & 1 & 2 & 3 & 2 & 8 & 0 & 0 & 0 & 0 \\
 \text{Cropping} & \hline & 0 & 1 & 2 & 3 & 2 & 8 & 0 & 0 & \hline \\
 f & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
 \end{array}$$

La **formula da applicare nella convoluzione** è uguale a quella della correlazione ma con s e t negati, così che ci si trovi esattamente a 180° rispetto alla correlazione

Correlazione

$$g(x, y) = w(x, y) * f(x, y) = \sum_{s=-a}^a \sum_{t=-b}^b w(s, t) f(x+s, y+t)$$

Convoluzione

$$g(x, y) = w(x, y) * f(x, y) = \sum_{s=-a}^a \sum_{t=-b}^b w(s, t) f(x-s, y-t)$$

Abbiamo quindi visto correlazione e convoluzione 1D, ma anche su **2D** il discorso non cambia. Di seguito possiamo osservare un esempio delle due formule applicate alle matrici.

		Padded f		
		0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		
↙ Origin $f(x, y)$		0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0		
\sum		$w(x, y)$		
		0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		
\sum		Initial position for w	Full correlation result	Cropped correlation result
\sum		1 2 -3 0 0 0 0 0 0 4 5 6 0 0 0 0 0 0 7 -8 9 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 9 8 7 0 0 6 5 4 0 0 3 2 1 0 0 0 0 0 0
\sum		Rotated w	Full convolution result	Cropped convolution result
\sum		9 8 7 0 0 0 0 0 0 6 5 4 0 0 0 0 0 0 3 -2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 1 2 3 0 0 4 5 6 0 0 7 8 9 0 0 0 0 0 0

Rappresentazione vettoriale

Quando la posizione relativa dei coefficienti non è importante, è possibile rappresentare la risposta del filtro utilizzando una rappresentazione vettoriale (e non una matrice), specificando però un'indicizzazione convenzionale.

L'operazione viene vista come **w trasposto**, ossia il **vettore del filtro ma trasposto, per z**, che è il **vettore che contiene i valori dei pixel dell'immagine originale**, dunque è un **prodotto righe per colonne**.

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

$$R = w_1 z_1 + \cdots + w_{mn} z_{mn} = \sum_{k=1}^{mn} w_k z_k = \mathbf{w}^T \mathbf{z}$$

Correlazione e convoluzione OpenCV

In OpenCV è possibile effettuare la **correlazione** utilizzando la funzione **filter2D()**

```
cv::filter2D(  
    cv::InputArray src,                      // Input image  
    cv::OutputArray dst,                     // Result image  
    int ddepth,                            // Output depth (e.g., CV_8U)  
    cv::InputArray kernel,                  // Your own kernel  
    cv::Point anchor = cv::Point(-1,-1),    // Location of anchor point  
    double delta = 0,                      // Offset before assignment  
    int borderType = cv::BORDER_DEFAULT // Border extrapolation to use  
)
```

che prende in input **immagine di input** e **immagine risultante dell'operazione, tipo di dato dell'immagine di output, maschera da applicare per filtrare e poi 3 argomenti opzionali**, ossia l'**anchor point**, che indica quale pixel dell'immagine di input corrisponde al centro del kernel durante l'applicazione del filtro (di default è posto al centro del filtro), un valore che può essere utilizzato per aggiungere un **offset** al risultato del filtraggio (di default è 0) e un parametro che specifica come gestire i **bordi dell'immagine di input** durante l'applicazione del filtro (di default è **BORDER_DEFAULT**, ossia non si desidera aggiungere nessun bordo all'immagine di input).

Per effettuare la **convoluzione**, invece, **dobbiamo ruotare il filtro di 180° prima di applicare la funzione filter2D()**. Si può effettuare questa operazione tramite la funzione **rotate()**

```
void rotate(InputArray src, OutputArray dst, int rotateCode);
```

che prende in input, oltre all'immagine di input e quella risultante dell'operazione, anche il tipo di rotazione che vogliamo effettuare, che può essere:

- **ROTATE_90_CLOCKWISE**, ossia rotazione di 90° in senso orario;
- **ROTATE_90_COUNTER_CLOCKWISE**, ossia rotazione di 90° in senso antiorario;
- **ROTATE_180**, ossia rotazione di 180°.

Specifiche del filtro

Per creare un filtro lineare spaziale è necessario specificarne i coefficienti. Questo può avvenire mediante **specifica diretta** o tramite **specifica basata su una funzione**

Specifica diretta:

$$R = \frac{1}{9} \sum_{i=1}^9 z_i \quad \Rightarrow \quad w_i = \frac{1}{9}, \quad \forall i$$

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

Specifica basata su una funzione:

$$h(x, y) = e^{-\frac{x^2+y^2}{2\sigma^2}} \quad \Rightarrow \quad w(s, t) = h(s, t)$$

Per i **filtri non lineari**, invece, **si specifica solo l'intorno** su cui si applicheranno determinate operazioni.

Filtri di smoothing

I filtri di smoothing vengono utilizzati per sfocare le immagini (**blurring**) e per la **riduzione del rumore (denoising)**.

Queste operazioni portano alla **rimozione dei dettagli più piccoli della dimensione del filtro**, al contempo **evidenziando gli oggetti più grandi**.

Per la riduzione del rumore è possibile utilizzare sia filtri lineari che non lineari, a seconda del tipo di rumore (con rumore, nell'ambito delle immagini, si intendono le variazioni casuali o indesiderate nella luminosità o nel colore dei pixel, che possono essere causate da diversi fattori, come ad es. disturbi nel sensore della fotocamera, compressioni digitali, ecc.).

Tipologie di filtri di smoothing lineare sono i **filtri di media**, nei quali i **valori di ogni pixel vengono sostituiti con la media dei livelli di intensità nell'intorno definito dalla maschera**, ciò porta ad enfatizzare le basse frequenze ed attenuare quelle alte.

Esistono **due tipologie principali** di filtri di media:

- **filtro media classico**, o **filtro box**, in cui ogni coefficiente della maschera ha lo stesso peso;
- **filtro media ponderata**, in cui i pesi sono inversamente proporzionali alla distanza dal centro, quindi i pixel più vicini al centro del filtro avranno un peso maggiore.

Filtro box	$\begin{array}{ c c c } \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array}$	$\frac{1}{9} \times$
Filtro media ponderata	$\begin{array}{ c c c } \hline 1 & 2 & 1 \\ \hline 2 & 4 & 2 \\ \hline 1 & 2 & 1 \\ \hline \end{array}$	$\frac{1}{16} \times$

In generale più aumentiamo le dimensioni del filtro e maggiore sarà la sfocatura, il che porterà a perdere gli oggetti più piccoli, evidenziando quelli di dimensioni maggiori. Così facendo, non saremo in grado di riconoscere questi oggetti “più importanti”, ma sapremo che in quell’area ci sono informazioni da estrarre.



In OpenCV esistono diverse funzioni di smoothing, tra cui blur(), boxFilter() e GaussianBlur().

La funzione **blur()** prende in input **immagine di input** e **immagine risultante dell’operazione**, **size** (righe, colonne) **della maschera** da applicare e **due parametri opzionali**, ossia l’**anchor point**, che indica quale pixel dell’immagine di input corrisponde al centro del kernel durante l’applicazione del filtro (di default è posto al centro del filtro) e un parametro che specifica come gestire i **bordi dell’immagine di input** durante l’applicazione del filtro (di default è BORDER_DEFAULT, ossia non si desidera aggiungere nessun bordo all’immagine di input).

```

void cv::blur(
    cv::InputArray src,                                // Input image
    cv::OutputArray dst,                               // Result image
    cv::Size ksize,                                    // Kernel size
    cv::Point anchor = cv::Point(-1,-1), // Location of anchor point
    int borderType = cv::BORDER_DEFAULT // Border extrapolation to use
);

```

La funzione **boxFilter()**, invece, prende in input, oltre ai **parametri utilizzati anche dalla funzione blur()**, il **tipo di dato dell’immagine di output** (ddepth) e, come **parametro opzionale**, un **valore booleano che specificherà se normalizzare o**

meno il filtro. Il processo di normalizzazione garantisce che il risultato finale dell'applicazione del filtro della media non influenzi la luminosità dell'immagine.

```
void cv::boxFilter(
    cv::InputArray src,                                // Input image
    cv::OutputArray dst,                               // Result image
    int ddepth,                                     // Output depth (e.g., CV_8U)
    cv::Size ksize,                                  // Kernel size
    cv::Point anchor = cv::Point(-1,-1),             // Location of anchor point
    bool normalize = true,                            // If true, divide by box area
    int borderType = cv::BORDER_DEFAULT // Border extrapolation to use
);
```

Infine abbiamo la funzione **GaussianBlur()**, la quale **determina i pesi attraverso una funzione gaussiana, anche detta a campana** per via della sua forma.

La funzione GaussianBlur() utilizza **tutti i parametri della funzione blur()**, a parte l'**anchor point**, ma **in più ha due parametri** che indicano **l'ampiezza e l'altezza della campana**. Nel caso in cui vengano lasciati a 0, allora verrà calcolato un valore ottimale in modo autonomo.

```
void cv::GaussianBlur(
    cv::InputArray src,                                // Input image
    cv::OutputArray dst,                               // Result image
    cv::Size ksize,                                  // Kernel size
    double sigmaX,                                 // Gaussian half-width in x-direction
    double sigmaY = 0.0,                            // Gaussian half-width in y-direction
    int borderType = cv::BORDER_DEFAULT // Border extrapolation to use
);
```

Filtri non lineari (basati su statistiche d'ordine)

I filtri non lineari basati su statistiche d'ordine si basano sull'idea, appunto, di **ordinare i pixel contenuti nell'intorno definito dal filtro, in base alla statistica utilizzata**, per poi **sostituire il valore del pixel centrale con un valore dell'insieme ordinato**.

Esempi di filtri di questo tipo sono il **filtro mediano**, che si basa sulla sostituzione con il valore mediano, e i **filtri massimo e minimo**, che si basano appunto sulla sostituzione con il valore massimo o minimo dell'intorno.

```
f = [100, 120, 98, 99, 110, 255, 100, 200, 10]
Ordinato: [10, 98, 99, 100, 100, 110, 120, 200, 255]
```

Media: 121

Mediana: 100

Min: 10

Max: 255

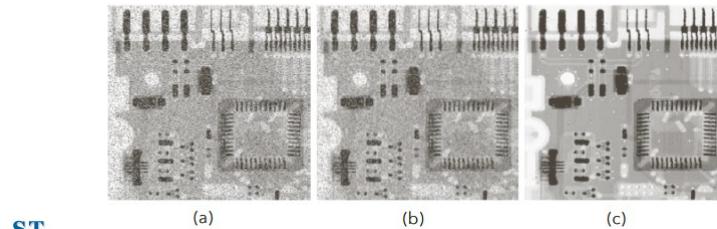
10	98	99
100	100	110
120	200	255

I filtri mediani risultano particolarmente efficaci in presenza di rumore ad **impulso**, anche conosciuto come rumore sale e pepe, che è un tipo di rumore caratterizzato dalla presenza di pixel molto luminosi (sale) e pixel molto scuri (pepe).

(a) Immagine originale, corrotta da rumore sale e pepe

(b) Immagine filtrata con filtro media 3×3

(c) Immagine filtrata con filtro mediano 3×3



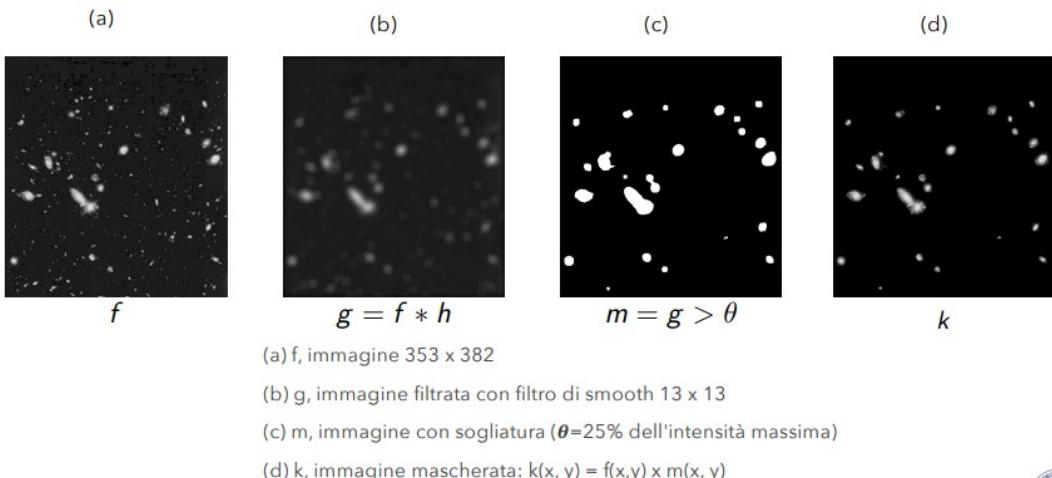
Per applicare il filtro mediano su OpenCV si utilizza la funzione **medianBlur()**, la quale prende in input solo **immagine di input**, **immagine risultante dell'operazione** e **size** (righe, colonne) del filtro.

```
void cv::medianBlur(  
    cv::InputArray src,           // Input image  
    cv::OutputArray dst,          // Result image  
    cv::Size      ksize          // Kernel size  
)
```

Thresholding

Nel processo di rimozione dei dettagli si deve innanzitutto effettuare un'operazione di **smoothing**, in modo da eliminare gli oggetti più piccoli. Dopodichè si procede alla fase di **thresholding (sogliatura)**, in cui tutti i dettagli con intensità minore al valore di soglia vengono impostati a 0, altrimenti a 1. Così facendo **creo una maschera** che verrà **sovraposta all'immagine originale**, dando **in output un'immagine che avrà valori solo nei pixel corrispondenti ai pixel con valore 1 della maschera**. Ovviamente quando vado a sovrapporre devo **moltiplicare i pixel dell'immagine originale con quelli di posizione corrispondente della maschera**, in modo da riottenere i valori nella scala di grigio (considerando di usare un'immagine di questo tipo).

ESEMPIO: RIMOZIONE DI DETTAGLI



UNIVERSITÀ
POLITECNICA
DI TORINO

In OpenCV, per effettuare la sogliatura, abbiamo la funzione **threshold()**, che prende in input **immagine di input** e **immagine risultante dell'operazione**, **valore di soglia**, **valore assegnato ai pixel che superano la soglia** e, infine, il **tipo di sogliatura** da applicare.

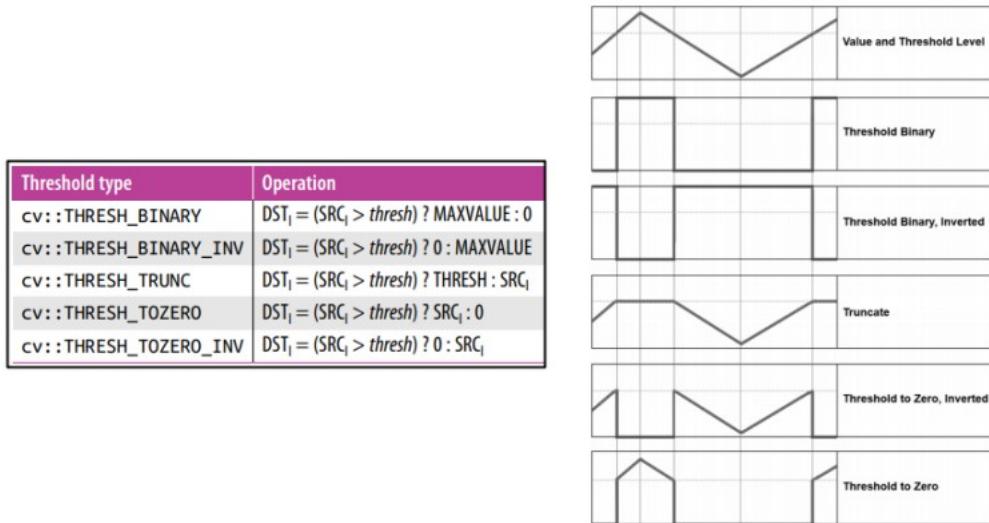
```
double cv::threshold(
    cv::InputArray    src,           // Input image
    cv::OutputArray   dst,           // Result image
    double           thresh,         // Threshold value
    double           maxValue,       // Max value for upward operations
    int              thresholdType  // Threshold type to use
);
```

Un'altra funzione utilizzata per effettuare la sogliatura è **adaptiveThreshold()**, la quale effettua il **processo di sogliatura adattando il valore di soglia localmente**, in base all'intorno dei pixel. La funzione prende in input, oltre a tutti i parametri utilizzati anche dalla funzione **threshold()**, tranne ovviamente il valore di soglia, un **parametro che riguarda il metodo utilizzato per calcolare la soglia adattiva**, che può essere **ADAPTIVE_THRESH_MEAN_C** o **ADAPTIVE_THRESH_GAUSSIAN_C**, la **dimensione dell'intorno utilizzata per calcolare la media pesata che serve per ricavare soglia adattiva** (se metto 3 ad es. si intende 3x3), che deve essere un numero dispari, e **C**, che è una **costante sottratta dalla media calcolata**.

```
void cv::adaptiveThreshold(
    cv::InputArray    src,           // Input image
    cv::OutputArray   dst,           // Result image
    double           maxValue,       // Max value for upward operations
    int              adaptiveMethod, // mean or Gaussian
    int              thresholdType, // Threshold type to use
    int              blockSize,      // Block size
    double           C              // Constant
);
```

La funzione **adaptiveThreshold()** può fornire risultati migliori in situazioni in cui una soglia globale, come quella definita nella funzione **threshold()**, non è efficace a causa di variazioni significative di luminosità e contrasto all'interno dell'immagine.

In precedenza abbiamo parlato dei **diversi tipi di thresholding**, che si differenziano in base al valore che assumono i pixel dopo il confronto con il valore soglia



Sharpening

Il termine **sharpening** si riferisce alle **tecniche adatte ad evidenziare le transizioni di intensità**. Nelle immagini, i bordi (edge) tra gli oggetti vengono percepiti proprio a causa del cambio di intensità. In pratica, più sono nette le transizioni di intensità e più l'immagine viene percepita in modo definito.



La transizione di intensità tra i pixel adiacenti è legata al concetto di derivata dell'immagine in quella posizione. Per questo motivo gli operatori di **sharpening sono definiti proprio attraverso l'uso delle derivate**, con la qualità della risposta che è proporzionale al grado di variazione di intensità nel punto in cui viene applicato l'operatore.

Derivata prima di un'immagine

Visto che l'immagine è una funzione discreta, e non continua, **non si può applicare la definizione classica di derivata**, ed è quindi **necessario definire un operatore che soddisfi le principali proprietà della derivata prima**, ossia:

- uguale a 0 dove l'intensità è costante;
- diversa da 0 per una transizione di intensità;
- costante sulle rampe in cui la transizione di intensità è costante, con le rampe che graficamente vengono rappresentate come delle rette con pendenza costante.

Per implementare queste proprietà utilizziamo la **differenziazione spaziale** a partire dall'operatore di derivazione naturale, ossia **calcoliamo la differenza tra l'intensità dei pixel vicini** (in particolare tra il successivo e quello che stiamo considerando)

$$\frac{\partial f}{\partial x} = f(x+1) - f(x)$$

Derivata seconda di un'immagine

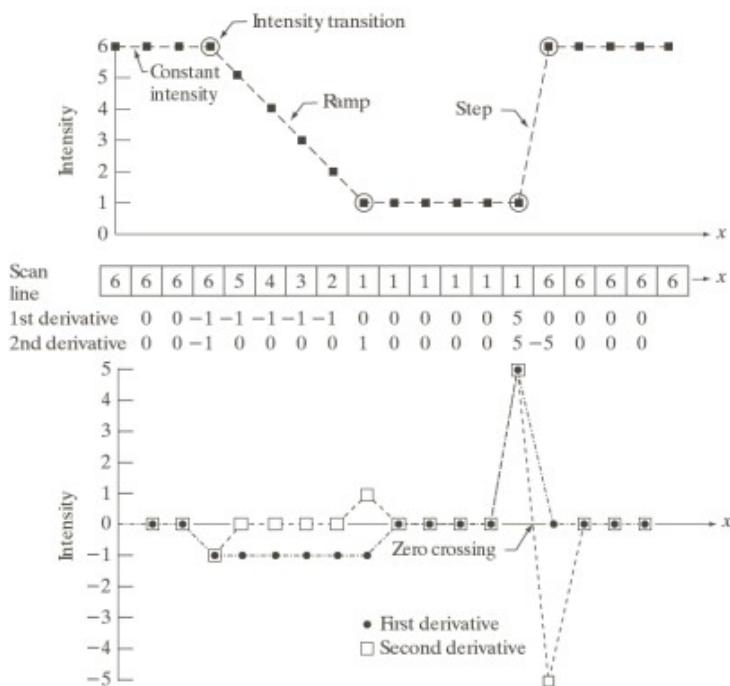
Anche per la derivata seconda **devono essere soddisfatte le principali proprietà**, ossia:

- uguale a 0 dove l'intensità è costante;
- diversa da 0 all'inizio di una rampa;
- uguale a 0 sulle pendenze costanti delle rampe.

Per implementare queste proprietà utilizziamo una **differenziazione sia con il pixel successivo che con quello precedente**, a partire sempre da quello che stiamo considerando

$$\begin{aligned}\frac{\partial^2 f}{\partial x^2} &= f(x+1) - f(x) - (f(x) - f(x-1)) \\ &= f(x+1) - 2f(x) + f(x-1)\end{aligned}$$

Esempio di derivata discreta



Possiamo effettuare osservazioni su questi grafici a partire dalle proprietà da soddisfare, come visto nei due paragrafi precedenti.

Lo **zero crossing** è un punto in cui la retta che passa per due valori attraversa lo 0.

Operatore Laplaciano

L'operatore Laplaciano **implementa una derivata del secondo ordine 2D**, alla cui base vi sono una formulazione discreta della derivata seconda e la costruzione di una maschera che la implementi.

L'idea è quella di realizzare **filtri isotropici**, ossia in cui la risposta è indipendente dalla direzione delle discontinuità dell'immagine ed infatti risultano **invarianti alle rotazioni**.

L'operatore Laplaciano è l'operatore derivativo isotropico più semplice, definito per un'immagine $f(x,y)$

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

In un'immagine digitale, le **derivate seconde rispetto a x e y sono calcolate come**

$$\begin{aligned}\frac{\partial^2 f}{\partial x^2} &= f(x+1, y) - 2f(x, y) + f(x-1, y) \\ \frac{\partial^2 f}{\partial y^2} &= f(x, y+1) - 2f(x, y) + f(x, y-1)\end{aligned}$$

e quindi l'**operatore Laplaciano risulta**

$$\begin{aligned}\nabla^2 f(x, y) &= f(x+1, y) + f(x-1, y) + f(x, y+1) \\ &\quad + f(x, y-1) - 4f(x, y)\end{aligned}$$

0	1	0
1	-4	1
0	1	0

$$\begin{aligned}\nabla^2 f(x, y) &= f(x+1, y) + f(x-1, y) + f(x, y+1) \\ &\quad + f(x, y-1) - 4f(x, y)\end{aligned}$$

Filtro Laplaciano invariante alle rotazioni di 90°

con il filtro che avrà dei pesi tali da risultare **invariante alle rotazioni di 90°** .

Inoltre può essere considerata anche la derivata lungo la diagonale, in modo da realizzare un filtro che avrà pesi tali da risultare **invariante alle rotazioni di 45°** . In tal caso **bisogna aggiungere i seguenti valori a quelli appena visti dell'operatore Laplaciano classico**

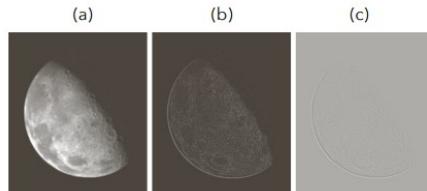
$$\begin{aligned}\nabla^2 f(x, y) &+ f(x-1, y-1) + f(x+1, y+1) \\ &+ f(x-1, y+1) + f(x+1, y-1) - 4f(x, y)\end{aligned}$$

1	1	1
1	-8	1
1	1	1

$$\nabla^2 f(x, y) = f(x-1, y-1) + f(x+1, y+1) + f(x-1, y+1) + f(x+1, y-1) - 4f(x, y)$$

Filtro Laplaciano invariante alle rotazioni di 45°

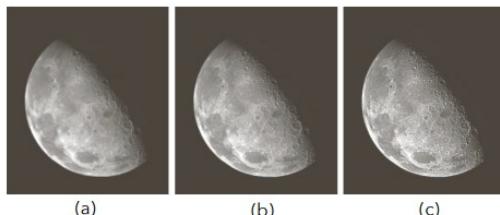
Il Laplaciano ha spesso valori negativi, dunque per visualizzarlo **deve essere opportunamente scalato** nell'intervallo di rappresentazione [0,L-1].



(a) Immagine originale, (b) il suo Laplaciano, (c) il suo Laplaciano scalato in modo che 0 sia visualizzato come livello di grigio intermedio

Il Laplaciano viene utilizzato per evidenziare i bordi (edge) di un'immagine, in quanto permette di trovare le discontinuità, mentre **per effettuare lo sharpening** bisogna poi sommare (teoricamente sottrarre visto che moltiplichiamo il Laplaciano per una costante compresa in [-1,0]) l'immagine originale al Laplaciano, il quale viene moltiplicato per una costante c compresa in [-1,0]. In pratica **più c si avvicina a -1 e più si esaltano i bordi**, mentre con c=0 non si ottiene alcun effetto.

$$g = f + c \nabla^2 f, -1 \leq c \leq 0$$



(a) Immagine originale (b) filtrato con Laplaciano 45° (c) filtrato con Laplaciano 90°

Se si ha un'immagine con angoli squadrati conviene utilizzare il filtraggio **Laplaciano a 90°**, in quanto quello a 45° potrebbe individuare dei bordi che risulterebbero solo fastidiosi. Invece **nel caso in cui l'immagine contenga angoli circolari**, allora è preferibile utilizzare il **Laplaciano a 45°**.

In generale, se non si conosce l'immagine, **si utilizza il filtraggio Laplaciano a 45°**, in quanto solitamente fornisce risultati migliori.

In OpenCV la funzione che implementa l'operatore Laplaciano è **Laplacian()** che prende in input le **immagini di input** e quella **risultante dell'operazione**, il tipo di **dato dell'immagine di output** (ddepth), il **size** da applicare, che se messo a 3 (come di default) indica il Laplaciano a 45° mentre se messo a 1 indica il Laplaciano a 90°, un **coefficiente moltiplicativo** (con default 1), applicato al risultato del calcolo Laplaciano per effettuare una scalatura, un **valore da aggiungere** (con default 0), applicato al risultato del calcolo Laplaciano, ed infine un parametro che specifica come gestire i **bordi dell'immagine di input** durante

l'applicazione del filtro (di default è BORDER_DEFAULT, ossia non si desidera aggiungere nessun bordo all'immagine di input).

```
void cv::Laplacian(
    cv::InputArray src,           // Input image
    cv::OutputArray dst,          // Result image
    int ddepth,                  // Depth of output image (e.g., CV_8U)
    cv::Size ksize = 3,           // Kernel size
    double scale = 1,             // Scale applied before assignment to dst
    double delta = 0,              // Offset applied before assignment to dst
    int borderType = cv::BORDER_DEFAULT // Border extrapolation to use
);
```

In questa funzione risultano **obbligatori solo i primi 3 parametri**, in quanto tutti gli altri hanno valori di default e quindi sono opzionali.

Unsharp masking

L'unsharp masking è un **metodo di uso comune in grafica per rendere le immagini più nitide**.

In pratica i **passi da seguire** sono:

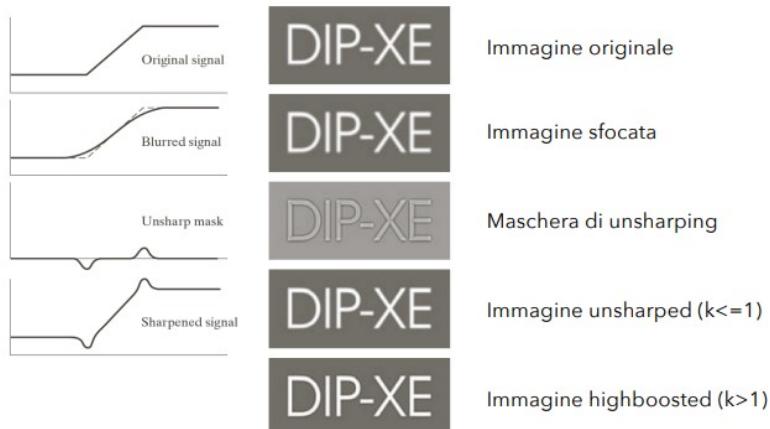
- sfocare l'immagine originale (smoothing);
- ottenere una maschera come differenza tra l'immagine originale e quella sfocata;
- aggiungere la maschera all'immagine originale.

Con il **processo che può essere formalizzato come**

$$g = f + k \cdot (f - f * h),$$

con **f** l'immagine originale, **h** l'immagine sfocata e **k** il coefficiente che regola l'intensità dell'effetto di **sharpening**.

E' bene osservare che se l'immagine fosse affetta da rumore verrebbe esaltato tutto, il che porterebbe ad avere nell'immagine di output un completo caos.



Gradiente e operatori di derivazione

La derivata prima, come visto in precedenza, viene implementata con un'approssimazione del gradiente, che è un vettore formato dalle sue derivate parziali che punta verso la direzione di massima variazione.

$$\nabla f \equiv \text{grad}(f) \equiv \begin{bmatrix} g_x \\ g_y \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix}$$

La magnitudo del gradiente $M(x,y)$ è un'immagine delle stesse dimensioni di f che misura la variazione di intensità, calcolabile con la norma, quindi la lunghezza del vettore, che equivale alla radice quadrata della somma delle derivate parziali al quadrato. Siccome la radice quadrata è un'operazione onerosa, si può sostituire la formula appena descritta con una più banale somma dei valori assoluti delle derivate parziali

$$M(x, y) = \text{mag}(\nabla f) = \sqrt{g_x^2 + g_y^2}$$
$$M(x, y) \approx |g_x| + |g_y|$$

Possiamo distinguere diversi operatori di derivazione, a partire dalle definizioni base, ossia quelle viste in precedenza

$$g_x(x, y) = f(x+1, y) - f(x, y)$$

$$g_y(x, y) = f(x, y+1) - f(x, y)$$

$$g_x: \begin{array}{|c|c|} \hline -1 & 1 \\ \hline \end{array} \quad g_y: \begin{array}{|c|} \hline -1 \\ \hline 1 \\ \hline \end{array},$$

poi abbiamo gli operatori di Roberts, che sono filtri 2x2 difficili da applicare in quanto si preferisce utilizzare filtri con righe e colonne dispari, ed infatti non sono molto utilizzati

$$g_x(x, y) = f(x+1, y+1) - f(x, y)$$

$$g_y(x, y) = f(x, y+1) - f(x-1, y)$$

$$g_x: \begin{array}{|c|c|} \hline -1 & 0 \\ \hline 0 & 1 \\ \hline \end{array} \quad g_y: \begin{array}{|c|c|} \hline 0 & -1 \\ \hline 1 & 0 \\ \hline \end{array},$$

gli operatori di Sobel, che sono filtri 3x3 nei quali è inglobata anche un'operazione di riduzione del rumore (ossia i valori -2 e 2)

$$g_x(x, y) = -f(x-1, y-1) - 2f(x-1, y) - f(x-1, y+1) + f(x+1, y-1) + 2f(x+1, y) + f(x+1, y+1)$$

$$g_y(x, y) = -f(x-1, y-1) - 2f(x, y-1) - f(x+1, y-1) + f(x-1, y+1) + 2f(x, y+1) + f(x+1, y+1)$$

$g_x:$

-1	-2	-1
0	0	0
1	2	1

 $g_y:$

-1	0	1
-2	0	2
-1	0	1

 ,

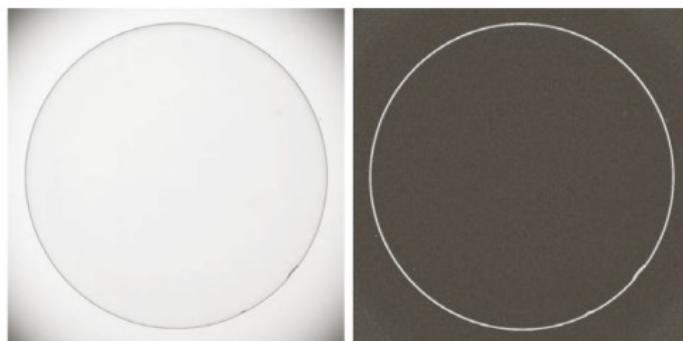
ed infine gli **operatori di Prewitt**, che sono filtri 3x3 che però non includono operazioni per la riduzione del rumore, a differenza degli operatori di Sobel

-1	-1	-1
0	0	0
1	1	1

-1	0	1
-1	0	1
-1	0	1

 .

Facciamo ora un **esempio di applicazione basata su gradiente, utilizzando il filtraggio di Sobel**, che permette di ridurre la visibilità delle regioni in cui l'intensità cambia lentamente, permettendo di evidenziare i dettagli e al contempo rendendo l'individuazione dei difetti più semplice per un'elaborazione automatica



In OpenCV la funzione che implementa il filtro di Sobel è proprio **Sobel()**, che prende in input **immagine di input** e **immagine risultante dell'operazione**, il **tipo di dato dell'immagine di output** (ddepth) e l'**ordine della derivata lungo le direzioni x e y**, quindi se ad es. viene specificato 1 ad entrambe, significa che si dovrà effettuare la derivata lungo entrambe le direzioni , se viene specificato 0 ad una e 1 all'altra allora si dovrà eseguire la derivata solo lungo la direzione che ha valore 1, e così via. Oltre a questi parametri ne **abbiamo degli altri opzionali**, che però solitamente possiamo non specificare, lasciando i valori di default.

```
void cv::Sobel(
    cv::InputArray src,           // Input image
    cv::OutputArray dst,          // Result image
    int ddepth,                  // Pixel depth of output (e.g., CV_8U)
    int xorder,                  // order of corresponding derivative in x
    int yorder,                  // order of corresponding derivative in y
    cv::Size ksize = 3,           // Kernel size
    double scale = 1,             // Scale (applied before assignment)
    double delta = 0,             // Offset (applied before assignment)
    int borderType = cv::BORDER_DEFAULT // Border extrapolation
);
```

Solitamente viene utilizzata una combinazione delle varie tecniche per migliorare un'immagine e non una sola tecnica.

Colore (Lez.5)

Il colore è un **potente descrittore** che permette di **identificare ed estrarre gli oggetti/dettagli dalla scena**.

L'elaborazione delle immagini a colori si divide in due classi:

- **full-color**, quando i colori vengono acquisiti da un sensore full-color, ossia un dispositivo, come ad es. una fotocamera;
- **falsi colori**, nel caso in cui, con immagini non acquisite a colori, venga fatto corrispondere ad un determinato valore di intensità un particolare colore.

Tenendo conto che la **luce visibile** è composta da una banda di frequenze relativamente stretta, il motivo per cui noi percepiamo i colori è che questi (colori), colpendo gli oggetti, vengono riflessi. Ad esempio un corpo che riflette la luce in tutte le lunghezze d'onda apparirà bianco.

Caratterizzazione del colore

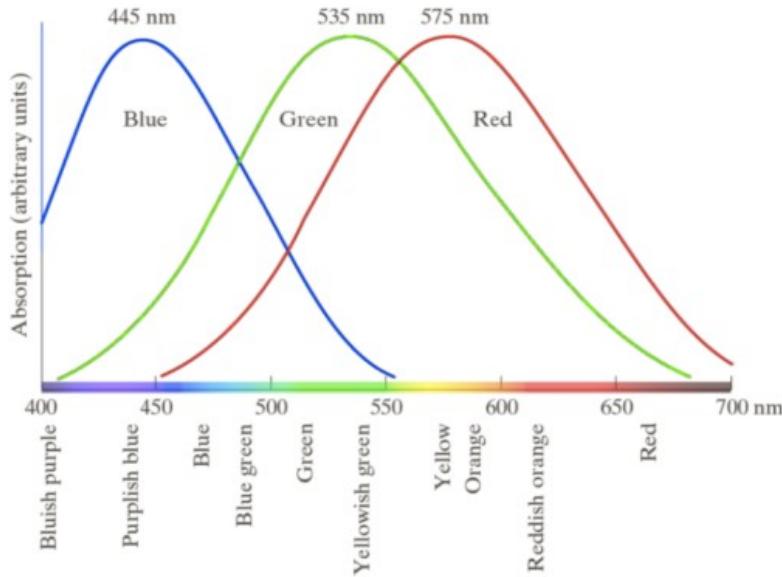
La luce si divide in:

- **acromatica**, se ha un **unico attributo**, ossia l'**intensità**, o livello di grigio;
- **cromatica**, che è descritta da **tre quantità**:
 - o **radianza**, ossia l'energia totale emessa dalla fonte di luce (si misura in watt - W);
 - o **luminanza**, cioè l'energia percepita dall'osservatore a partire da quella emessa dalla fonte di luce (si misura in lumen - lm);
 - o **luminosità**, che invece è una percezione soggettiva e di conseguenza non misurabile.

I colori vengono percepiti attraverso i **coni**, divisi in **tre categorie percettive**:

- il 65% è sensibile alla **luce rossa**;
- il 33% è sensibile alla **luce verde**;
- solo il 2% è sensibile alla **luce blu**, ma bisogna tenere conto che i coni appartenenti a questa categoria sono anche **i più sensibili**.

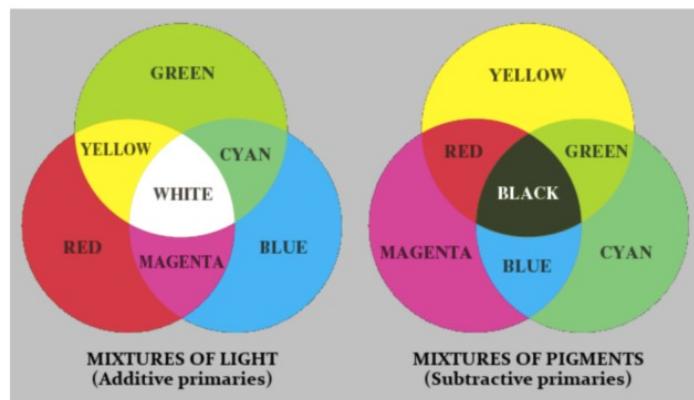
Le caratteristiche dell'occhio consentono di vedere i colori come combinazioni variabili dei **colori primari RGB (Red – Green – Blue)**.



Come si può notare anche dall'immagine soprastante, esistono delle aree in cui i colori primari si sovrappongono, producendo i **colori secondari**, ossia:

- **magenta**, dato dalla combinazione di rosso e blu;
- **ciano**, dato dalla combinazione di verde e blu;
- **giallo**, dato dalla combinazione di rosso e verde.

I **colori secondari corrispondono ai colori primari dei pigmenti**, ossia quelli che assorbono/riflettono i colori primari. Al contempo, i **colori secondari dei pigmenti corrispondono ai colori primari della luce**. In pratica con i pigmenti si ha una situazione opposta rispetto a quella che si ha con la luce.



Per distinguere un colore da un altro si utilizzano **tre caratteristiche**:

- **luminosità**, che misura l'**intensità**;
- **tonalità (hue)**, associata alla lunghezza d'onda dominante, quindi descrive il **colore puro**;
- **saturazione**, che misura la purezza della tonalità, ossia la **quantità di bianco** mescolato alla tonalità.

L'insieme di tonalità e saturazione viene detto **cromaticità**.

Le quantità di rosso, verde e blu necessarie per formare un colore sono dette **valori tristimolo** (X, Y e Z). Un colore, quindi, viene specificato mediante i **coefficienti tricromatici**

$$x = \frac{X}{X+Y+Z} \quad y = \frac{Y}{X+Y+Z} \quad z = \frac{Z}{X+Y+Z} \quad \rightarrow \quad x + y + z = 1$$

Nell'esempio soprastante possiamo vedere anche che è possibile normalizzare i valori tristimolo in un intervallo [0,1], in modo che la loro somma sarà 1.

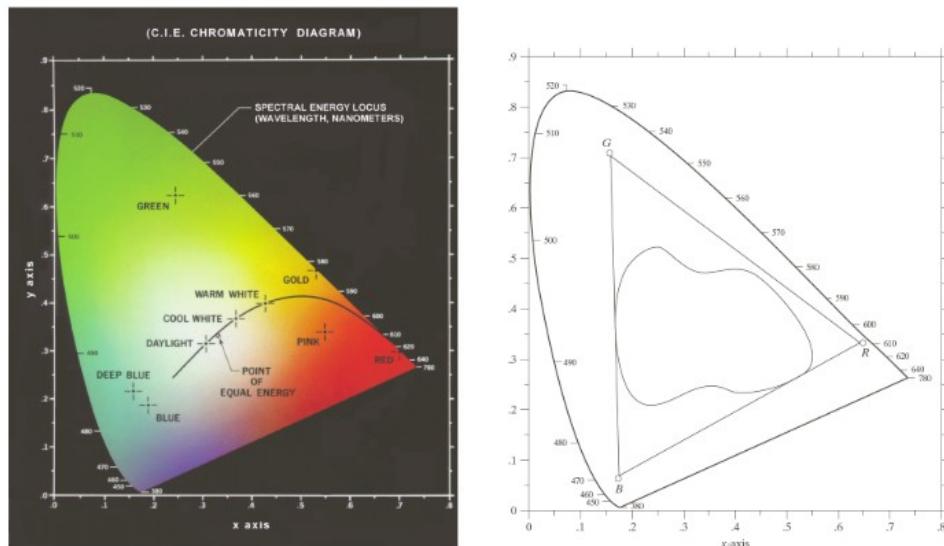
Diagramma di cromaticità

Un altro modo per specificare il colore è mediante il diagramma di cromaticità, che **mostra la composizione del colore in funzione di x (rosso) e y (verde)**.

In funzione della relazione vista nel paragrafo precedente, $x+y+z=1$, è poi possibile ricavare il valore z (blu).

Il punto in cui le tre componenti assumono valore uguale viene detto **punto di uguale energia**, che corrisponde al colore **bianco**.

I punti sul bordo del diagramma sono completamente **saturi**, ossia **senza bianco**, e man mano che ci si avvicina al punto di uguale energia diminuisce la **saturazione**, fino ad arrivare a **0** nel punto di uguale energia.



Modello (spazio) colore

Un modello colore, o spazio colore, è **un sistema di coordinate, ed un sottospazio di quel sistema, in cui ogni colore è rappresentato da un punto**.

I modelli colore più utilizzati sono **RGB**, per monitor e telecamere, **CMY** (Cyan Magenta Yellow) e **CMYK** (Cyan Magenta Yellow e un quarto colore nero, K), per stampanti, e **HSI** (Hue Saturation Intensity), che è più simile al modo in cui gli

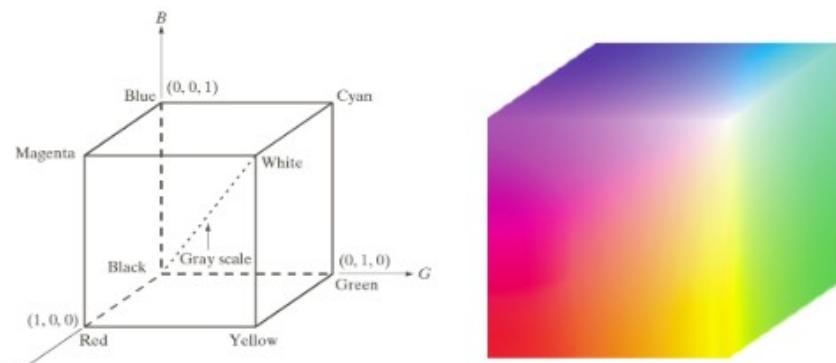
uomini descrivono i colori, essendo basato sulla percezione di un colore in termini di luminosità, tonalità e saturazione.

Un **vantaggio del modello HSI** molto utile per l'elaborazione delle immagini è quello di **decorrelare le informazioni dei colori dall'intensità**, ossia tonalità e saturazione sono indipendenti dalla luminosità.

Modello RGB (Red Green Blue)

Nel modello RGB **ogni colore è rappresentato dalle sue componenti primarie**: rosso, verde e blu.

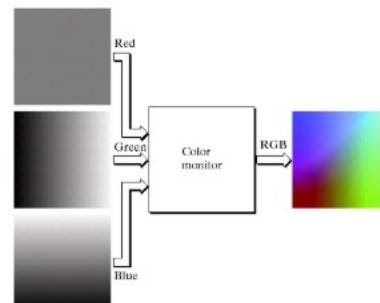
Il modello si basa su un **sistema di coordinate cartesiane**, ed il **sottospazio di interesse è un cubo**, ai cui vertici troviamo i colori primari, quelli secondari, il bianco ed il nero.



Nell'asse tra nero e bianco, come possiamo notare, **abbiamo la scala di grigio**.

Le immagini rappresentate nel modello RGB sono formate da 3 immagini, una per ogni colore primario. Solitamente ognuna delle immagini è rappresentata su 8 bit e, di conseguenza, la profondità del pixel RGB è 24 bit, data dal prodotto tra le 3 immagini e il numero di bit di ognuna, ossia 8.

165	187	209	58	7
14	125	233	201	90
253	144	120	251	41
67	100	32	241	23
209	118	124	27	59
210	236	105	159	19
35	178	199	187	4
115	104	34	111	19
32	89	231	203	74



Modello CMY-CMYK (Cyan Magenta Yellow)

Come detto in precedenza, ciano, magenta e giallo sono sia colori secondari di luce che colori primari di pigmenti.

Le **conversioni da RGB a CMY** e viceversa sono date dalle seguenti relazioni, che discendono dal fatto che i colori primari della luce sono complementari dei primari dei pigmenti, e viceversa.

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} C \\ M \\ Y \end{bmatrix}$$

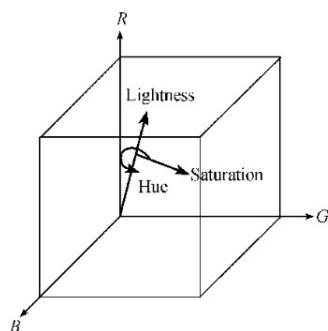
Visto che uguali quantità dei pigmenti primari non producono il nero puro, si aggiunge a questo modello anche un quarto colore, appunto **il nero (K)**.

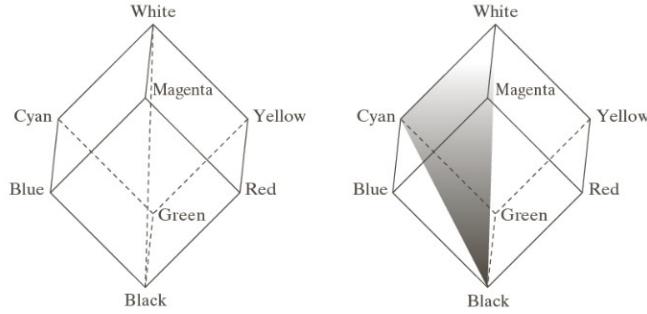
Modello HSI (Hue Saturation Intensity)

Nel modello HSI **ogni colore è descritto mediante le caratteristiche di luminosità, tonalità e saturazione** viste in precedenza.

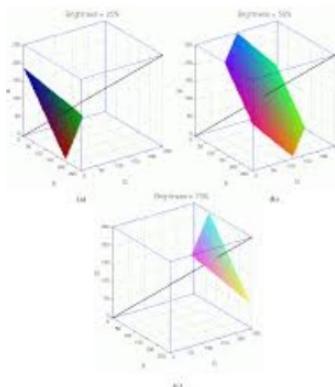
Per determinare la **componente di intensità di un punto colore RGB**, mediante il modello HSI, **bisogna far passare un piano perpendicolare all'asse di intensità che contenga il punto colore**. Il punto di intersezione tra l'asse di intensità ed il piano darà quindi il **valore di intensità**, con la **saturazione** che, invece, è **data dalla distanza del punto colore dall'asse di intensità** (ricordiamo che l'intensità sarebbe la scala di grigio).

Unendo il bianco, il nero ed un punto colore, tutti i punti del triangolo ottenuto sono caratterizzati dalla **stessa tonalità**. Quest'ultima dipende dall'**angolo rispetto ad un punto** (solitamente la tonalità 0 corrisponde ad un angolo di 0° rispetto al rosso, con il verde che si troverà a 120° ed il blu a 240°), con il valore che **aumenta in senso orario**.

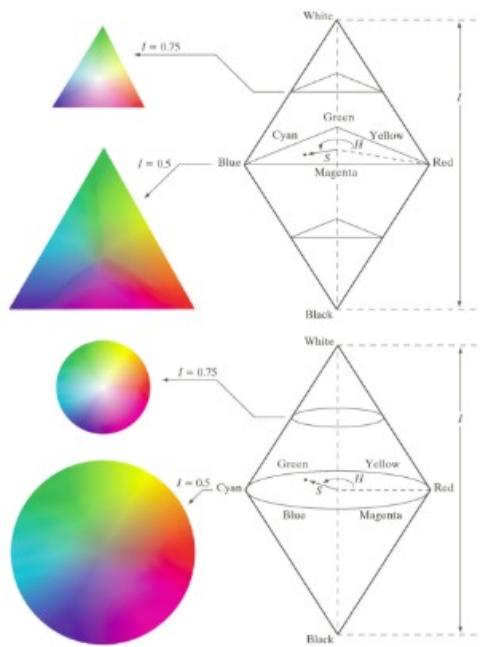
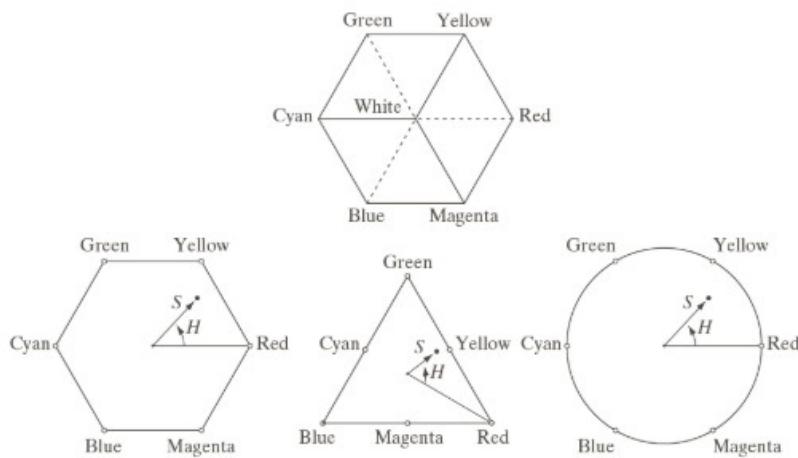




Lo spazio HSI è quindi rappresentato da un'asse di intensità verticale e dal luogo dei punti colore che giacciono sui piani perpendicolari all'asse. Visto che i piani si muovono lungo l'asse di intensità, l'intersezione con le facce del cubo ha forma triangolare o esagonale.



HSI

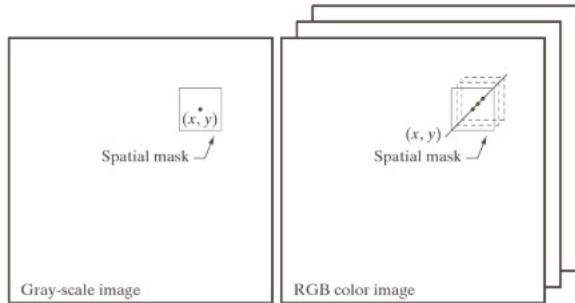


Elaborazioni full-color

Esistono due metodi di elaborazione delle immagini full-color:

- ogni componente viene elaborata separatamente e combinata con le altre;

- vengono elaborati i pixel colore, quindi tutte e tre le componenti sono elaborate insieme.



Smoothing RGB e HSI

Lo smoothing di un'immagine in scala di grigio si può realizzare con un'operazione di filtraggio spaziale ed una maschera opportuna.

In particolare, il valore di ogni pixel viene sostituito con la media dei valori dei pixel nell'intorno definito dalla maschera.

$$\bar{c}(x, y) = \frac{1}{K} \sum_{(x, y) \in S_{xy}} c(x, y)$$

Utilizzando il modello RGB si dovrà effettuare uno **smoothing per ognuno dei colori primari, per poi sommare le tre matrici prodotte, componente per componente**.

$$\bar{c}(x, y) = \begin{bmatrix} \frac{1}{K} \sum_{(x, y) \in S_{xy}} R(x, y) \\ \frac{1}{K} \sum_{(x, y) \in S_{xy}} G(x, y) \\ \frac{1}{K} \sum_{(x, y) \in S_{xy}} B(x, y) \end{bmatrix}$$

Se invece si utilizza la rappresentazione HSI, è possibile eseguire lo **smoothing solo sulla componente intensità**, lasciando inalterate saturazione e tonalità. Così facendo preservo la componente cromatica (insieme di saturazione + tonalità) smussando i picchi di intensità.



Sharpening RGB e HSI

Come sappiamo, lo sharpening si realizza mediante l'utilizzo dell'operatore laplaciano.

Il valore del laplaciano di un vettore è definito come un vettore le cui componenti sono uguali al valore laplaciano delle singole componenti del vettore di input.

$$g(x,y) = f(x,y) \pm \nabla^2 f(x,y)$$

Anche in questo caso, quindi, **bisogna applicare il laplaciano ad ognuno dei colori primari, per poi combinare le tre matrici prodotte, componente per componente.**

$$\nabla^2 [\bar{c}(x,y)] = \begin{bmatrix} \nabla^2 R(x,y) \\ \nabla^2 G(x,y) \\ \nabla^2 B(x,y) \end{bmatrix}$$

Anche nel caso dello sharpening, se si utilizza la rappresentazione **HSI**, è possibile eseguire l'operazione solo sulla componente intensità, lasciando inalterate la tonalità e la saturazione.



Funzione imread per immagini a colori e accesso ai canali

Abbiamo già visto il funzionamento di [imread\(\)](#).

Il secondo argomento della suddetta funzione ha come **valore di default IMREAD_COLOR**, che permette di caricare l'immagine in una matrice (Mat) a 3 canali, con una profondità di 8 bit per canale. **Se l'immagine in input è a colori**, viene allocata una matrice con 3 canali in formato BGR (al contrario, per convenzione), mentre **se l'immagine è in scala di grigio**, viene allocata una matrice in cui il livello di grigio viene replicato in tutti i canali.

Nel caso in cui si passi **IMREAD_ANY_COLOR** come secondo argomento, le immagini vengono caricate in matrici con il corretto numero di canali.

L'**accesso ai canali** avviene utilizzando una struttura [Vec3b](#), in quanto ogni elemento sarà dato dal valore di blu, verde e rosso (Vec3b sta per vettore di 3 byte, con un 1 byte, che equivale ad 8 bit, per ogni canale).

```

        for(int i=0;i<image.rows;i++)
            for(int j=0;j<image.cols;j++)
                image.at<Vec3b>(i,j)[0];    //B
                image.at<Vec3b>(i,j)[1];    //G
                image.at<Vec3b>(i,j)[2];    //R

```

Funzione cvtColor

La funzione cvtColor() serve per **modificare lo spazio colore con cui è rappresentata un'immagine, mantenendo lo stesso tipo.**

La funzione prende come argomenti **due matrici**, quella relativa all'immagine di input e in cui memorizzeremo il risultato dell'operazione, una variabile che conterrà una delle macro che determina il **tipo di conversione** e, come ultimo argomento, il **numero di canali dell'immagine di output**. Se non viene inserito nessun valore per quest'ultimo parametro, il numero di canali verrà calcolato automaticamente.

```

void cv::cvtColor(
    cv::InputArray src,           // Input array
    cv::OutputArray dst,          // Result array
    int code,                   // color mapping code
    int dstCn = 0                // channels in output (0='automatic')
);

```

Di seguito i **diversi codici da inserire come tipo di conversione**:

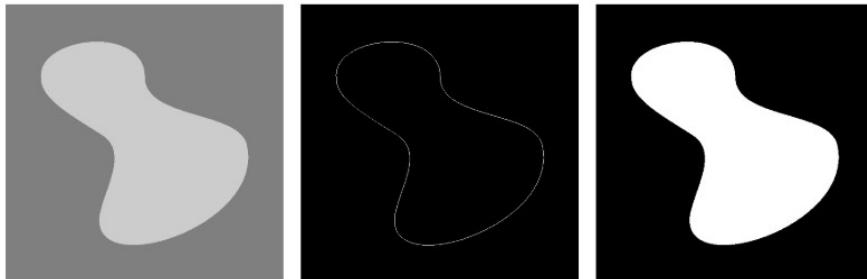
Conversion code	Meaning
cv::COLOR_BGR2RGB cv::COLOR_RGB2BGR	Convert between RGB and BGR color spaces
cv::COLOR_RGB2GRAY cv::COLOR_BGR2GRAY	Convert RGB or BGR color spaces to grayscale
cv::COLOR_GRAY2RGB cv::COLOR_GRAY2BGR	Convert grayscale to RGB or BGR color spaces (optionally removing alpha channel in the process)
cv::COLOR_RGB2XYZ cv::COLOR_BGR2XYZ	Convert RGB or BGR image to CIE XYZ representation or vice versa (Rec 709 with D65 white point)
cv::COLOR_RGB2HSV cv::COLOR_BGR2HSV cv::COLOR_HSV2RGB cv::COLOR_HSV2BGR	Convert RGB or BGR image to HSV (hue saturation value) color representation or vice versa
cv::COLOR_RGB2HLS cv::COLOR_BGR2HLS cv::COLOR_HLS2RGB cv::COLOR_HLS2BGR	Convert RGB or BGR image to HLS (hue lightness saturation) color representation or vice versa

Segmentazione (Lez.6)

L'obiettivo della segmentazione è quello di **suddividere un'immagine nelle regioni** (o negli oggetti) **che la compongono**, con la qualità della stessa (della segmentazione) che può influire sull'esito delle elaborazioni successive.

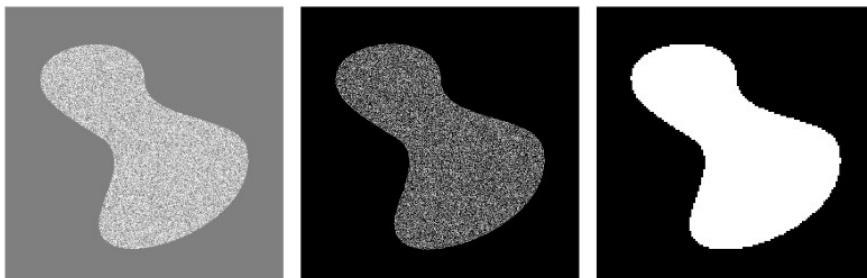
La maggior parte degli algoritmi di segmentazione si basa su una delle due proprietà di base dei valori di intensità, ossia **discontinuità e similarità**.

Se si sfruttano le discontinuità (segmentazione edge based), la segmentazione sarà guidata da bruschi cambiamenti di intensità (come ad es. gli edge, che si ottengono unendo tutti i punti in cui è avvenuta la discontinuità). In pratica, in questa tipologia di algoritmi, si assume che i bordi siano sufficientemente diversi tra le regioni e dallo sfondo, in modo da poter sfruttare le intensità locali.



Esempi di algoritmi di questo tipo sono **Canny, Harris e Hough**.

Se si sfruttano le similarità (segmentazione region based), invece, si partiziona l'immagine in regioni al cui interno saranno contenuti pixel tra loro simili, in base ad un criterio di similarità.



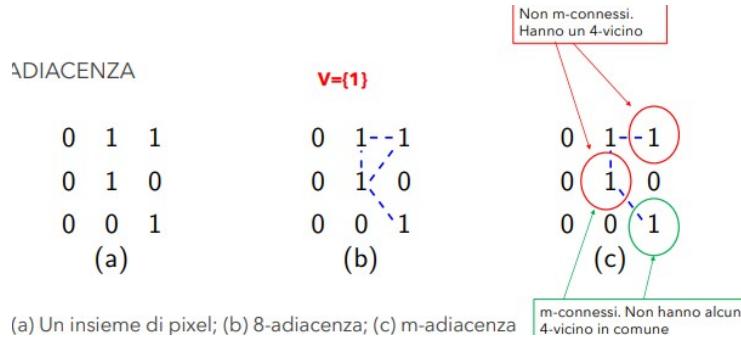
Esempi di algoritmi di questo tipo sono **sogliatura, region growing, split and merge e clustering**.

Adiacenza, connettività e regione

L'adiacenza tra due pixel dipende, come prima cosa, dalla loro intensità, che deve essere compresa nelle intensità contenute in un insieme V.

Considerando due pixel p e q che hanno valori di intensità in V, questi sono:

- **4-adiacenti** se q è 4-vicino di p;
- **8-adiacenti** se q è 8-vicino di p;
- **m-adiacenti**, con la m che sta per **mista** (quindi adiacenza mista), se q è 4-vicino di p oppure se è vicino diagonale di p con nessun 4-vicino in comune.



Un **percorso** tra due pixel è una sequenza di pixel adiacenti tra i due (pixel).

Considerando un sottoinsieme S di pixel dell'immagine, due pixel si dicono **connessi** in S (o anche in una regione) se esiste un percorso tra loro formato interamente da pixel appartenenti a S .

Per ciascun pixel di S , l'insieme dei pixel ad esso connessi forma una **componente connessa** e, se in questo sottoinsieme ne esiste solo una (di componente connessa), allora S è una **regione**.

Considerando R la regione occupata dall'immagine, **la segmentazione consiste nel partizionare R in n sottoregioni tali che:**

- la loro unione è uguale all'intera regione R occupata dall'immagine (questo prende il nome di segmentazione completa);
- ogni sottoregione R_i è un insieme connesso, ossia esiste un percorso tra ogni coppia di pixel della sottoregione formato interamente da pixel appartenenti a R_i ;
- le regioni sono disgiunte, ossia ogni pixel appartiene ad una sola regione;
- tutti i pixel appartenenti ad una regione devono rispettare un determinato predicato;
- applicato un predicato a due regioni diverse, questo deve risultare falso, poiché il contrario implicherebbe che le due regioni in realtà sarebbero un'unica regione.

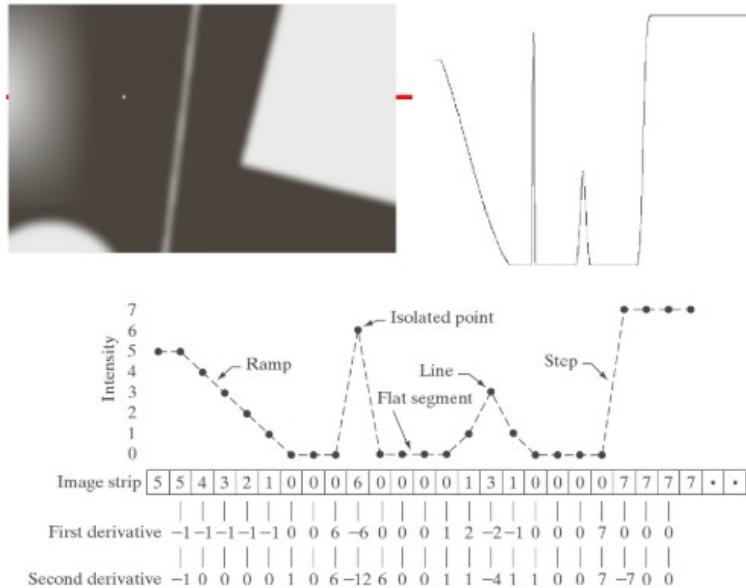
Individuazione delle caratteristiche di un'immagine

In un'immagine siamo interessati a **3 caratteristiche**:

- **edge**, ossia insiemi di pixel in cui si presenta una repentina variazione di intensità;
- **linee**, cioè segmenti di edge in cui l'intensità ai lati della linea è minore o maggiore dell'intensità dei pixel della linea;
- **punti**, ossia linee di lunghezza e larghezza pari a 1 pixel.

Per individuare le variazioni di intensità a cui siamo interessati, si utilizzano le **approssimazioni delle derivate prime** e **seconde**, definite in termini di differenze.

Di seguito troviamo un'immagine che riassume le varie caratteristiche che abbiamo visto e che vedremo in questo capitolo. In particolare, a partire dall'immagine originale, è stata presa una **scan line**, ossia una linea (quella rossa) che passa per l'immagine e la “scansiona”.



Proprietà delle derivate

Le derivate prima e seconda hanno diverse proprietà:

- in presenza di edge a rampa, la derivata prima produce edge spessi, a differenza della derivata seconda che ne produce di sottili (edge);
- la risposta della derivata seconda in presenza di punti isolati è più forte rispetto a quella della derivata prima;
- sia sugli edge a rampa che su quelli a gradino (step), la derivata seconda ha segni opposti, e questo può essere utilizzato per determinare se un edge rappresenta una transizione da chiaro a scuro o viceversa.

Punti isolati

Per individuare i punti isolati, dopo aver applicato il filtro [Laplaciano](#), si utilizza una soglia sulla risposta del filtro, in modo da determinare i punti di discontinuità:

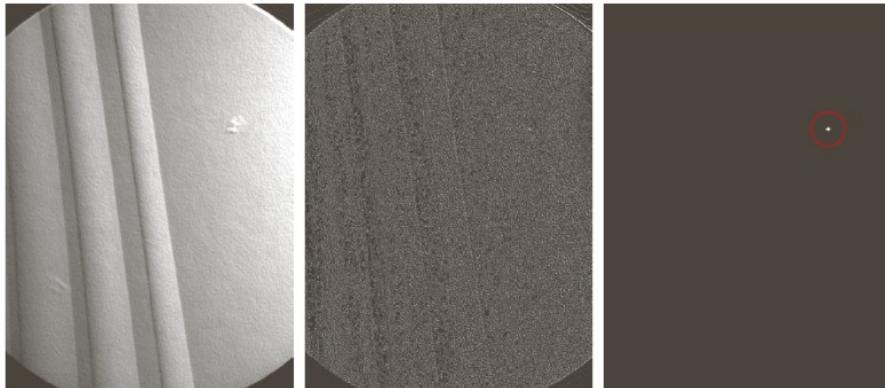
$$g(x, y) = \begin{cases} 1 & \text{se } |R(x, y)| \geq T \\ 0 & \text{altrimenti} \end{cases},$$

con **g** l'immagine di output, **T** una **soglia non negativa** e **R** la **risposta del filtro**. Quest'ultima viene presa in modulo poichè può generare valori sia positivi che negativi in base alla direzione della variazione di intensità; di conseguenza

prendere il valore assoluto garantisce che stiamo considerando l'intensità indipendentemente dalla direzione.

L'intensità di un punto isolato sarà abbastanza diversa da quella (intensità) dei suoi 8 vicini ed infatti l'idea è appunto quella di calcolare la differenza di intensità tra i punti vicini tramite il parametro T.

Di seguito abbiamo un'immagine in cui viene individuato un punto isolato, che risulta visibile poichè avrà valore maggiore della soglia.

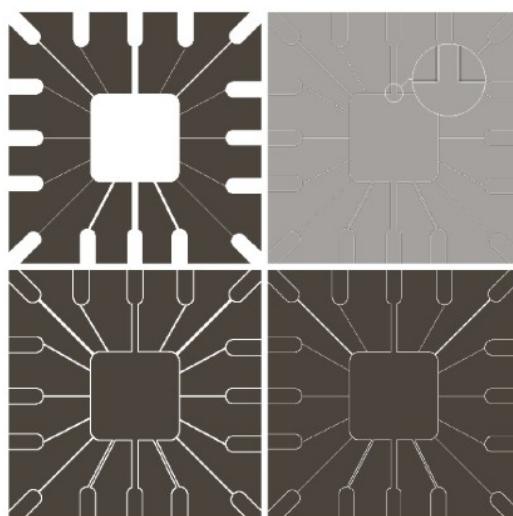


Linee

Anche per individuare le linee è possibile utilizzare il Laplaciano, ma bisogna gestire la doppia risposta della derivata seconda. In questo caso è possibile utilizzare il valore assoluto della risposta, oppure solo i valori positivi, eventualmente sogliati per attenuare l'effetto del rumore.

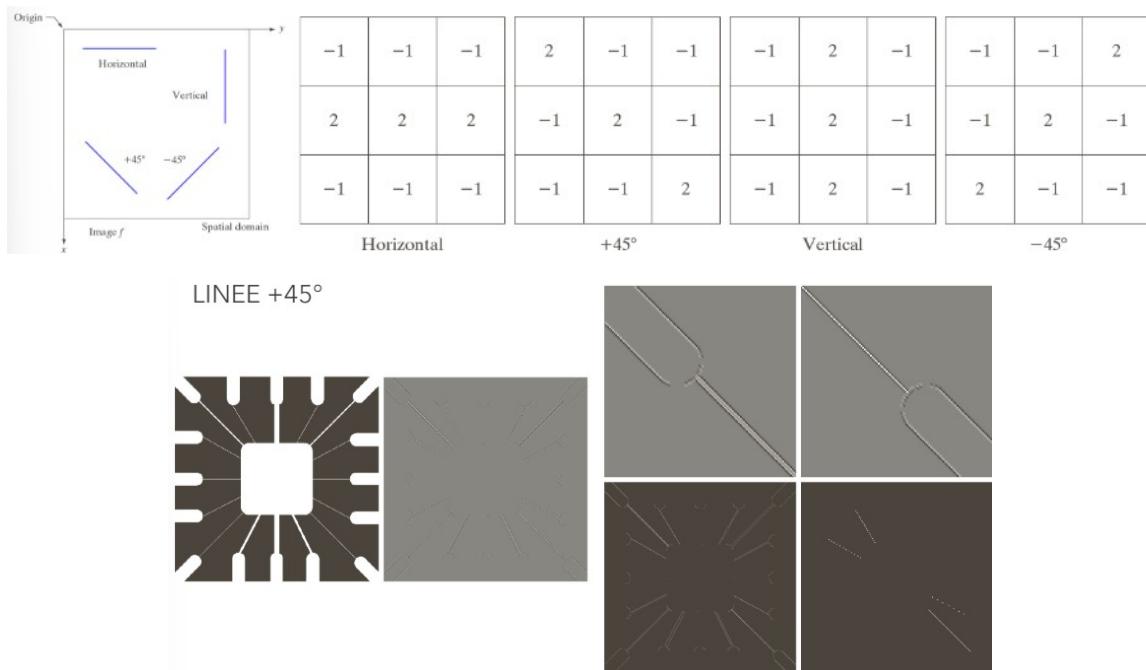
Scegliendo la prima soluzione, però, non sapremo dove passa l'edge, e di conseguenza otteniamo linee più spesse, al contrario del secondo caso, nel quale si ottengono linee più sottili, ma si vanno a "tagliare" tutti i valori negativi.

Nel caso in cui le linee siano più ampie rispetto alla dimensione del filtro, si otterrà l'effetto di una "valle" di valori nulli che separa le due linee.



Come sappiamo, il filtro Laplaciano è isotropico, ossia la risposta è indipendente dalla direzione. Di conseguenza, **se volessimo individuare rette con direzioni specifiche**, è possibile utilizzare **filtri specifici**.

Come possiamo osservare dall'immagine seguente, in base al tipo di filtro verranno individuate alcune rette piuttosto di altre.

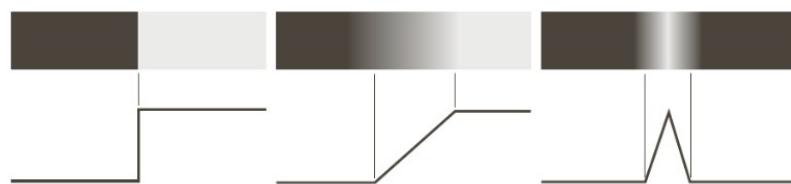


Modelli di edge

I modelli di edge vengono **classificati in base ai profili di intensità**:

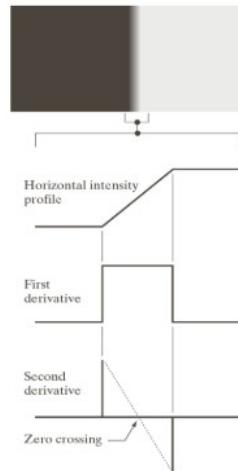
- **edge a gradino** (step), che consiste in una transizione tra due livelli di intensità ad una distanza ideale di 1 pixel;
- **edge a rampa** (ramp), ossia edge sfocati e rumorosi che appaiono come una transizione graduale e non netta (come nel caso dell'edge a gradino insomma). Gli edge a rampa difficilmente corrispondono a linee sottili;
- **roof edge**, che sono edge associati al bordo di una regione e hanno una base determinata dallo spessore e dalla sfocatura della linea.

Spesso nelle immagini si trovano più tipi di edge con profili diversi da quelli idealì.



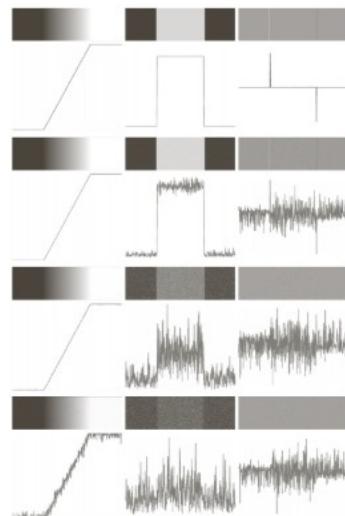
Per individuare gli **edge a rampa** è possibile utilizzare la derivata prima, mentre il segno della derivata seconda può essere utilizzato per determinare se un pixel si trova sul lato scuro o chiaro di un edge, visto che la derivata seconda produce due valori per ogni edge.

Inoltre, sempre poichè la derivata seconda produce due valori per ogni edge, si può utilizzare lo **zero-crossing** (attraversamento dello 0) per trovare il centro di un edge.

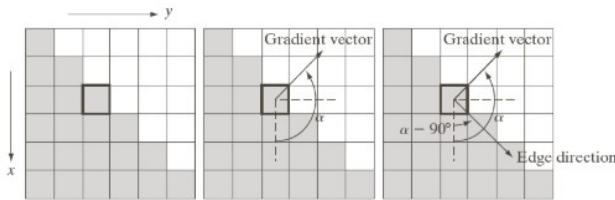


Sebbene la derivata seconda dia una risposta più forte rispetto a quella (la risposta) della derivata prima, ad una risposta più forte coincide un'esaltazione del rumore. Di conseguenza, **prima di individuare i punti di edge**, selezionando solo quelli (i punti di edge) che fanno parte di un edge, **si può applicare lo smoothing per ridurre il rumore**.

Nella seguente immagine abbiamo, oltre al caso ideale, degli esempi di individuazione di edge a rampa con rumore gaussiano, rispettivamente con media nulla e deviazione standard pari a 0.1, 1.0 e 10.0. Possiamo notare come negli ultimi due casi si abbiano dei picchi della derivata seconda superiori ai bordi veri e propri, il che porterebbe alla visualizzazione di bordi non veri una volta applicata la sogliatura.



Per individuare gli edge è anche possibile utilizzare il gradiente. Questo perchè il gradiente punta sempre nella direzione di massima variazione, che quindi sarà ortogonale alla direzione dell'edge.



La direzione del gradiente è data dall'arcotangente del rapporto delle sue due componenti:

$$\alpha(x, y) = \tan^{-1} \left[\frac{g_y}{g_x} \right].$$

Per ricavare le componenti del gradiente, ossia le derivate parziali, esistono, come sappiamo, degli operatori di derivazione (Sobel, Prewitt, Roberts, ecc.).

Di seguito abbiamo un'immagine seguita dalla sua scomposizione nelle componenti del gradiente, rispettivamente lungo l'asse x e lungo l'asse y.

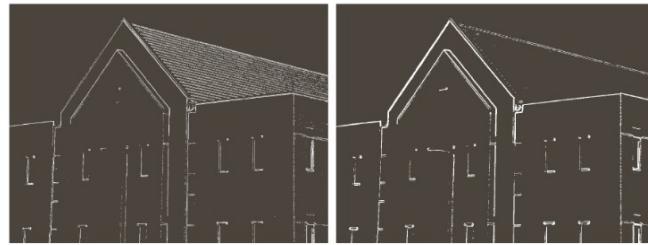


Anche la risoluzione dell'immagine può determinare l'esito dell'operazione di estrazione degli edge, in quanto dettagli fini sono assimilati al rumore, producendo molti edge. Di conseguenza è buona norma, prima di applicare un estrattore di edge, effettuare uno smoothing dell'immagine.



Per enfatizzare gli **edge diagonali** (prima immagine sotto) è necessario utilizzare **maschere specifiche**, mentre per ottenere **immagini gradiente più pulite** è possibile utilizzare la **tecnica del thresholding** (seconda immagine sotto), in modo da utilizzare solo i pixel la cui risposta superi una determinata soglia.



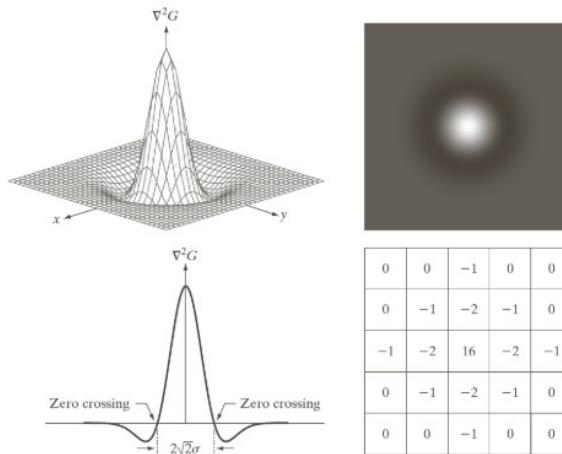


Algoritmo di Marr-Hildreth e LoG

Marr e Hildreth proposero il cosiddetto filtro LoG (**Laplacian of Gaussian**), anche detto **operatore a sombrero** per via della forma della Gaussiana, che equivale al Laplaciano della funzione Gaussiana, ossia la **somma delle derivate seconde parziali della Gaussiana rispetto a x e y** (la seconda formula delle due seguenti non serve saperla per forza). Come possiamo vedere dalle seguenti immagini, questo filtro ha un termine positivo centrale circondato da una regione negativa, i cui valori aumentano in funzione della distanza dall'origine, e una regione esterna nulla.

$$\nabla^2 G(x, y) = \frac{\partial^2 G(x, y)}{\partial^2 x^2} + \frac{\partial^2 G(x, y)}{\partial^2 y^2}$$

$$\nabla^2 G(x, y) = \left[\frac{x^2 + y^2 + 2\sigma^2}{\sigma^4} \right] e^{-\frac{x^2+y^2}{2\sigma^2}}$$



E' possibile modificare l'ampiezza della funzione Gaussiana modificando il valore σ , considerando che **più è piccolo questo valore e minore sarà l'effetto di smoothing**.

A livello di codice, invece, bisogna tenere conto del fatto che la dimensione n del filtro deve essere scelta pari al più piccolo intero dispari maggiore o uguale a 6σ (ad es se $\sigma=4$ allora $n = 25$).

In OpenCV è possibile utilizzare la funzione **getGaussianKernel** per ottenere, appunto, **un filtro Gaussiano** delle dimensioni (ksize) e dell'ampiezza (sigma) richieste.

```

cv::Mat cv::getGaussianKernel(
    int         ksize,           // Kernel size
    double      sigma,          // Gaussian half-width
    int         ktype = CV_32F   // Type for filter coefficients
);

```

Dopodichè, per applicarlo, possiamo utilizzare la funzione filter2D, in modo da eseguire la convoluzione.

```
filter2D(src, dst, CV_32F, getGaussianKernel(n, sigma));
```

L'algoritmo di Marr-Hildreth consiste nella convoluzione del filtro LoG con l'immagine di input

$$g(x,y) = [\nabla^2 G(x,y)] * f(x,y),$$

che equivale a filtrare l'immagine con un filtro Gaussiano di dimensione NxN e poi applicare il filtro Laplaciano all'immagine filtrata, quindi

$$g(x,y) = \nabla^2 [G(x,y) * f(x,y)].$$

Infine bisogna individuare gli zero-crossing, ossia i punti in cui avviene la transizione tra positivo e negativo, che equivale alla ricerca di **$g(x,y) = 0$** .

Considerando un intorno 3x3 centrato in un pixel p dell'immagine filtrata, **uno zero-crossing nel pixel p implica che i segni di almeno due pixel vicini opposti siano diversi**, tenendo conto del fatto che con opposti si intende i pixel sopra-sotto, destra-sinistra e le due diagonali. **Il valore assoluto della differenza tra questi due pixel vicini opposti, inoltre, deve superare una certa soglia**.

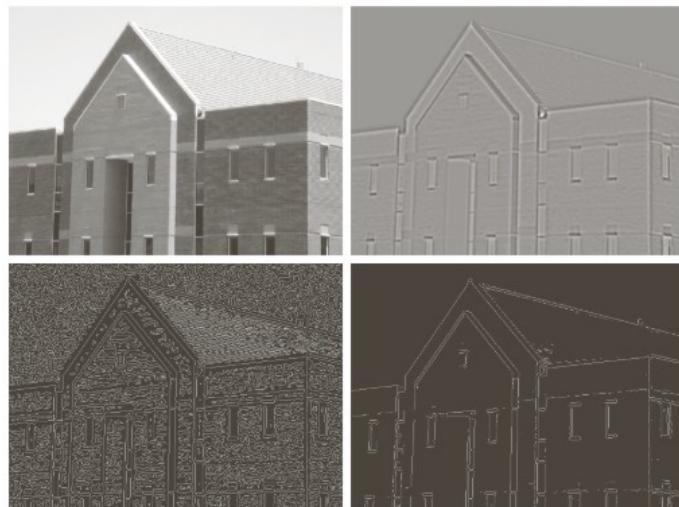
Di seguito abbiamo un possibile **pseudocodice** per quanto riguarda **l'individuazione dei punti di zero-crossing**. Come prima cosa si calcola il valore soglia come il 40% del valore massimo di intensità all'interno dell'immagine filtrata con il LoG. Dopodichè si estrae l'intorno 3x3 centrato in ogni possibile pixel (i,j), calcolando i valori con intensità minima e massima contenuti in tale intorno. Se il valore di intensità del pixel (i,j) che stiamo considerando come centro dell'intorno 3x3 è maggiore di 0, allora si imposta una variabile flag a true se il valore di intensità minima dell'intorno è minore di 0, mentre se il valore di intensità del pixel (i,j) è minore di 0, allora si imposta la variabile flag a true se il valore di intensità massimo dell'intorno è maggiore di 0. Come ultimo passo, si controlla se la differenza tra i valori di intensità massimo e minimo sia maggiore della soglia calcolata inizialmente e se la variabile flag è stata messa a true, poichè in questo caso si può affermare che ci troviamo in un punto di zero-crossing, che segnaliamo mettendo ad 1 il pixel corrispondente nell'immagine di output.

```

th= max(in)*.4
for i=1:in.rows
    for j=1:in.cols
        N = intorno 3x3
        m = min(N)
        M = max(N)
        if(in(i,j)>0)
            flag=m<0?true:false
        else
            flag=M>0?true:false
        if(max-min>th && flag=true)
            out=1

```

Il LoG, inoltre, può anche essere approssimato come differenza di Gaussiane (DoG).



In definitiva possiamo quindi affermare che lo **scopo dell'algoritmo di Marr-Hildreth** è quello di rilevare i bordi di un'immagine, combinando il concetto di filtro LoG con la ricerca dei punti di zero-crossing.

Canny e Harris (Lez.7)

Algoritmo di Canny (edge detector)

L'algoritmo di Canny è uno dei migliori algoritmi per l'**individuazione degli edge** e si pone **tre obiettivi**:

- **basso tasso di errore**, ossia ridurre al minimo la probabilità di **falsi positivi** e **falsi negativi** (falsi positivi = pixel ritenuti di edge ma che in realtà non lo sono);
- **punti di edge ben localizzati**, cioè gli edge rilevati devono essere il più vicino possibile agli edge reali, quindi la distanza tra il punto rilevato e quello reale deve essere minima;

- **risposta puntuale per edge singolo**, ossia deve restituire un solo punto per ogni punto di edge.

La caratteristica di questo algoritmo è la **rigida formulazione matematica**.

Canny dimostrò che una **buona approssimazione per l'individuazione di edge a gradino è la derivata prima della Gaussiana**.

$$\frac{d}{dx} e^{-\frac{x^2}{2\sigma^2}} = \frac{-x}{\sigma^2} e^{-\frac{x^2}{2\sigma^2}}$$

Poichè la direzione dell'edge non è conosciuta a priori, bisognerebbe applicare l'operatore in tutte le possibili direzioni. Questo processo può essere approssimato effettuando la convoluzione dell'immagine con una Gaussiana 2D e calcolando il gradiente del risultato. La magnitudo e l'orientazione del gradiente possono poi essere utilizzati per calcolare l'intensità e l'orientazione dell'edge.

In pratica, **data un'immagine di input $f(x,y)$ e la funzione Gaussiana $G(x,y)$** , cioè

$$G(x, y) = e^{-\frac{x^2+y^2}{2\sigma^2}},$$

per conoscere direzione e magnitudo dobbiamo appunto effettuare la **convoluzione tra le due** (immagine di input e Gaussiana)

$$f_s(x, y) = G(x, y) * f(x, y)$$

A questo punto **possiamo calcolare le derivate parziali**, in modo da calcolare il **gradiente per conoscerne magnitudo e direzione** (del gradiente).

$$g_x = \frac{\partial f_s}{\partial x} \quad g_y = \frac{\partial f_s}{\partial y}$$

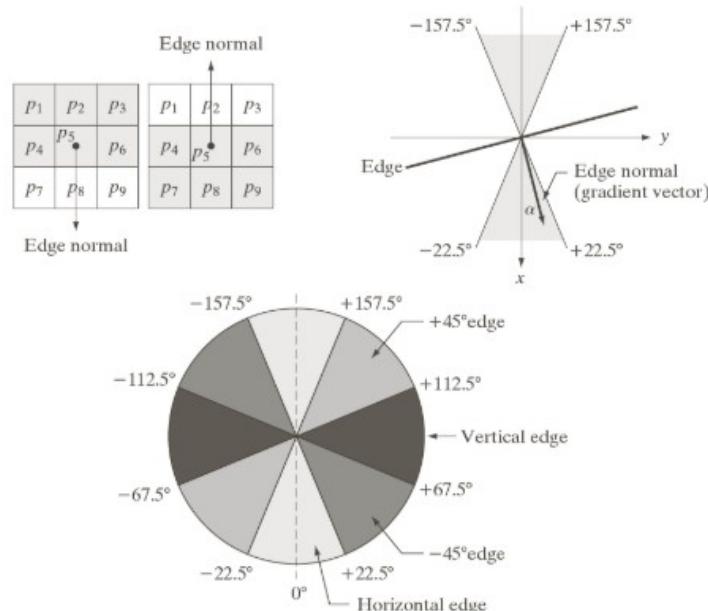
$$M(x, y) = \text{mag}(\nabla f) = \sqrt{g_x^2 + g_y^2}$$

$$\alpha(x, y) = \tan^{-1} \left[\frac{g_y}{g_x} \right]$$

Visto che l'immagine **magnitudo $M(x,y)$** è stata generata utilizzando il gradiente, essa presenterà molti picchi in corrispondenza dei massimi locali. E' poi necessario sopprimere tali valori, ad esempio tramite il metodo della **Non Maxima Suppression (NMS)**, che viene **applicata considerando un intorno 3x3 ed un numero finito di orientazioni** (ad es. 4, ossia orizzontale, verticale, $+45^\circ$, -45°). Considerando il pixel centrale di questo intorno, la NMS verifica se uno dei suoi vicini nella direzione del gradiente abbia magnitudo maggiore della sua, poichè in tal caso sopprimiamo il pixel centrale, altrimenti lo conserviamo.

Poichè è necessario quantizzare tutte le possibili direzioni in 4 settori, bisogna definire degli opportuni intervalli, mentre la direzione dell'edge, come sappiamo, viene ricavata dalla direzione del gradiente (normale all'edge).

Bisogna inoltre tenere conto del fatto che **ogni edge ha due possibili orientazioni**, infatti un edge la cui normale è orientata a 0° o a 180° è comunque considerato un edge orizzontale. Di conseguenza, visto che ogni edge ha 2 possibili orientazioni, per ognuna di esse (orientazioni) possiamo considerare 2 aree opposte che combaciano nel centro della regione. Così facendo, **distinguiamo 4 orientazioni per un totale di 8 aree nella regione**, con **ogni area che avrà un angolo di $360^\circ/8 = 45^\circ$** . Ad esempio, un edge sarà considerato orizzontale se la sua normale è orientata tra -22.5° e 22.5° oppure tra -157.5° e 157.5° (poichè sopra partiamo da 180° , invece che da 0° , essendo nel lato opposto).



Volendo definire la **NMS a livello matematico**, consideriamo d_1, d_2, d_3, d_4 le 4 direzioni orizzontale, verticale, $+45^\circ$ e -45° . Lo schema per un intorno 3×3 prevede di trovare la direzione d_k che sia più vicina a $\alpha(x, y)$, ossia la direzione del gradiente, e se il valore della magnitudo del gradiente $M(x, y)$ è minore di almeno uno dei suoi due vicini lungo d_k , viene soppressa l'intensità del pixel centrale dell'intorno, ossia $g_n(x, y) = 0$, altrimenti l'intensità assumerà il valore della magnitudo, quindi $g_n(x, y) = M(x, y)$.

L'operazione successiva consiste nel sogliare l'immagine $g_n(x, y)$ per ridurre i falsi positivi. Infatti se utilizzassimo una soglia troppo bassa otterremmo falsi positivi, mentre con una soglia troppo alta otterremo falsi negativi.

Per risolvere questo problema l'algoritmo di Canny utilizza un **thresholding con isteresi**, che si serve di **due soglie, una bassa T_L ed una alta T_H** . Questo passo dell'algoritmo permette di suddividere i punti di edge tra quelli che superano la soglia alta, e di conseguenza anche quella bassa, detti **edge "forti"**, quelli che

superano solo la soglia bassa, definiti **edge “deboli”**, e quelli che sicuramente non saranno considerati edge poiché non superano la soglia bassa.

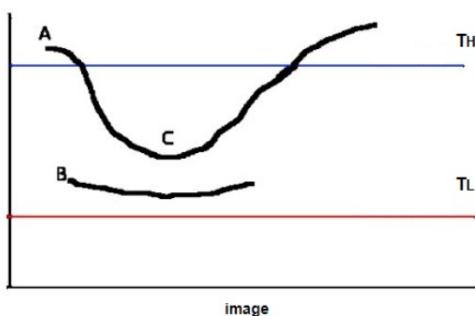
$$g_{NH}(x, y) = g_N(x, y) \geq T_H$$

$$g_{NL}(x, y) = g_N(x, y) \geq T_L$$

Per ricavare i pixel di edge che superano solo la soglia bassa, bisogna sottrarre dall'insieme di pixel di edge “deboli” l'insieme di pixel di edge “forti”.

$$g_{NL}(x, y) = g_{NL}(x, y) - g_{NH}(x, y)$$

Graficamente possiamo vedere questi pixel come quelli contenuti nella cosiddetta **zona di “incertezza”**, ossia quella compresa tra la soglia bassa e la soglia alta.



Per decidere **se questi punti di edge che si trovano nella zona di “incertezza”** debbano essere considerati edge “forti” o meno bisogna controllare se questi **sono connessi ad un edge “forte”**. In tal caso, infatti, i punti di edge “deboli” **verranno promossi ad edge “forti”**, come nel caso dell'edge C dell'immagine soprastante, mentre al contrario, **se i punti di edge “deboli” non sono connessi ad alcun punto di edge “forte”**, come ad esempio l'edge B nell'immagine soprastante, questi **vengono scartati** e dunque non considerati edge validi.

Quest'operazione aggiuntiva viene effettuata poiché gli edge che superano la soglia alta danno vita ad immagini in cui (gli edge) presentano dei “vuoti”.

Quindi, per evitare questa problematica, si esegue la seguente procedura:

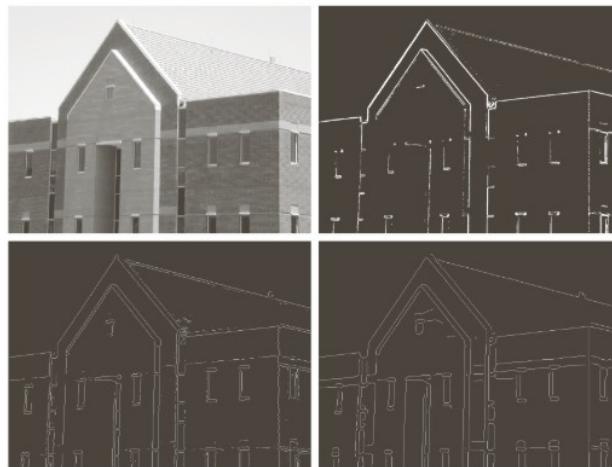
1. si localizza il prossimo pixel di edge p nell'insieme dei pixel di edge “forti”;
2. si etichettano come pixel di edge “forte” tutti i pixel di edge “debole” presenti nell'intorno 3×3 del pixel di edge p ;
3. si ripete il passo 1 per tutti i pixel appartenenti all'insieme di pixel di edge “forti”.

Dal punto di vista implementativo non è necessario creare le due immagini contenenti gli insiemi di pixel di edge “forti” e “deboli”, in quanto è possibile eseguire il thresholding con isteresi direttamente sull'immagine.

In definitiva, ricapitolando, **l'algoritmo di Canny segue alcuni step**:

1. convoluzione dell'immagine di input con filtro Gaussiano;
2. calcolo della magnitudo e dell'orientazione del gradiente;

3. applicazione della Non Maxima Suppression;
4. applicazione del thresholding con isteresi.

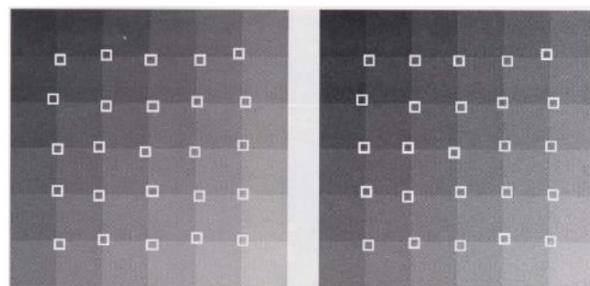


Algoritmo di Harris (corner detector)

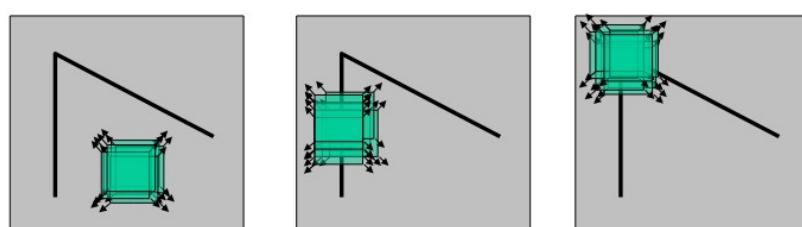
L'algoritmo di Harris permette di **individuare i corner** (angoli), che sono un'altra caratteristica importante delle immagini.

In particolare **un corner rappresenta l'intersezione di linee all'interno di un'immagine o strutture legate a pattern di intensità, come gli edge**. Ne viene di conseguenza che **il concetto di corner è strettamente legato a quello di edge**.

I corner, inoltre, hanno una caratteristica particolare, infatti **si dimostrano stabili in sequenze di immagini** (quindi video).



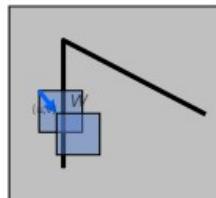
L'idea alla base è che **in una zona di intensità costante la variazione di intensità rilevata è nulla, in corrispondenza di un edge la variazione di intensità si sviluppa in un'unica direzione**, che come sappiamo è ortogonale alla direzione del gradiente, mentre **in corrispondenza di un corner la variazione di intensità si sviluppa in più direzioni**.



Partendo da ciò si va ad analizzare l'immagine considerando delle **finestre**, ed in particolare **si va ad osservare ciò che accade per ogni finestra se questa (la finestra) viene spostata di una certa quantità (u,v)**.

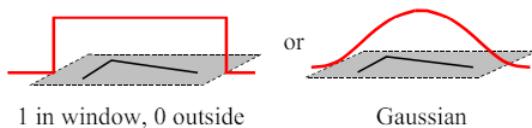
Le variazioni all'interno di una finestra possono essere quindi calcolate come

$$E(u, v) = \sum_{x,y} w(x, y)[I(x + u, y + v) - I(x, y)]^2$$



All'interno di questa formula la finestra è rappresentata dalla funzione $w(x,y)$, utilizzata per assegnare un peso ad ogni pixel dell'immagine. Questa funzione può essere di due tipi:

- **composta da 0 e 1** (funzione box), con valore 1 se il pixel considerato si trova all'interno della finestra e 0 altrimenti;
- **Gaussiana**, sempre con valore 0 se il pixel considerato si trova fuori dalla finestra e con valori che aumentano man mano che ci si avvicina ai pixel centrali.



L'altra parte della formula, ossia

$$[I(x + u, y + v) - I(x, y)]^2,$$

è detta **Sum of Squared Distance (SSD)** e permette di calcolare la variazione di intensità tra il pixel di posizione (x,y) ed il pixel shiftato, a partire da esso (il pixel di posizione (x,y)), **di una quantità (u,v)** . La differenza viene poi elevata al quadrato per esaltare le variazioni di intensità.

Per piccoli spostamenti, la SSD può essere approssimata dal gradiente, visto che $I(x+u, y+v) - I(x, y)$ può essere scomposta in $I(x+u, y) - I(x, y)$ e $I(x, y+v) - I(x, y)$, che rappresentano le derivate parziali e quindi le componenti del vettore gradiente. Dopodiché **per effettuare il quadrato di questo vettore bisogna moltiplicarlo per il suo trasposto, dando luogo ad una matrice** come segue

$$E(u, v) = \sum_{x,y} w(x, y)[I(x + u, y + v) - I(x, y)]^2$$

$$E(u, v) = \sum_{x,y} w(x, y)[\nabla I(u, v)\nabla I(u, v)^T], \text{ dove } \nabla I(u, v) = [I_u \ I_v]$$

$$E(u, v) = \sum_{x,y} w(x, y) \begin{bmatrix} I_u^2 & I_u * I_v \\ I_u * I_v & I_v^2 \end{bmatrix}$$

$$E(u, v) = \begin{bmatrix} \sum_{x,y} w(x, y) I_u^2 & \sum_{x,y} w(x, y) I_u * I_v \\ \sum_{x,y} w(x, y) I_u * I_v & \sum_{x,y} w(x, y) I_v^2 \end{bmatrix}$$

La matrice E risulta simmetrica, ossia è uguale alla sua trasposta, per cui può essere decomposta tramite la tecnica della decomposizione ai valori singolari (SVD). Tramite questa tecnica la matrice E viene decomposta in 3 sottomatrici:

- U, che contiene gli autovettori di E sulle colonne;
- U^T , che contiene gli autovettori di E sulle righe;
- S, che avrà sulla diagonale gli autovalori (λ_1, λ_2) della matrice E.

$$E = USU^T = U \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} U^T$$

Viene eseguita tale decomposizione per utilizzare autovalori e autovettori in modo da descrivere le variazioni di intensità all'interno della matrice E(u,v). In particolare:

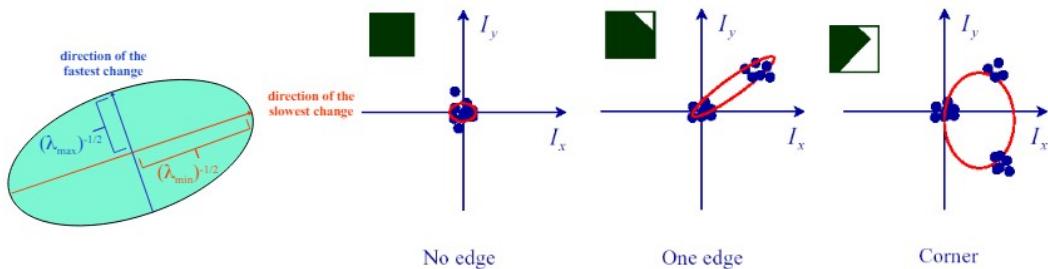
- se avrò entrambi gli autovalori della matrice S circa uguali a 0, allora significa che ci troviamo in una **zona di intensità uniforme**;
- se avrò $\lambda_1 \gg 0$ e $\lambda_2 \approx 0$, allora significa che nell'intorno del punto considerato si ha **una sola variazione di intensità**, che corrisponde alla presenza di un **edge**, nella direzione dell'autovettore associato a λ_1 , con l'edge che è ortogonale all'autovettore;
- se avrò entrambi gli autovalori molto maggiori di 0, allora significa che nell'intorno del punto considerato si ha variazione di intensità sia nella direzione dell'autovettore associato a λ_1 che nella direzione dell'autovettore associato a λ_2 e, di conseguenza, **ci troviamo in presenza di un corner**.

Dall'interpretazione geometrica degli autovalori possiamo formare un'ellisse, nella quale gli autovettori indicano le direzioni di variazione di intensità, mentre la lunghezza degli assi è data dagli autovalori, che indicano quanto è forte la variazione di intensità nella direzione indicata appunto dall'autovettore corrispondente. Di conseguenza:

- se gli assi sono molto piccoli, tutti i valori sono prossimi allo 0, e dunque mi trovo in un'area ad intensità costante;
- se un'asse è molto grande e l'altra è molto piccola, allora ci troviamo in corrispondenza di un **edge**, poiché alcuni valori sono prossimi allo 0 mentre

altri sono caratterizzati da una forte intensità, però ovviamente lungo una sola direzione;

- se entrambi gli assi sono molto grandi, allora ci troviamo in corrispondenza di un **corner**, in quanto ci saranno forti variazioni di intensità in più direzioni.



Il calcolo della SVD andrebbe ripetuto per ogni punto dell'immagine, ma ovviamente si tratta di un'operazione **computazionalmente onerosa**, di conseguenza **l'idea è quella di utilizzare degli indici più veloci da calcolare**, ossia **traccia e determinante di una matrice**. In particolare **la traccia è data dalla somma degli autovalori**, mentre **il determinante è dato dal prodotto degli stessi (autovalori)**.

$$\text{trace}(C(u,v)) = \lambda_1 + \lambda_2$$

$$\det(C(u,v)) = \lambda_1 * \lambda_2$$

Se dalla matrice E calcoliamo traccia e determinante, possiamo poi ottenere un **indice R** che offre un'approssimazione della SVD, in quanto dipende sempre dall'andamento degli autovalori. L'indice R di un pixel è **dato dal determinante della matrice calcolata nell'intorno meno il quadrato della traccia, moltiplicato per un certo valore k**.

$$R(u,v) = \det(C(u,v)) - k \text{ trace}^2(C(u,v))$$

Il valore di questo indice R deve essere poi sogliato, ma se è molto maggiore di 0 significa che nell'intorno è presente un **corner**.

Per calcolare l'indice R consideriamo la matrice E(u,v), identificando le sue componenti C come $C_{00}, C_{01}, C_{10}, C_{11}$. I passaggi da seguire sono:

- **calcolo del determinante**, dato dal prodotto della diagonale principale meno il prodotto della diagonale secondaria;
- **calcolo della traccia**, dato dalla somma delle componenti della diagonale principale;
- **calcolo dell'indice R**, che come detto è dato dal determinante appena calcolato meno il quadrato della traccia, moltiplicato per un certo valore k.

$$E(u, v) = \begin{bmatrix} \sum_{x,y} w(x, y) I_u^2 & \sum_{x,y} w(x, y) I_u * I_v \\ \sum_{x,y} w(x, y) I_u * I_v & \sum_{x,y} w(x, y) I_v^2 \end{bmatrix}$$

$$E(u, v) = \begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix}$$

determinante = $C_{00} * C_{11} - C_{01} * C_{10}$

traccia = $C_{00} + C_{11}$

$R = \text{determinante} - k * \text{traccia}^2$

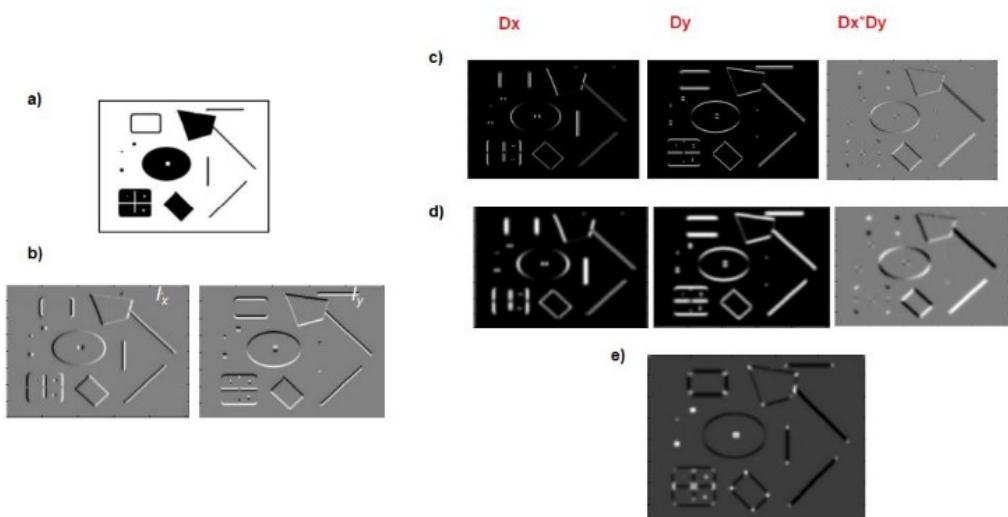
Ricapitolando, nell'algoritmo di Harris, gli step da seguire sono:

1. calcolare le derivate parziali rispetto a x e y (Dx, Dy);
2. calcolare $Dx^2, Dy^2, Dx * Dy$;
3. applicare un filtro Gaussiano a $Dx^2, Dy^2, Dx * Dy$, in modo da ottenere C_{00}, C_{01}, C_{11} (e quindi anche C_{10});
4. calcolare l'indice R;
5. normalizzare l'indice R in [0,255];
6. sogliare R.

Step

- a) immagine originale
- b) derivata lungo l'asse X e l'asse Y
- c) le riporto nell'intervallo [0,255]. Da sinistra verso destra abbiamo Dx, Dy e $Dx * Dy$.
- d) effettuo lo smoothing col filtro Gaussiano;
- e) calcolo gli indici ed individuo i corner (i puntini più bianchi).

Notare che ho una risposta anche all'estremità dei segmenti perché di fatto è l'inizio e la fine dei due bordi, che però non è forte come negli "angoli veri e propri".



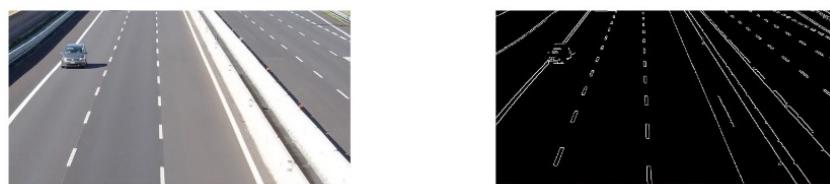
Trasformata di Hough (Lez.8)

La trasformata di Hough è una tecnica che permette di riconoscere particolari configurazioni di punti presenti nell'immagine, come ad es. rette, cerchi e altre forme (vedremo solo le prime 2).

In generale questa tecnica può essere applicata se la forma cercata può essere espressa tramite una funzione nota che fa uso di un insieme di parametri, i quali (parametri) definiscono una particolare istanza della forma.

La trasformata di Hough viene definita come **operatore globale**, in quanto non opera su un intorno di ogni pixel, ma analizza le sue (del pixel) proprietà globali, ossia rispetto agli altri pixel dell'immagine.

La caratteristica della **trasformata di Hough** è che sfrutta gli edge trovati da altri algoritmi, come ad es. quello di Canny, per capire se i pixel di edge rilevati si dispongono globalmente all'interno dell'immagine secondo la forma che stiamo cercando. Ad es. con la trasformata di Hough è possibile individuare delle monete all'interno di un'immagine, cercando dei cerchi, oppure le corsie di un'autostrada, cercando delle rette. In quest'ultimo caso gli edge rilevati da Canny relativi alle linee tratteggiate della carreggiata non danno problemi, in quanto con Hough possiamo sfruttare tutti i punti di edge, anche non vicini, per rilevare delle linee, cercando in quella direzione.



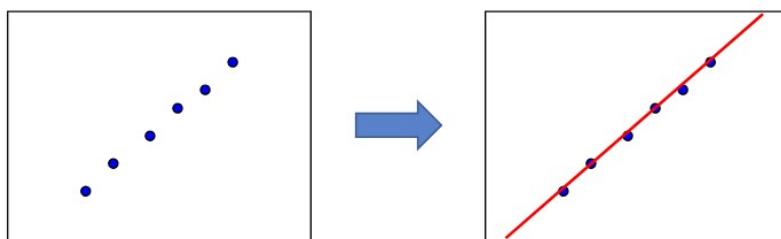
Trasformata di Hough per rette

Le rette possono essere rappresentate sia tramite **forma canonica**, ossia $y = mx + b$, che tramite rappresentazioni diverse, come ad es. la **forma normale di Hesse**, ossia $\rho = x \cos \theta + y \sin \theta$.

Nel primo caso qualunque retta è completamente specificata dal valore dei parametri m e b , ossia **coefficiente angolare e intercetta**. Fissati questi due parametri, al variare di x (**variabile indipendente**) ricavo la y (**variabile dipendente**) e dunque ottengo tutte le coppie di punti (x, y) che si trovano su quella retta.

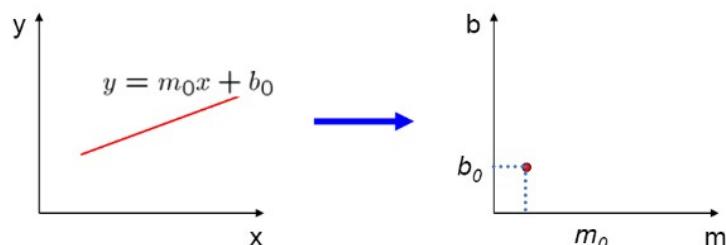
Nel caso della forma normale, invece, un'istanza della retta è caratterizzata dai parametri ρ e θ , che rappresentano **rispettivamente la distanza dall'origine e l'angolo formato dal segmento ρ con l'asse delle x** .

Il nostro scopo, utilizzando la trasformata di Hough per rette e **dati dei punti dell'immagine**, è trovare i parametri delle rette su cui giacciono.



Fissata la forma di interesse, che in questo caso è la retta, e la sua rappresentazione, è possibile considerare una **trasformazione dallo spazio dell'immagine allo spazio dei parametri m e b** , ovvero uno spazio dove l'asse delle ascisse è rappresentata dal **coefficiente angolare m** mentre l'asse delle ordinate è rappresentata dall'**intercetta b** .

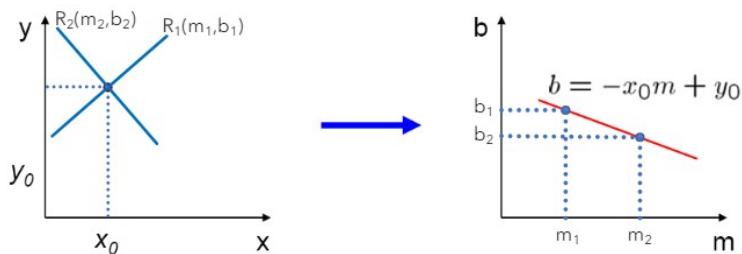
Nello spazio dei parametri, una particolare istanza di retta viene rappresentata da un punto, visto che sarà specificata dalla coppia coefficiente angolare e intercetta.



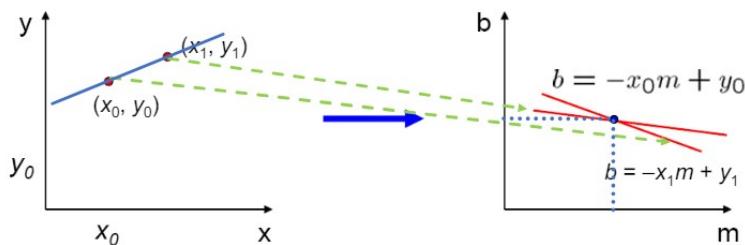
Viceversa, un punto nello spazio immagine viene rappresentato come una retta nello spazio dei parametri. Ciò è diretta conseguenza del fatto che se riscriviamo la forma canonica $y = mx + b$ in funzione dei parametri m (variabile indipendente) e b (variabile dipendente), avremo $b = -xm + y$. Tramite questa formula, fissando x e y , e variando m , possiamo ottenere b . Di conseguenza,

tenendo conto delle diverse combinazioni di m e b , avremo tutte le rette che passano per il punto (x_0, y_0) .

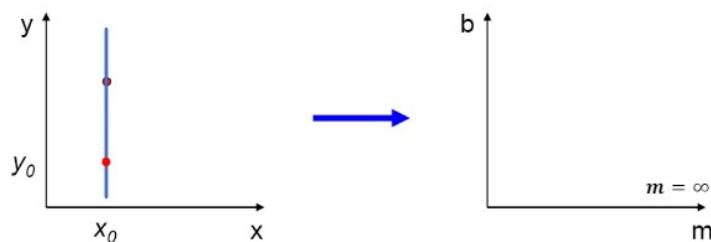
Ovviamente ciò è possibile poiché sappiamo che **per un punto passano infinite rette** e, visto che **nello spazio dei parametri** invertiamo la funzione della forma canonica andando a fissare x e y , **tutti i punti su una stessa retta condividono il coefficiente angolare e l'intercetta**, che in questo caso saranno rappresentati da x_0 e y_0 (ad esempio, ma si potrebbero chiamare in qualsiasi modo).



Se si considerano due punti nello spazio immagine, sappiamo che per entrambi i punti passano infinite rette, ma che **esiste una sola retta passante per entrambi i punti**, la quale è caratterizzata dai parametri m e b . **Nello spazio dei parametri avremo dunque due rette**, il cui punto di intersezione si troverà in **corrispondenza degli stessi parametri m e b** che caratterizzavano l'unica retta passante per entrambi i punti nello spazio immagine.

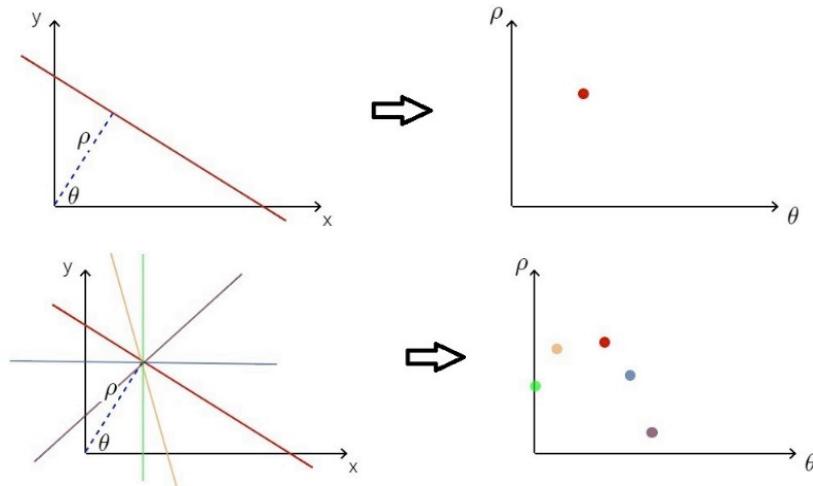


La rappresentazione in forma canonica della retta presenta tuttavia un problema, infatti quando le rette sono verticali, il **coefficiente angolare m sarebbe uguale a ∞** . In tale situazione, tutti i punti che si trovano su questa retta dovrebbero considerare un m infinito, che però **non è rappresentabile nello spazio dei parametri**.

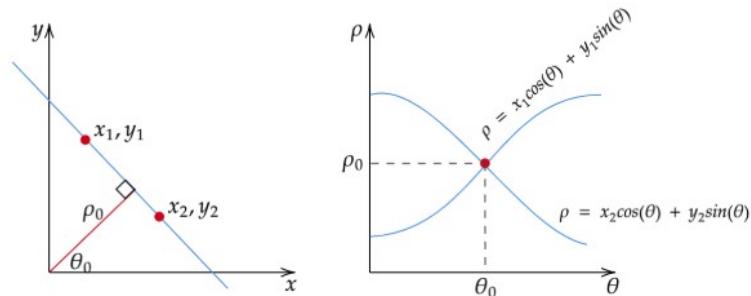


Per questo motivo si utilizza la forma normale di Hesse, o rappresentazione polare, ossia $\rho = x \cos \theta + y \sin \theta$. Anche in questo caso **una retta nello spazio immagine viene rappresentata da un punto nello spazio dei parametri**, che in questo caso sono ρ e θ , che rappresentano **rispettivamente l'asse delle ordinate e l'asse delle ascisse**. Al contrario, però, **un punto nello spazio**

immagine non viene rappresentato da una retta ma **da una sinusoide**. In realtà cambia solo la forma, ma l'idea rimane la stessa.

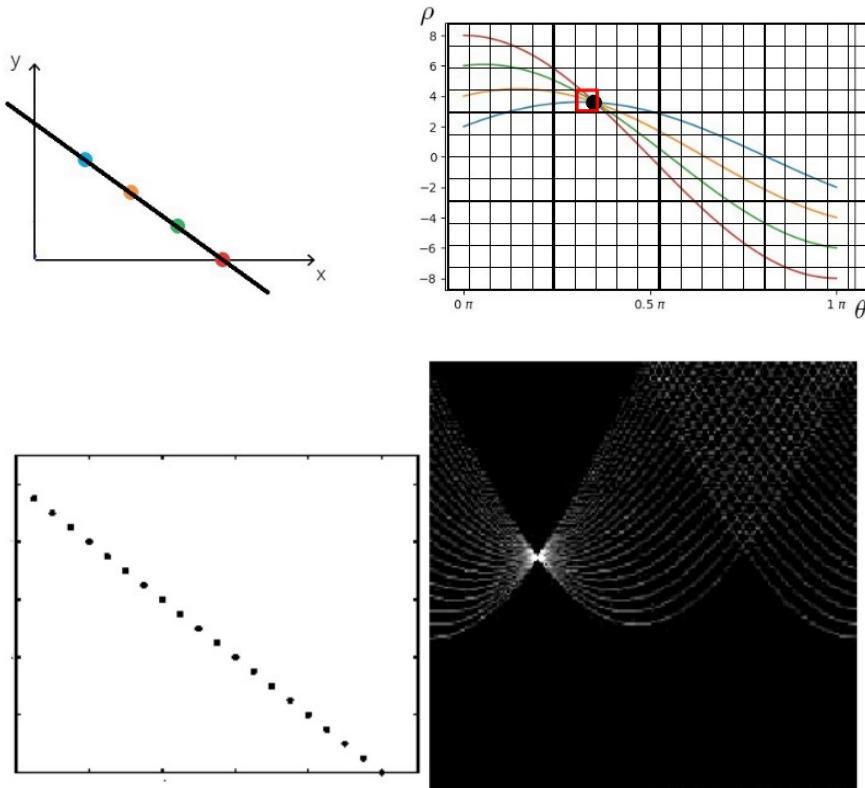


Anche in questo caso, **dati due punti dello spazio immagine, possiamo individuare due sinusoidi nello spazio dei parametri, con il punto di intersezione** tra queste due sinusoidi **che avrà come coordinate quelle relative alle coordinate polari ρ_0 e θ_0 che identificavano la retta passante per i due punti nello spazio immagine.**



Di fatto la forma normale è una soluzione di comodo che non presenta il problema del coefficiente angolare m all'infinito.

Visto che non abbiamo le posizioni precise dei punti di edge a causa del rumore a cui sono soggette le immagini, **per trovare i punti di intersezione delle rette dobbiamo quantizzare lo spazio dei parametri**, considerando **intervalli di valori di ρ e θ** . Lo spazio quantizzato, detto **spazio dei voti** (che in effetti è una matrice), è **caratterizzato da celle dette accumulatori**. Per ogni punto nello spazio immagine, e di conseguenza per ogni sinusode nello spazio dei parametri, **si aggiunge un voto nelle celle attraversate dalla sinusode corrispondente**. Quello che ci aspettiamo, ovviamente, è che **nei punti di intersezione verrà trovato il maggior numero di voti**. Inoltre, **più piccolo sarà l'intervallo $[\rho, \theta]$ e più precisa sarà l'individuazione della retta**. Alla fine della votazione è prassi **effettuare una sogliatura**, in modo da mantenere solo le celle che hanno ricevuto un numero di voti superiori ad una certa soglia appunto.



I passi da seguire nell'implementazione dell'algoritmo di Hough per rette sono:

1. inizializzare l'accumulatore H ;
2. applicare l'algoritmo di Canny per individuare i punti di edge;
3. per ogni punto di edge (x,y) :
 - a. calcolare $\rho = x \cos \theta + y \sin \theta$, per ogni angolo θ tra 0 e 360;
 - b. incrementare $H(\rho, \theta) = H(\rho, \theta) + 1$;
4. una volta che tutti i punti di edge hanno votato, si controlla nello spazio di voto se le celle $H(\rho, \theta)$ hanno valore maggiore di una soglia, quindi semplicemente si effettua un processo di sogliatura.

Trasformata di Hough per cerchi

La trasformata di Hough può essere utilizzata anche per trovare i cerchi presenti all'interno delle immagini, come sappiamo.

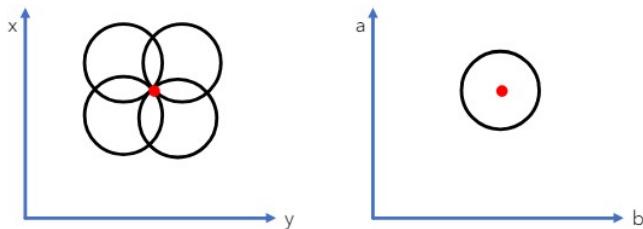
Anche in questo caso **utilizzeremo lo stesso procedimento con spazio dei parametri**, ed in particolare **una circonferenza con centro (a,b) e raggio R** è rappresentata dalle seguenti equazioni parametriche, al variare dell'angolo θ in $[0,360]$.

$$\begin{aligned} x &= a + R \cos \theta \\ y &= b + R \sin \theta \end{aligned}$$

Siccome in un'immagine possono esistere cerchi con centri e raggi differenti, dovremo tenere in considerazione 3 parametri, ossia a , b e R , e quindi sia lo spazio dei parametri che lo spazio dei voti sono tridimensionali.

Partiamo supponendo che R sia fissato, e quindi dei tre parametri delle equazioni parametriche solo due possono variare, ossia a e b , per cui lo spazio dei parametri è un piano bidimensionale classico.

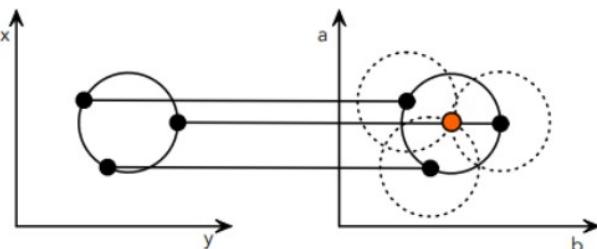
Per un punto nello spazio immagine passano infinite circonferenze di raggio R e al contempo ogni punto nello spazio immagine corrisponde ad una circonferenza nello spazio dei parametri, il cui centro sarà dato da (x_0, y_0) e i punti sulla circonferenza corrispondono ai centri delle circonferenze nello spazio immagine.



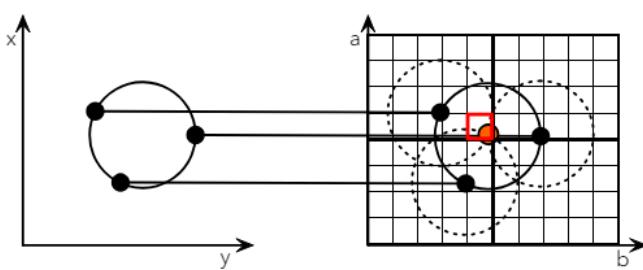
Se si considerano tre punti non allineati nello spazio immagine, per questi passa una sola circonferenza le cui coordinate del centro (a, b) corrispondono al punto di intersezione nello spazio dei parametri delle tre circonferenze che corrispondono ai tre punti nello spazio immagine.

$$a = x - r * \cos\left(\theta * \frac{\pi}{180}\right)$$

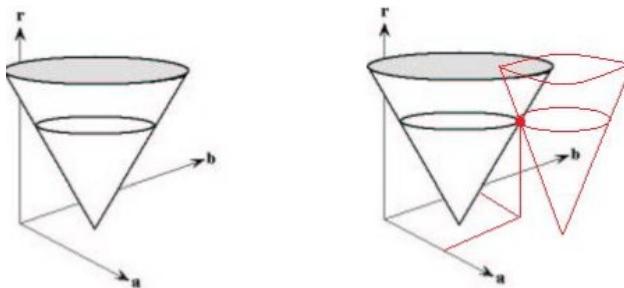
$$b = y - r * \sin\left(\theta * \frac{\pi}{180}\right)$$



A questo punto si procede alla quantizzazione dello spazio considerando solo degli intervalli di valori a e b , in modo da ottenere lo spazio dei voti. In pratica ogni punto della circonferenza vota le celle che attraversa, e di conseguenza, anche in questo caso, ci aspettiamo che il maggior numero di voti si avrà in corrispondenza del punto di intersezione tra le circonferenze nello spazio dei parametri. Chiaramente, più piccolo sarà l'intervallo $[a, b]$ e più precisa sarà l'individuazione della circonferenza.



Nel caso in cui la lunghezza del raggio non sia nota, e quindi fissata, invece, tutti i parametri possono variare, per cui il loro spazio (dei parametri) è tridimensionale. In particolare, ogni punto nello spazio immagine corrisponde alla superficie di un cono nello spazio dei parametri, cioè abbiamo una circonferenza per ogni raggio, tutte con lo stesso centro.



In pratica ogni punto all'interno di un cono voterà per tutti i possibili raggi delle circonferenze con lo stesso centro. Le intersezioni delle superfici di questi coni, invece, individuano le coordinate del centro (a,b) della circonferenza ed il suo raggio R.

Gli step da seguire nell'implementazione dell'algoritmo di Hough per cerchi sono:

1. inizializzare l'accumulatore H;
2. applicare l'algoritmo di Canny per individuare i punti di edge;
3. per ogni punto di edge (x,y):
 1. calcolare $a = x - r \cdot \cos\left(\frac{\theta * \pi}{180}\right)$ e $b = y - r \cdot \sin\left(\frac{\theta * \pi}{180}\right)$, per ogni angolo θ tra 0 e 360 e per ogni raggio r tra R_{min} e R_{max} ;
 2. incrementare $H(a,b,r) = H(a,b,r) + 1$;
4. una volta che tutti i punti di edge hanno votato, si controlla nello spazio di voto se le celle $H(a,b,r)$ hanno valore maggiore di una soglia, quindi semplicemente si effettua un processo di sogliatura.

Caratteristiche della trasformata di Hough

La trasformata di Hough ha alcune caratteristiche:

- per ridurre il numero dei parametri è possibile utilizzare l'**informazione relativa alla direzione del gradiente**;
- è **robusta al rumore** presente nell'immagine e ad eventuali interruzioni (gap) nelle forme ricercate;
- può individuare **più istanze di una forma in una singola esecuzione**;
- è **parallelizzabile**, in modo da suddividere il carico di lavoro che altrimenti risulta essere computazionalmente molto oneroso.

Trasformata di Hough in OpenCV

La funzione che implementa la trasformata di Hough per rette è **HoughLines()**, che **richiede in input gli edge dell'immagine**, quindi **bisogna prima applicare Canny**. L'immagine deve essere a 8 bit e verrà trattata come se fosse un'immagine binaria, mentre invece **lines** è un **vettore che contiene i parametri di ciascuna linea rilevata**, ossia (ρ, θ) . Dopodichè abbiamo ρ che è la **distanza dalle coordinate di origine, in pixel**, e θ che è l'**angolo della linea, in radianti**. Infine abbiamo **threshold** che semplicemente è il **valore soglia** che si utilizza nel processo di sogliatura.

```
void cv::HoughLines(
    cv::InputArray image,           // Input single channel image
    cv::OutputArray lines,          // N-by-1 two-channel array
    double rho,                   // rho resolution (pixels)
    double theta,                 // theta resolution (radians)
    int threshold,                // Unnormalized accumulator threshold
    double srn = 0,                // rho refinement (for MHT)
    double stn = 0                 // theta refinement (for MHT)
);
```

La funzione che implementa la trasformata di Hough per cerchi, invece, è **HoughCircles()**, che **richiama al suo interno l'algoritmo di Canny**, e quindi basterà passargli l'immagine originale. In questo caso **l'output dell'immagine circles sarà un vettore che contiene i 3 parametri a, b e R relativi ai cerchi trovati**. Poi abbiamo **method**, che equivale al metodo utilizzato per individuare i cerchi, ma in pratica si utilizza sempre la macro **HOUGH_GRADIENT**, **dp**, che sarebbe la **risoluzione dell'accumulatore**, il che permette di modificare le **dimensioni dello spazio di voto**, e **minDist**, che è la **distanza minima che deve esserci tra due centri per considerarli di cerchi diversi**. I due parametri successivi sono relativi alla **soglia alta dell'algoritmo di Canny** e al **valore soglia applicato nel processo di sogliatura**.

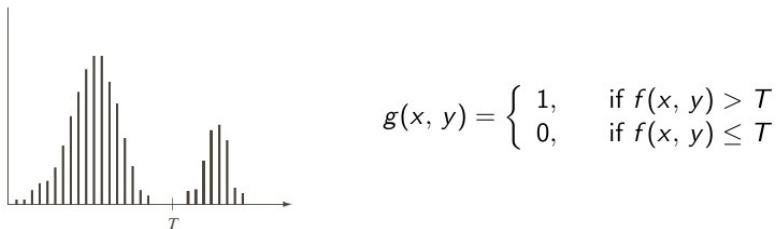
```
void cv::HoughCircles(
    cv::InputArray image,           // Input single channel image
    cv::OutputArray circles,         // N-by-1 3-channel or vector of Vec3f
    int method,                    // Always cv::HOUGH_GRADIENT
    double dp,                      // Accumulator resolution (ratio)
    double minDist,                // Required separation (between lines)
    double param1 = 100,            // Upper Canny threshold
    double param2 = 100,            // Unnormalized accumulator threshold
    int minRadius = 0,              // Smallest radius to consider
    int maxRadius = 0               // Largest radius to consider
);
```

Sogliatura (Lez.9)

La sogliatura è una delle tecniche più importanti per la segmentazione delle immagini, in quanto **intuitiva, semplice da implementare** e **veloce computazionalmente**.

Le tecniche di sogliatura si basano sull'analisi dei valori di intensità e su alcune proprietà per la partizione delle immagini in regioni, con la prima (l'analisi) che viene solitamente effettuata sull'istogramma dei valori di intensità.

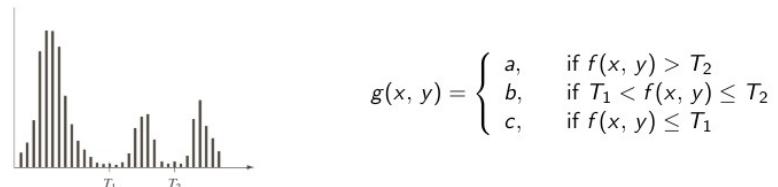
Per spiegare questa tecnica consideriamo di avere l'**istogramma** (diagramma che permette di rappresentare la distribuzione dei dati in un insieme, che nel nostro caso si occupa di valori di intensità in un'immagine) di un'immagine $f(x,y)$ composta da oggetti chiari su sfondo scuro. In questo caso, infatti, i valori saranno raggruppati in due mode (valori che si presentano con maggior frequenza) principali e per segmentare l'immagine è possibile scegliere una soglia T che le separi (le due mode).



Esistono diversi tipi di sogliatura:

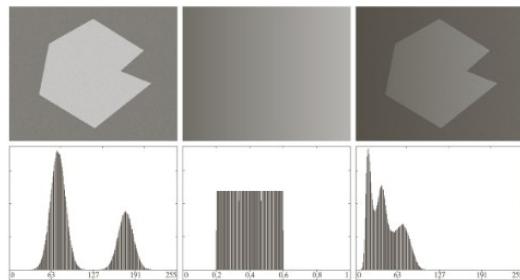
- **globale**, quando la soglia viene applicata a tutta l'immagine;
- **variabile**, quando la soglia viene modificata durante il processo;
- **locale**, quando la soglia dipende dall'intorno di ogni pixel;
- **dinamica**, se la soglia dipende dalla posizione del pixel;
- **multipla**, se prevede l'utilizzo di più soglie.

Di seguito abbiamo un esempio di sogliatura multipla con 2 soglie e, di conseguenza, 3 gruppi (mode).



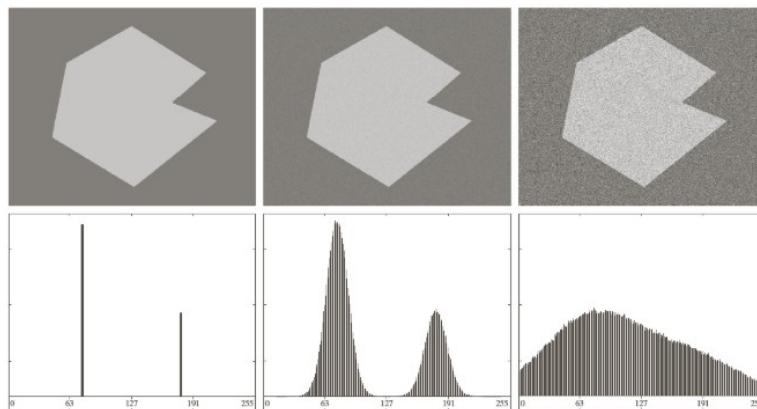
La sogliatura è influenzata da diversi fattori:

- **distanza tra i picchi**, poiché più sono distanti tra loro e più è facile trovare la soglia ottimale;
- **rumore nell'immagine**, in quanto ne modifica l'istogramma (dell'immagine);
- **dimensione degli oggetti rispetto allo sfondo**, infatti se ho un oggetto piccolo, il numero di pixel che contribuiscono alla formazione dell'istogramma dell'oggetto sono pochi;
- **uniformità della fonte di illuminazione**, che se non è uniforme si traduce nella scomparsa della "valle" che separa i picchi, rendendo impossibile la segmentazione (come nell'immagine sottostante);
- **uniformità delle proprietà di riflettanza dell'immagine**.



Per spiegare meglio il ruolo del rumore ed il suo impatto sull'istogramma di un'immagine, consideriamo di avere un'immagine senza rumore, il cui istogramma presenta due mode "a chiodo" ben distinte (prima figura sotto).

Nel caso di un'immagine corrotta (seconda figura sotto) da rumore **Gaussiano con media nulla e deviazione standard pari a 10 livelli di intensità** (significa che ci sarà una dispersione tale da far sì che i valori del rumore si distribuiscano in modo che la maggior parte di essi si trovi entro 10 livelli di intensità dalla media, sia verso sinistra (in meno) che verso destra (in più)), invece, le due mode risultano più ampie, ma ancora facilmente separabili. Se però aggiungiamo molto rumore (terza figura sotto), ad esempio un rumore **Gaussiano con media nulla e deviazione standard pari a 50 livelli di intensità**, non è possibile trovare una soglia che consenta di segmentare l'immagine.



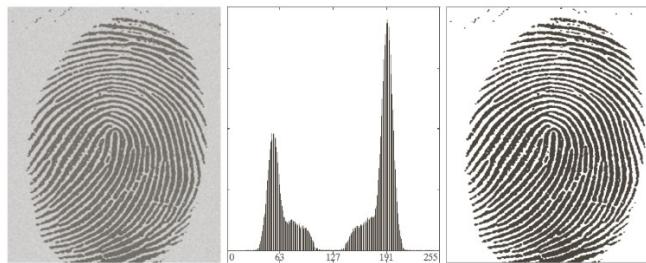
Sogliatura globale

Quando le distribuzioni di intensità dei pixel del background e del foreground sono sufficientemente distinte, è possibile utilizzare un'unica **soglia globale**, che permette di avere 2 mode separate.

Un semplice algoritmo in 5 passi per trovare il valore della soglia potrebbe essere:

1. stimare il valore iniziale della soglia T ;
2. segmentare l'immagine utilizzando T in modo da ottenere due gruppi di pixel, G_1 e G_2 ;
3. calcolare l'intensità media m_1 e m_2 , rispettivamente di G_1 e G_2 ;

4. ricalcolare la soglia T come $T = \frac{(m_1+m_2)}{2}$, calcolando una sorta di media delle media, in modo da riposizionare T in una nuova posizione più equilibrata rispetto al passo precedente;
5. ripetere i passi da 2 a 4 finché le soglie in due iterazioni successive non saranno minori di un valore ΔT tale che $|T_i - T_{i+1}| < \Delta T$, ossia la variazione di T tra due iterazioni successive è minore di un valore da noi stabilito.



Metodo di Otsu (soglia singola)

La sogliatura può essere vista come un problema di statistica in cui si vuole minimizzare l'errore medio dividendo i pixel in due classi.

La soluzione a questo problema è **conosciuta come regola di decisione di Bayes**, la quale, tuttavia, risulta complessa e non adatta ad applicazioni pratiche. **Per questo motivo si utilizza il metodo di Otsu**, che è detto **ottimale in quanto massimizza la varianza interclasse**, ossia la **separazione tra i valori di intensità delle due classi**. Il metodo di Otsu opera esclusivamente sugli **istogrammi delle immagini**, presupponendo, almeno in questo caso, che nell'immagine da sogliare vi siano solo due classi.

Per spiegare il metodo di Otsu supponiamo di avere un'immagine $M \times N$ con livelli di intensità compresi in $[0, L-1]$ e denotiamo con n_i il numero di pixel dell'immagine con intensità i .

Tramite l'istogramma normalizzato, che si ottiene dividendo n_i per il numero di pixel $M \times N$, si può ricavare la probabilità p_i che ha un pixel di avere l' i -esimo livello di intensità. Ovviamente da questa relazione discende il fatto che la sommatoria delle probabilità di tutti i livelli di intensità è uguale ad 1.

$$p_i = \frac{n_i}{MN}, \quad \text{da cui deriva } \sum_{i=0}^{L-1} p_i = 1$$

A questo punto, **selezionando una soglia k compresa tra 0 e $L-1$, è possibile segmentare l'immagine, ottenendo due classi C_1 e C_2 contenenti, rispettivamente, i pixel con intensità minore e maggiore della soglia.**

La probabilità che un pixel appartenga alla classe C_1 è quindi uguale a

$$P_1(k) = \sum_{i=0}^k p_i,$$

mentre la **probabilità che un pixel appartenga alla classe C_2** può essere ricavata a partire dalla relazione precedente, per cui

$$P_2(k) = \sum_{i=k+1}^{L-1} p_i = 1 - P_1(k).$$

Queste due relazioni appena viste prendono anche il nome di somme cumulative per ogni classe.

Dopodichè calcoliamo il **valore medio dell'intensità dei pixel assegnati alle due classi**

$$m_1(k) = \frac{1}{P_1(k)} \sum_{i=0}^k (i+1) p_i \quad m_2(k) = \frac{1}{P_2(k)} \sum_{i=k+1}^{L-1} (i+1) p_i,$$

utilizzando $i+1$ poichè altrimenti la prima moltiplicazione avverrebbe con $i=0$.

Dai valori medi possiamo ricavare la media cumulativa, ossia la **media dei valori di intensità dell'immagine fino al livello di intensità k** , che ovviamente varia all'aumentare dei livelli di intensità considerati, e la **media globale**, ossia l'intensità media dell'intera immagine.

$$m(k) = \sum_{i=0}^k (i+1) p_i \quad m_G = \sum_{i=0}^{L-1} (i+1) p_i$$

Per stimare l'**efficienza della soglia al livello k** si utilizza il valore

$$\eta = \frac{\sigma_B^2}{\sigma_G^2},$$

con η che è la lettera greca "eta", σ_B^2 che rappresenta la **varianza interclasse** (quanto i pixel all'interno di una classe variano rispetto alla loro media) e σ_G^2 che è la **varianza globale** (varianza complessiva di tutti i livelli di intensità presenti nell'immagine), con **queste ultime definite**, a loro volta, **come**

$$\sigma_G^2 = \sum_{i=0}^{L-1} (i+1 - m_G)^2 p_i$$

$$\sigma_B^2 = P_1(m_1 - m_G)^2 + P_2(m_2 - m_G)^2,$$

ma per evitare di calcolare la probabilità P_2 e il valore medio delle intensità m_2 si può ricavare la varianza interclasse come

$$\sigma_B^2 = P_1 P_2 (m_1 - m_2)^2 = \frac{(m_G P_1 - m)^2}{P_1 (1 - P_1)}.$$

Notiamo come **al crescere della distanza delle due medie, la varianza interclasse σ_B^2 aumenta e, di conseguenza, si avrà una maggiore separabilità tra le classi**. Inoltre, dato che nella formula

$$\eta = \frac{\sigma_B^2}{\sigma_G^2}$$

il denominatore è costante, anche eta è una misura della separabilità tra le classi e massimizzare questa quantità significa massimizzare anche σ_B^2 . L'obiettivo tramite il metodo di Otsu sarà, quindi, trovare un valore di intensità k che massimizzi la varianza interclasse σ_B^2 , ossia un k che distanzi il più possibile le due mode dalla media globale. Tale soglia prende il nome di soglia ottimale k^* .

Dopo aver esplicitato k nelle formule precedenti

$$\eta(k) = \frac{\sigma_B^2(k)}{\sigma_G^2} \quad \sigma_B^2(k) = \frac{[m_G P_1(k) - m(k)]^2}{P_1(k)[1 - P_1(k)]},$$

dobbiamo trovare la soglia ottimale k^* che massimizzi la varianza interclasse, con quest'ultima (la massima varianza interclasse) che equivale al massimo di tutte le varianze interclasse **in $[0, L-1]$** al variare di k.

$$\sigma_B^2(k^*) = \max_{0 \leq k \leq L-1} \sigma_B^2(k)$$

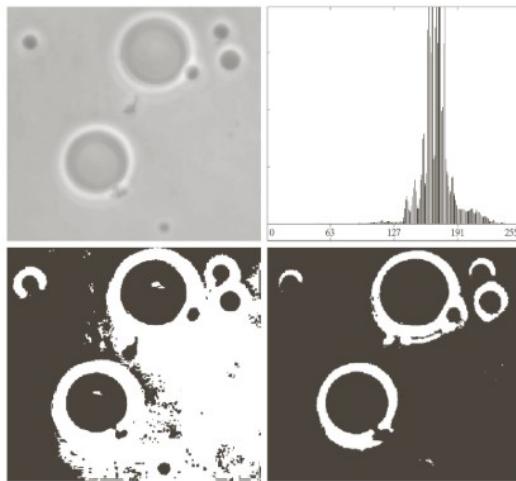
Se la varianza interclasse è massima in più di un valore, per ottenere k^* si calcola la media dei valori di k.

Una volta trovato il valore della soglia ottimale k^* , si può procedere alla segmentazione dell'immagine a partire da questo valore.

Riassumendo, quindi, i **passi da seguire nell'implementazione dell'algoritmo di Otsu** sono:

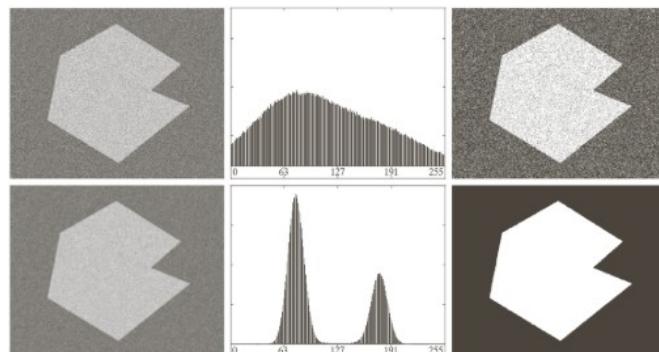
- calcolare l'istogramma normalizzato dell'immagine;
- calcolare le somme cumulative $P_1(k)$ per k in $[0, L-1]$;
- calcolare le medie cumulative $m(k)$ per k in $[0, L-1]$;
- calcolare l'intensità globale media m_G ,
- calcolare la varianza interclasse $\sigma_B^2(k)$ per k in $[0, L-1]$;
- calcolare la soglia ottimale k^* , ovvero il valore k per cui $\sigma_B^2(k)$ è massima;
- (opzionale) calcolare il valore di separabilità $\eta(k)$.

Di seguito troviamo un confronto tra un algoritmo di sogliatura globale (in basso a sinistra) e l'algoritmo di Otsu (in basso a destra).

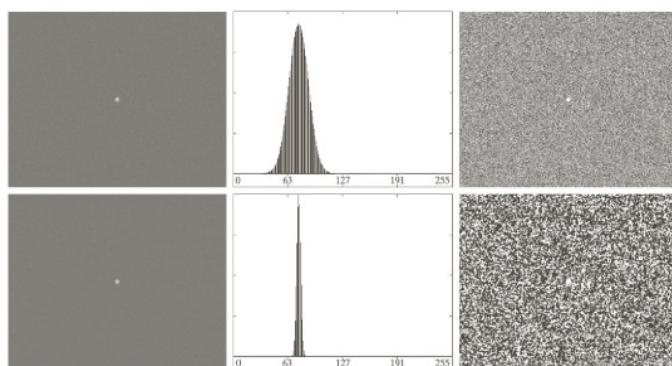


Smoothing per migliorare la sogliatura

Visto che la presenza di rumore può rendere difficile il problema della sogliatura, **può essere effettuato lo smoothing dell'immagine prima di eseguire il processo di sogliatura, in modo da ridurre il rumore.**



Nel caso in cui la regione da segmentare sia talmente piccola da rendere il suo contributo all'istogramma trascurabile rispetto a quello del rumore, lo smoothing non è utile poiché non risolve il problema.



Utilizzo degli edge per migliorare la sogliatura

Per migliorare l'istogramma di un'immagine ai fini della segmentazione è possibile considerare solo i pixel che si trovano sugli edge tra gli oggetti e il

background. Questo perché la probabilità che un pixel di edge appartenga al foreground è pressochè uguale alla probabilità che appartenga al background.

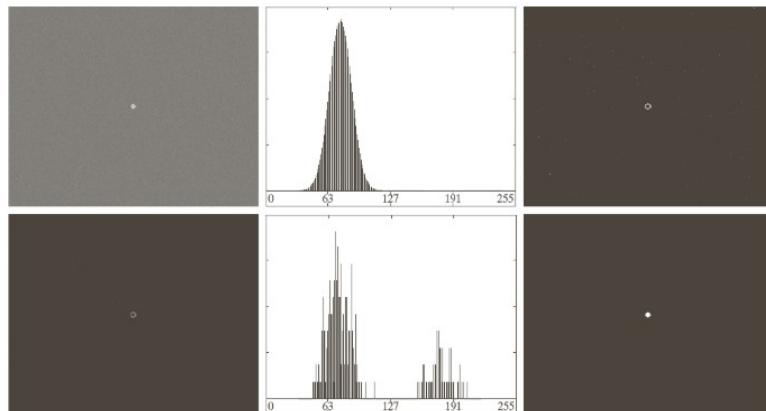
Questo tipo di tecnica potrebbe essere utilizzata per effettuare la segmentazione di regioni (e quindi oggetti) **piccole**, ossia dove falliva l'approccio con lo smoothing insomma.

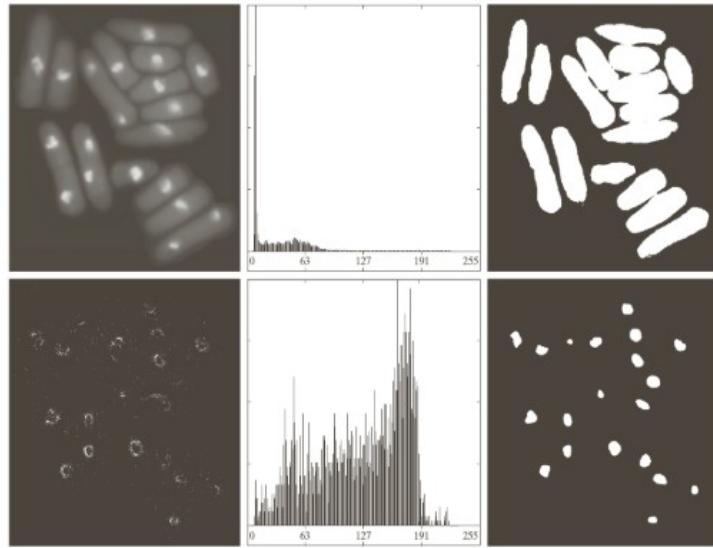
Per estrarre i pixel di edge è possibile utilizzare sia il Laplaciano che ad esempio l'algoritmo di Canny.

Nel caso in cui si utilizzino gli edge, i **passi da seguire** sono:

- calcolare un'immagine che al suo interno contenga gli edge di un'immagine di partenza;
- individuare un valore soglia T , che inizialmente viene scelto per essere maggiore del 90% dei valori di intensità dell'insieme, ma visto che stiamo trattando dei pixel di edge significa che ne stiamo prendendo la maggior parte;
- applicare la soglia T all'immagine di edge, in modo da ottenere un'immagine binaria g_T ;
- calcolare l'istogramma utilizzando solo i pixel dell'immagine di input che corrispondono alle posizioni dei pixel con valore 1 nell'immagine binaria g_T ;
- utilizzare l'istogramma ottenuto per la segmentazione con il metodo di Otsu.

Di seguito abbiamo la prima immagine, che è relativa ai vari passaggi di questa tecnica ricavando gli edge tramite gradiente (ad es. Canny), e la seconda immagine, che invece esegue questa tecnica mediante individuazione degli edge tramite Laplaciano.





Metodo di Otsu (soglie multiple)

Il metodo di Otsu può anche essere generalizzato al caso di K soglie, tenendo conto, però, che **con troppe classi il metodo perde di significato.** In linea generale, **per il metodo di Otsu, non si utilizza mai un numero di soglie k>2.**

Se consideriamo k=2, ovvero 3 classi, la varianza interclasse è data da

$$\sigma_B^2(k_1, k_2) = P_1(m_1 - m_G)^2 + P_2(m_2 - m_G)^2 + P_3(m_3 - m_G)^2,$$

le **soglie ottimali** sono

$$\sigma_B^2(k_1^*, k_2^*) = \max_{0 < k_1 < k_2 < L-1} \sigma_B^2(k_1, k_2)$$

ed il **grado di separabilità** è

$$\eta(k_1^*, k_2^*) = \frac{\sigma_B^2(k_1^*, k_2^*)}{\sigma_G^2}.$$

In questo caso le **somme cumulative e le medie cumulative** per ogni classe saranno date da

$$P_1 = \sum_{i=0}^{k_1} p_i$$

$$P_2 = \sum_{i=k_1+1}^{k_2} p_i$$

$$P_3 = \sum_{i=k_2+1}^{L-1} p_i$$

$$m_1 = \frac{1}{P_1} \sum_{i=0}^{k_1} (i+1) p_i$$

$$m_2 = \frac{1}{P_2} \sum_{i=k_1+1}^{k_2} (i+1) p_i$$

$$m_3 = \frac{1}{P_3} \sum_{i=k_2+1}^{k_3} (i+1) p_i,$$

mentre la **media globale** e la **somma delle probabilità** sono uguali a

$$P_1 m_1 + P_2 m_2 + P_3 m_3 = m_G$$

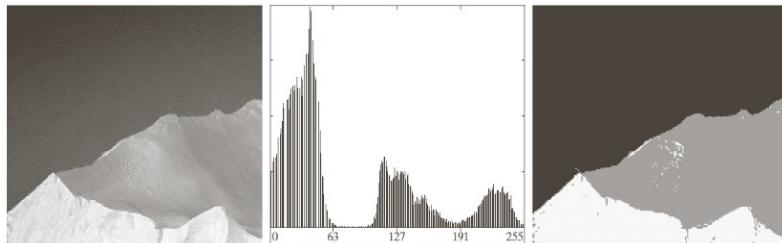
$$P_1 + P_2 + P_3 = 1$$

La procedura inizia selezionando la prima soglia $k_1=1$, in quanto non ha senso cercare una soglia con intensità pari a 0, **e poi si incrementa la soglia** k_2 **a partire dai valori maggiori di** k_1 **e minori di** $L-1$.

Successivamente si incrementa k_1 **e**, di conseguenza, si incrementa **anche la soglia** k_2 **a partire dai valori maggiori di** k_1 . Al termine della procedura **si ottiene una matrice** $\sigma_B^2(k_1, k_2)$ (delle varianze interclasse con le diverse coppie di soglie), **all'interno del quale troviamo il valore massimo in corrispondenza della coppia di soglie ottimali** k_1^* **e** k_2^* .

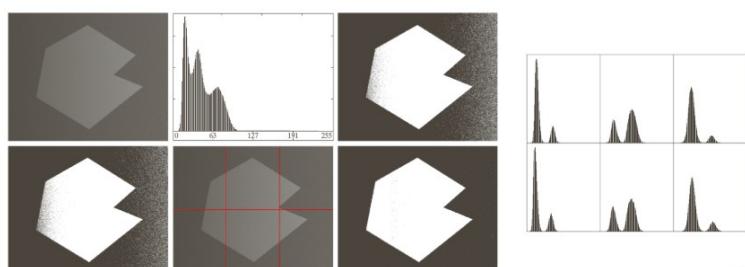
L'immagine segmentata si ottiene da (a, b e c sono classi ovviamente)

$$g(x, y) = \begin{cases} a & \text{se } f(x, y) \leq k_1^* \\ b & \text{se } k_1^* < f(x, y) \leq k_2^* \\ c & \text{se } f(x, y) > k_2^* \end{cases}$$



Sogliatura variabile

Uno dei metodi più semplici di sogliatura variabile consiste nel **suddividere l'immagine in rettangoli non sovrapposti**, ma si possono ottenere risultati efficaci anche mediante approcci più generali.



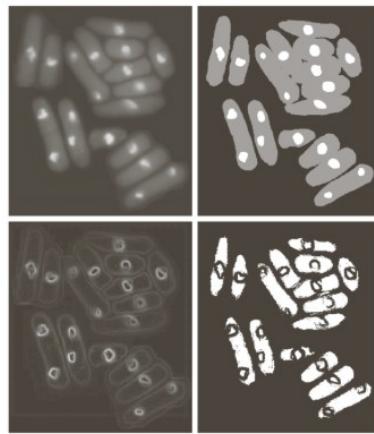
Un approccio più generale consiste nel calcolare una soglia considerando una o più proprietà dell'intorno di ogni pixel, ad esempio considerando la **media** (m_{xy}) e la **deviazione standard** (σ_{xy} , la deviazione standard dovrebbe essere uguale alla radice quadrata della varianza) **dei valori di intensità nell'intorno** (S_{xy}) **di ogni pixel**.

$$T_{xy} = a\sigma_{xy} + bm_{xy}$$

$$T_{xy} = a\sigma_{xy} + bm_G$$

In alternativa è anche possibile utilizzare un predicato sui parametri locali, come ad esempio

$$Q(\sigma_{xy}, m_{xy}) = \begin{cases} \text{vero} & \text{se } f(x, y) > a\sigma_{xy} \text{ AND } f(x, y) > bm_{xy} \\ \text{falso} & \text{altrimenti} \end{cases}$$



Thresholding OpenCV

Per effettuare la vera e propria sogliatura a partire da un certo valore soglia, come ad esempio quello che ricaviamo dal metodo di Otsu, [OpenCV ci offre diverse possibilità che abbiamo già esplorato precedentemente](#).

L'unica aggiunta è che è possibile utilizzare **THRESH_OTSU** come tipo di thresholding nella function `threshold()` nel caso in cui, appunto, si voglia eseguire la sogliatura mediante metodo di Otsu.

Region Growing e Split and Merge (Lez.10)

Region Growing

Innanzitutto ricordiamo le [nozioni principali sulle regioni](#), incontrate qualche lezione fa.

La tecnica del region growing (accrescimento di regione) consiste nel **raggruppare i pixel in regioni sempre più grandi, in base a dei criteri predefiniti**.

Il procedimento inizia da punti detti seed (seme) e si propaga ai pixel adiacenti che rispettano delle proprietà predefinite, i quali vengono aggiunti alla regione. Quando la regione non può più essere espansa, si ricontrollano i pixel partendo dal seed e si controlla che non facciano già parte di una regione, poichè in tal caso quel pixel viene considerato, a sua volta, un seed. Tale procedimento permette di avere una segmentazione completa (ossia l'unione di tutte le regioni è uguale all'intera immagine e, per fare ciò, ogni pixel deve essere contenuto all'interno di una regione).

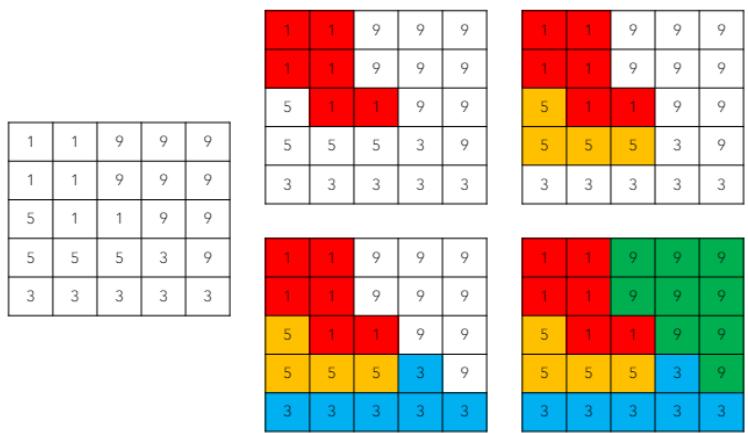
Il criterio di similarità dipende dal tipo di immagine, infatti, ad esempio, **per immagini in scala di grigio si può basare sui livelli di intensità**. Bisogna però sempre fare attenzione alla connettività, in quanto raggruppare pixel “simili” ma non adiacenti può generare segmentazioni inconsistenti.

Un altro problema riguarda la **regola d'arresto**, che viene **utilizzata per arrestare l'accrescimento della regione quando i pixel non soddisfano più i criteri di inserimento all'interno della regione considerata**. In tal senso non è sufficiente considerare solo il livello di intensità dei pixel adiacenti, ad esempio, poichè non si tiene conto della “storia” del processo di accrescimento. E' perciò **opportuno considerare le caratteristiche di tutti i pixel che sono già stati inseriti nella regione, come ad esempio il seed da cui si è partiti**.

Per spiegare meglio il funzionamento di questa tecnica, **consideriamo di avere un'immagine di input $f(x,y)$, una matrice dei seed $S(x,y)$ che assegna valore 1 alle posizioni dei seed e 0 alle altre posizioni e un predicato Q da applicare ad ogni pixel**. L'algoritmo di region growing segue alcuni passi:

1. formare l'immagine f_Q che nel punto (x,y) contiene il valore 1 se il predicato applicato a questo punto $Q(f(x,y))$ è vero, 0 altrimenti;
2. aggiungere ad ogni seed i pixel impostati ad 1 in f_Q che risultano 4 o 8-connessi, in base a come si vuole procedere, al seed stesso;
3. marcare ogni componente连通 con un'etichetta diversa.

Di seguito troviamo un primo esempio in cui il livello di intensità del pixel deve essere uguale a quello del seed, utilizzando la 8-connettività



mentre nell'esempio successivo il livello di intensità del pixel deve essere al massimo più grande di due unità rispetto a quello del seed (infatti $3-1 = 2$).

1	1	9	9	9
1	1	9	9	9
5	1	1	9	9
5	5	5	3	9
3	3	3	3	3

1	1	9	9	9
1	1	9	9	9
5	1	1	9	9
5	5	5	3	9
3	3	3	3	3

1	1	9	9	9
1	1	9	9	9
5	1	1	9	9
5	5	5	3	9
3	3	3	3	3

1	1	9	9	9
1	1	9	9	9
5	1	1	9	9
5	5	5	3	9
3	3	3	3	3

Questo algoritmo non è ideale se si devono trattare immagini con molti colori diversi, o anche con molti oggetti al suo interno (in generale con **immagini complesse insomma), come possiamo vedere dall'esempio seguente.**

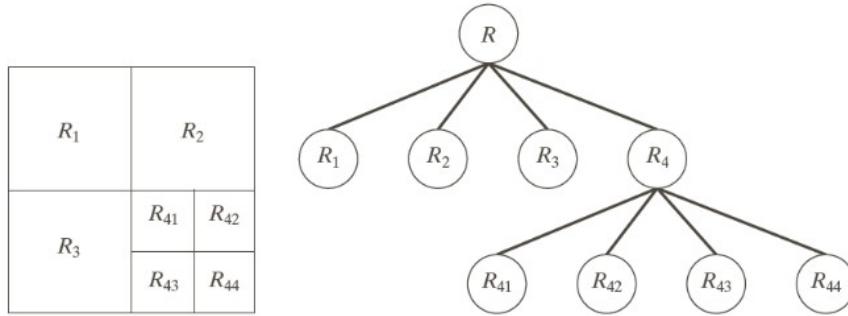


Split and Merge

Lo split and merge è una **tecnica che consiste nel dividere (split) l'immagine in regioni disgiunte di forma e dimensioni arbitrarie e successivamente fonderle (merge) in base a dei criteri di similarità.**

Considerando R la regione corrispondente all'intera immagine e Q un predicato, è possibile dividere R in regioni sempre più piccole finché il predicato non risulta vero, o anche finché la regione diventa troppo piccola per poter essere ulteriormente divisa.

Una strategia molto utilizzata consiste nel dividere le regioni in quadranti mediante **quadtrees**, ossia **alberi di quadranti**. Partendo da R, se il predicato applicato alla regione, ossia $Q(R)$, è falso, si divide R in quattro quadranti. Dopodiché i quadranti per cui il predicato è falso si dividono ulteriormente e così via, finché il predicato non risulterà vero o finché la regione non diventa troppo piccola per poter essere ulteriormente divisa.



Al termine della fase di splitting, la partizione finale potrebbe contenere regioni adiacenti con caratteristiche simili e, per questo motivo, tali regioni possono essere fuse (nella fase di merging), secondo la proprietà delle regioni per cui se uno stesso predicato viene applicato a due regioni e risulta vero, allora queste regioni in realtà ne rappresentano una unica.

I passi da seguire nell'implementazione dell'algoritmo split and merge sono:

- dividere in 4 quadranti tutte le regioni per cui il predicato Q risulta falso;
- quando non è più possibile dividere le regioni, applicare il processo di merging a tutte le regioni adiacenti per cui uno stesso predicato applicato ad entrambe restituisce vero;
- il processo termina quando non è più possibile effettuare unioni (fusioni).

Come abbiamo già detto, **solitamente, si definisce una dimensione minima della regione, oltre la quale non si effettuano divisioni (split).**

Per ragioni di efficienza, la fase di merge si può eseguire se il predicato è vero per le singole regioni adiacenti, poichè risulta computazionalmente oneroso controllare che uno stesso predicato applicato all'unione di due regioni restituisca vero (visto che si tratterebbe di un'operazione insiemistica).



Clustering (Lez.11)

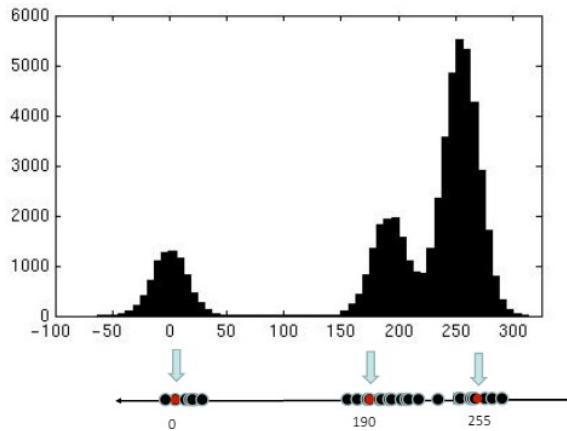
Il clustering è una tecnica che consiste nel raggruppare i pixel in base alle caratteristiche di intensità, colore e posizione, che sono i fattori principali che consentono al sistema visivo umano di raggruppare un insieme di elementi.

Algoritmo K-means

Nell'algoritmo k-means si creano **k** gruppi (cluster) disgiunti, ognuno rappresentato da un centro, ossia la media dei valori di intensità dei pixel appartenenti al cluster.

In questo algoritmo, ogni pixel viene assegnato al gruppo rappresentato dalla media ad esso più vicina e, di conseguenza, i pixel di ogni gruppo vengono etichettati con tale valore (il centro, ossia la media di quel cluster).

La miglior scelta per i centri è quella che minimizza la **Sum of Squared Distance** (SSD) tra tutti i punti ed il centro più vicino, con l'obiettivo di minimizzare la varianza all'interno di ogni cluster.



Il funzionamento del k-means può essere descritto attraverso la seguente funzione

$$\mathbf{c}^*, \delta^* = \operatorname{argmin}_{\mathbf{c}, \delta} \frac{1}{N} \sum_j^K \sum_i^K \delta_{ij} (\mathbf{c}_i - \mathbf{x}_j)^2$$

Cluster center Data
Whether x_j is assigned to c_i

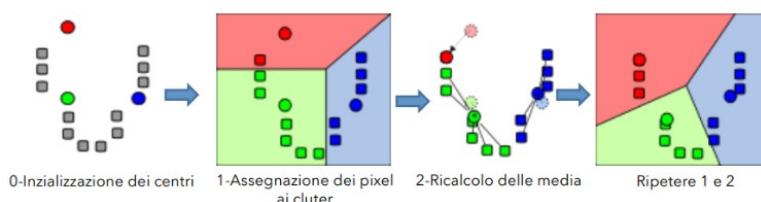
cluster		
δ_{ij}	1	2
1	1	0
2	0	1
3	0	1
4	1	0

pixel

in cui si vogliono trovare i parametri c e δ che minimizzino la funzione, con c che rappresenta il centro del cluster e δ che rappresenta una matrice che ha sulle righe i pixel x e sulle colonne i cluster. In pratica, per ogni pixel nella matrice, si avrà valore 1 in corrispondenza del cluster a cui appartiene e 0 per gli altri cluster.

Il problema di questa funzione è che **si dovrebbero conoscere i centri dei cluster**, in modo da assegnare i pixel al cluster con il centro più vicino, **oppure si dovrebbero conoscere i gruppi**, con all'interno i diversi pixel, in modo da calcolare i centri.

Per ovviare a questo problema i k centri possono essere inizialmente scelti casualmente, in modo da assegnare i pixel al cluster con il centro più vicino. Una volta assegnati tutti i pixel si deve procedere al ricalcolo delle medie, ripetendo il procedimento appena spiegato fino a quando le medie non si spostano più, o comunque si spostano di una quantità trascurabile.



Visto che i centri iniziali vengono scelti casualmente (**l'algoritmo potrebbe terminare in un minimo locale**), per controllare che il risultato di **questo algoritmo** sia abbastanza preciso, **si può rilanciare più volte**, in modo da considerare i risultati più frequenti come "attendibili".

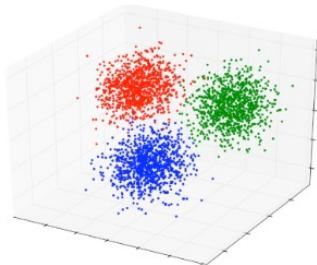
Visto che possono esserci pixel con valori di intensità simili anche se essi si trovano molto distanti tra loro all'interno di un'immagine, **in ogni cluster non è detto che ci sia un'unica componente连通**. Ovviamente però, quando si segmenta un'immagine, si cercano gruppi di pixel "compatti" e quindi, **per ottenere regioni più compatte, è possibile utilizzare l'informazione relativa alla posizione dei pixel nell'immagine**. In questo modo **ogni pixel sarà rappresentato da un vettore le cui componenti saranno le coordinate spaziali ed il livello di intensità**, nel caso in cui si stia trattando un'**immagine in scala di grigio**, oppure **le coordinate spaziali e le tre componenti colore**, nel caso in cui si stia lavorando con un'**immagine a colori**.

I passi da seguire nell'implementazione dell'algoritmo k-means sono:

1. inizializzare i centri di ogni cluster;
2. assegnare ogni pixel al cluster con il centro più vicino, calcolando la distanza (solitamente si utilizza quella euclidea) dei pixel dai k centri;
3. aggiornare i centri, ossia ricalcolare la media dei pixel di ogni cluster;
4. ripetere i punti 2 e 3 finché il centro (media) di ogni cluster non viene più modificato (di conseguenza i cluster non vengono modificati), o comunque le variazioni sono minime.

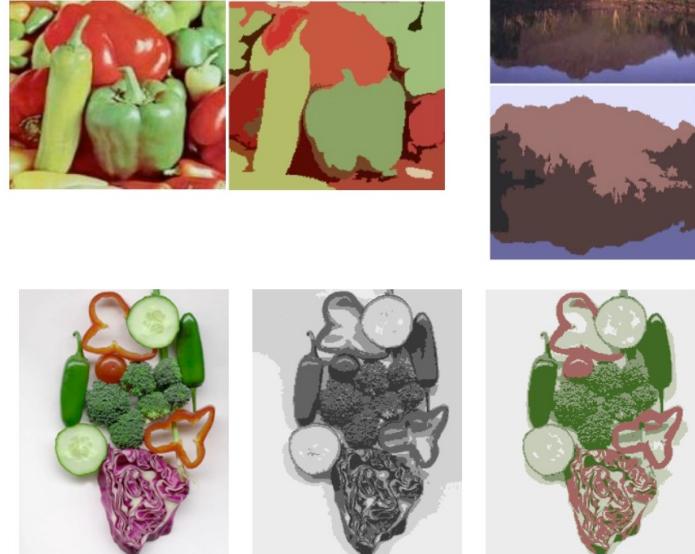


Questo algoritmo può essere utilizzato anche nel caso in cui ci troviamo a lavorare con immagini a colori. In tal caso, ovviamente, **ogni pixel sarà un vettore di tre componenti** e, di conseguenza, **anche i centri di ogni cluster saranno vettori di tre componenti**.



Come si può notare dai seguenti esempi, **applicando il k-means ad immagini a colori si ottiene una clusterizzazione migliore** di quella che si avrebbe applicando l'algoritmo (k-means) sulla stessa immagine ma in scala di grigio.

CLUSTERING: K-MEANS



Il valore k, ossia il numero di cluster disgiunti in cui si vuole dividere un'immagine, **deve essere scelto dall'utente** e dipende ovviamente dal tipo di immagine che si sta trattando.

Per valutare la qualità dei cluster al variare di k ci si basa sulla compattezza degli stessi (cluster). In pratica **più i cluster sono compatti, quindi minore è la**

varianza all'interno degli stessi (cluster), e **più il clustering può essere considerato buono**.

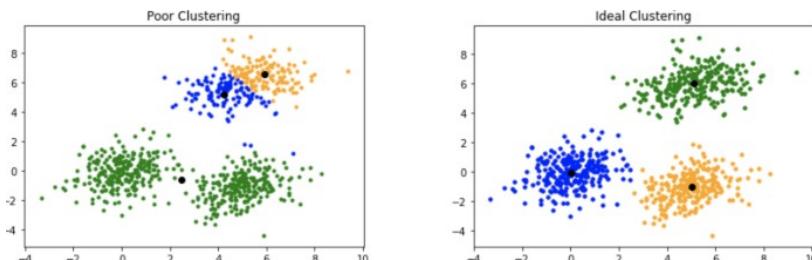
L'algoritmo k-means ha complessità $O(kNd)$, per ogni iterazione, dove k è il numero di cluster, N è il numero di pixel e d è il numero di feature (1 in scala di grigi e 3 a colori, oppure 3 e 5 se consideriamo anche le componenti spaziali). Se fissiamo k e d , quindi, la complessità è grossomodo lineare.

Di seguito abbiamo l'implementazione dell'algoritmo k-means di OpenCV, il quale vuole in input (data) un vettore di dati.

```
double cv::kmeans(                                     // returns (best) compactness
    cv::InputArray      data,                      // Your data, in a float type
    int                 K,                         // Number of clusters
    cv::InputOutputArray bestLabels,                // Result cluster indices (int's)
    cv::TermCriteria     criteria,                  // iterations and/or min dist
    int                 attempts,                  // starts to search for best fit
    int                 flags,                     // initialization options
    cv::OutputArray      centers = cv::noArray() // (optional) found centers
);
```

K-means++

Nell'algoritmo k-means, **nel caso in cui i centri iniziali siano troppo vicini, il risultato finale potrebbe non essere ottimale. Per ovviare a questo problema, i centri iniziali possono essere scelti in modo da essere distanti tra loro.**



I passi da seguire, per inizializzare i centri con questo metodo, sono:

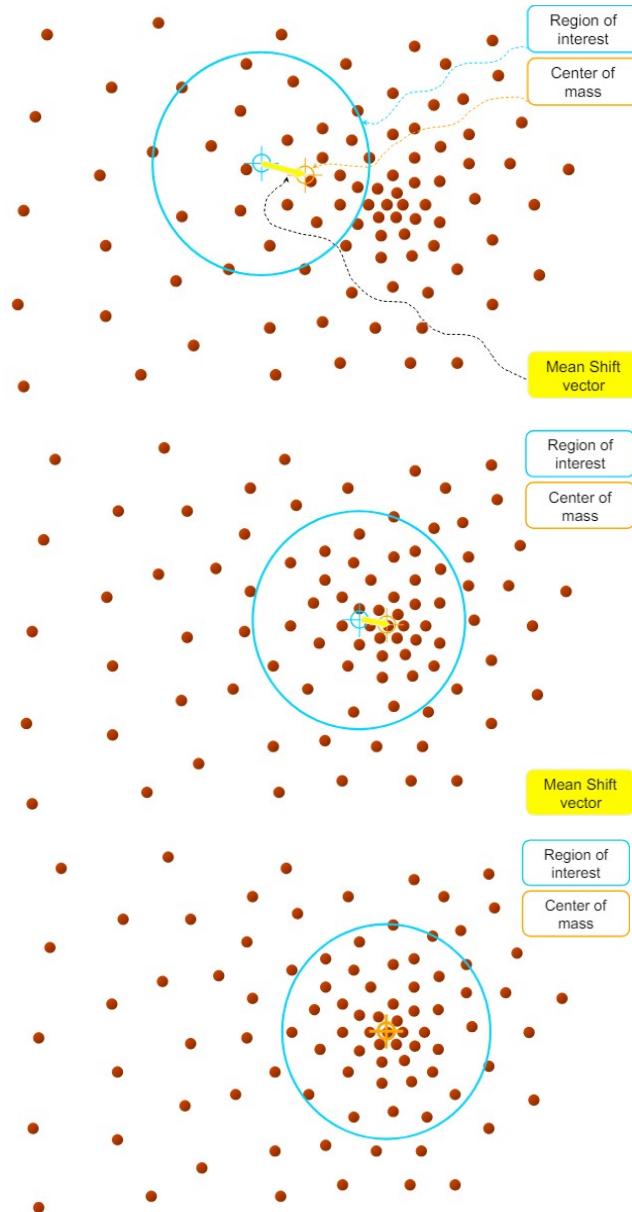
1. scegliere il primo centro c_1 in maniera random;
2. calcolare la distanza di ogni punto dal centro c_1 ;
3. scegliere tra i restanti pixel il prossimo centro c_2 , che deve essere il pixel più lontano da c_1 ;
4. ripetere il punto 3 fino a quando non sono stati scelti tutti i k centri iniziali.

Algoritmo mean-shift

L'algoritmo mean-shift è un **algoritmo di segmentazione più avanzato**, con un'idea simile a quella del k-means. In questo caso, però, il parametro k non deve essere deciso a priori, poiché il numero ottimale di cluster viene individuato a partire dai valori di intensità presenti nelle immagini.

Il mean-shift **consiste nel cercare, per ogni valore, la moda** (valore di intensità con la frequenza più elevata) **più vicina nello spazio delle feature**. Per ogni

valore di intensità si considera una finestra di ampiezza W , si calcola la media (mean) di questa finestra, e la si sposta (shift) sulla media calcolata, ripetendo il procedimento fino a quando la media non cambia, o comunque il cambiamento è minimo.



Di seguito troviamo **due pseudocodici** che implementano l'algoritmo del mean-shift. **Consideriamo prima quello di sinistra:** per ogni punto p dell'immagine l'algoritmo esegue iterazioni finché la distanza tra la posizione (centro) corrente e la posizione al passo precedente è maggiore di una certa soglia. All'interno del ciclo while si inizializzano shift e n a 0, con il primo che accumulerà la somma dei valori di intensità dei pixel contenuti nella finestra ed il secondo che invece conterrà il numero dei punti che si trovano in tale finestra. A questo punto si itera nuovamente su ogni punto dell'immagine, in modo da calcolare la nuova posizione del punto p . Se il punto p_t che stiamo considerando in questo ciclo for appartiene alla finestra W del punto p , allora si aggiunge il valore di intensità di p_t a shift e si incrementa il valore di n . Dopo aver controllato tutti i punti dell'immagine per valutare se fanno

parte o meno della finestra W del punto p, si calcola la media dei loro valori di intensità dividendo shift per n, in modo da avere la nuova posizione del punto.

Nella versione a destra, invece, si inserisce un kernel (una maschera) che permette di calcolare il valore di intensità di un punto in modo pesato, in base alla distanza del suddetto punto dal punto p iniziale che vogliamo shiftare. A livello di pseudocodice abbiamo differenze solo nel ciclo for interno, in cui, appunto, andiamo a calcolare la distanza d tra il punto p che vogliamo shiftare e il punto pt. Dopo aver ottenuto la distanza creiamo un kernel w a partire da tale valore (distanza) e dalla finestra W del punto p, dopodichè nel calcolo dello shift andiamo a moltiplicare il punto pt per il peso w ottenuto dal kernel appena calcolato, mentre invece di calcolare il numero di elementi che fanno parte della finestra W, andiamo a sommare, di volta in volta, il peso di tali elementi a n. In questo modo, in pratica, stiamo eseguendo una media “ponderata”.

```

for p in f(x,y)
    while |p_outi-p_outi-1|>th
        shift=n=0
        for pt in f(x,y)
            if pt ∈ W
                shift+=pt
                n++
        p_out=shift/n

for p in f(x,y)
    while |p_outi-p_outi-1|>th
        shift=n=0
        for pt in f(x,y)
            d=dist(p,pt)
            w=kernel(d,W)
            shift+=pt * w
            n+=w
        p_out=shift/n

```



Come abbiamo detto, **in questo algoritmo non è necessario specificare il numero di cluster**, poichè viene ricavato.

L'ampiezza della finestra, ovviamente, **è fondamentale**, in quanto più è piccola e più considero valori simili tra loro.

L'algoritmo mean-shift ha **complessità $O(k N^2 d)$** , **per ogni iterazione**, dunque più pesante rispetto al k-means ma offre anche risultati migliori.

Di seguito troviamo l'**implementazione dell'algoritmo mean-shift in OpenCV**, in cui viene utilizzata una versione piramidale (pyr), dove se mettiamo maxLevel = 1

avremo esattamente il risultato del mean-shift che abbiamo appena visto, in quanto viene considerato uno spazio bidimensionale e non tridimensionale.

```
void cv::pyrMeanShiftFiltering(  
    cv::InputArray src, // 8-bit, Nc=3 image  
    cv::OutputArray dst, // 8-bit, Nc=3, same size as src  
    cv::double sp, // Spatial window radius  
    cv::double sr, // Color window radius  
    int maxLevel = 1, // Max pyramid level  
    cv::TermCriteria termcrit = TermCriteria(  
        cv::TermCriteria::MAX_ITER | cv::TermCriteria::EPS,  
        5,  
        1  
)  
);
```

Morfologia matematica (Lez.12)

La morfologia di un'immagine **descrive le forme rappresentate** all'interno di essa (dell'immagine).

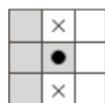
Gli oggetti sono visti come un insieme di punti del piano (pixel di foreground) e, di conseguenza, i processi morfologici sono definiti come operazioni su tali insiemi. Nelle immagini binarie gli insiemi sono membri dello spazio 2D, definiti come $Z^2(x,y)$, mentre nelle immagini in scala di grigio vengono rappresentati da un vettore 3D ($x, y, \text{intensità}$).

Elementi strutturanti

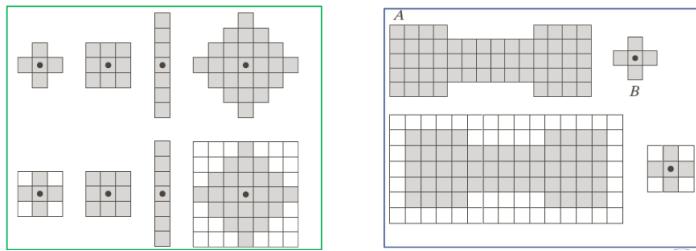
Le operazioni morfologiche sono generalmente definite rispetto ad un particolare insieme, chiamato **elemento strutturante (ES)**, che per le immagini equivale ad un array di pixel. Gli elementi strutturanti sono definiti in relazione ad un'origine e sono descritti utilizzando una convenzione per cui:

- **cella riempita** significa che appartiene all'elemento strutturante;
- **cella vuota** significa che non appartiene all'elemento strutturante;
- **croce** significa che è indifferente e non ci interessa.

Soltamente un puntino nero indica l'origine dell'ES, come nella seguente figura.

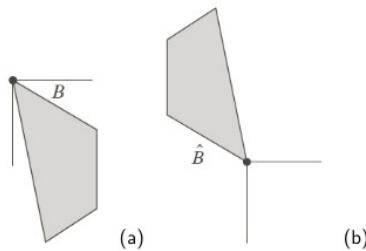


Gli ES possono avere forma arbitraria, dunque per poterli utilizzare devono essere prima trasformati in array di pixel, di forma quadrata o rettangolare. Le immagini hanno sempre una forma di questo tipo (quadrata o rettangolare), ma bisogna stare attenti al fatto che l'ES debba essere completamente contenuto nell'immagine senza fuoriuscire, ed è per questo che si utilizza il padding.



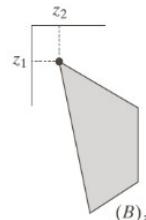
Considerando B un insieme e $z = (z_1, z_2)$ un punto d'origine, possono essere definiti gli operatori di riflessione e traslazione. La riflessione consiste nella rotazione di 180° di un ES intorno alla sua origine, con tale effetto che si ottiene invertendo il segno delle coordinate.

$$\hat{B} = \{-\mathbf{b} | \mathbf{b} \in B\}$$



La traslazione, invece, **consiste nel traslare tutti i punti dell'insieme B di un certo valore z.**

$$(B)_z = \{\mathbf{b} + \mathbf{z} | \mathbf{b} \in B\}$$



Operazioni morfologiche

Le espressioni morfologiche sono scritte in termini di ES (B) e di un insieme di pixel di foreground (A).

Erosione

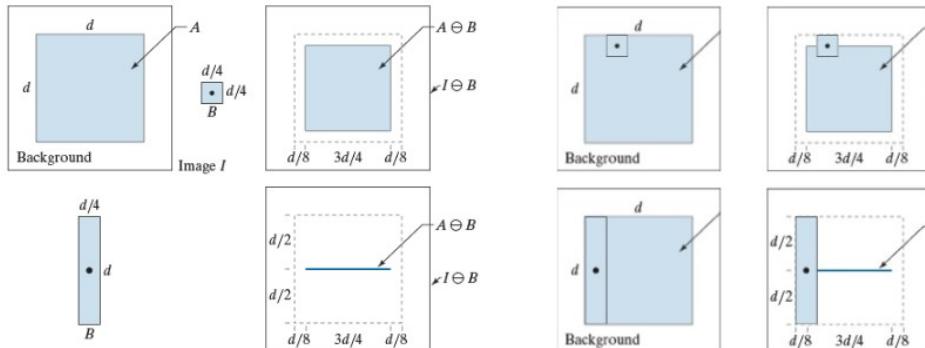
L'erosione dell'insieme di pixel di foreground A attraverso l'ES B è **definita come l'insieme dei punti z tali che B traslato di z sia completamente contenuto in A.**

$$A \ominus B = \{z | (B)_z \subseteq A\}$$

Dire che B deve essere contenuto in A equivale a dire che B non ha elementi in comune con il background e, di conseguenza, **possiamo definire l'erosione**

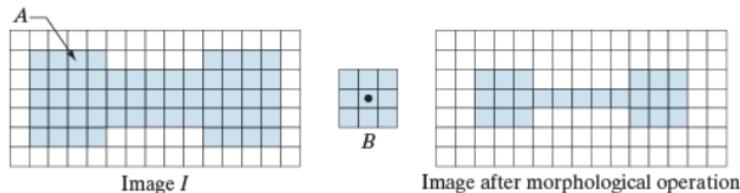
come tutti i punti di z tali che B traslato di z e intersecato con il complemento di A , ossia l'insieme dei pixel di background A^c , dia un insieme vuoto.

$$A \ominus B = \{z | (B)_z \cap A^c = \emptyset\}$$



Nel caso in cui sia A che B siano immagini, le operazioni morfologiche vengono calcolate spostando l'origine dell'ES in ogni pixel dell'immagine A , valutando se la definizione dell'operazione viene soddisfatta.

Se l'origine di B è traslata su un pixel appartenente ad A e tutti gli elementi di B sono coperti da un elemento di A , il suddetto pixel appartiene all'immagine risultante dall'operazione morfologica di erosione.



L'erosione può essere utilizzata per realizzare un **filtraggio morfologico**, nel quale maggiore è la dimensione dell'ES e maggiore sarà l'erosione. L'operazione consiste fondamentalmente in un'operazione di convoluzione in cui si fa scorrere l'ES lungo l'immagine, con conseguente cancellazione dei dettagli più piccoli dell'ES.

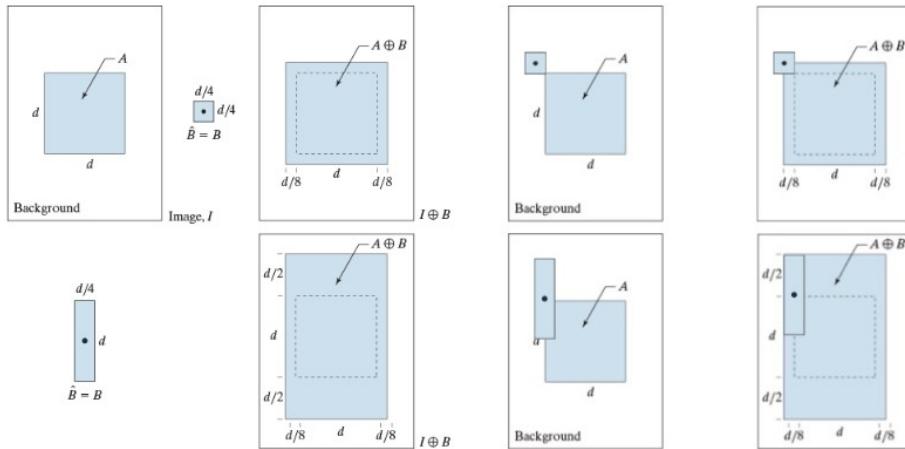
Dilatazione

La dilatazione di A attraverso B è definita come l'insieme di tutti i punti z tali che B riflesso ed A si sovrappongano almeno per un elemento, ossia la loro intersezione non deve dare un insieme vuoto.

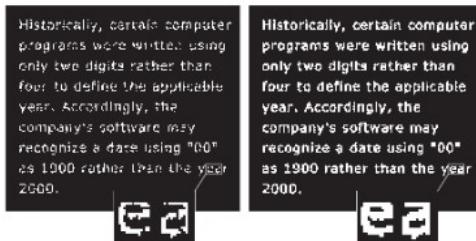
$$A \oplus B = \{z | (\hat{B})_z \cap A \neq \emptyset\}$$

Dire che B riflesso e intersecato ad A non debba dare un insieme vuoto equivale a definire la dilatazione come tutti i punti di z tali che B riflesso e intersecato ad A sia contenuto nell'insieme di pixel di foreground A .

$$A \oplus B = \{z | ((\hat{B})_z \cap A) \subseteq A\}$$



La dilatazione ha effetti simili a quelli del filtraggio passa-basso, infatti i dettagli vengono assorbiti.



Dualità

Erosione e dilatazione sono operazioni duali rispetto al complemento e alla riflessione, cioè il complemento dell'erosione di A attraverso B è uguale alla dilatazione dell'insieme dei pixel del background A^C attraverso B riflesso

$$(A \ominus B)^C = A^C \oplus \hat{B}$$

e il complemento della dilatazione di A attraverso B è uguale all'erosione del complemento di A attraverso B riflesso.

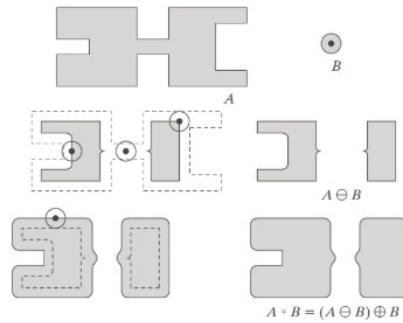
$$(A \oplus B)^C = A^C \ominus \hat{B}$$

La proprietà di dualità è particolarmente utile quando l'elemento è simmetrico rispetto alla sua origine, in modo che B riflesso sia uguale a B. In tal caso è possibile ottenere l'erosione di un'immagine attraverso B semplicemente dilatando il suo background, ovvero A^C , con lo stesso ES e complementando il risultato (viceversa per la dilatazione).

Apertura

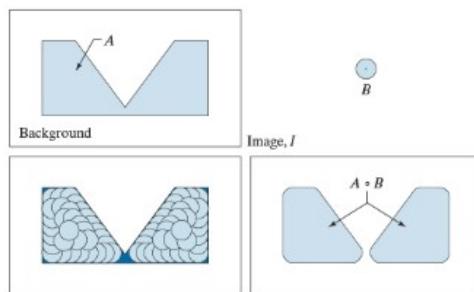
L'apertura di un insieme A attraverso B è **definita come l'esecuzione in sequenza delle operazioni di erosione e dilatazione**.

$$A \circ B = (A \ominus B) \oplus B$$



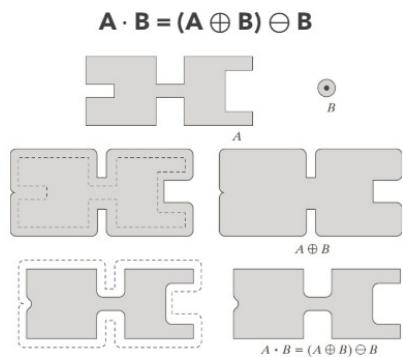
L'effetto dell'apertura è quello di rendere più omogenei i contorni di un oggetto, eliminando ponti e protusioni.

Geometricamente, l'apertura è rappresentata dai punti di A coperti dalla traslazione di B lungo il bordo interno di A.



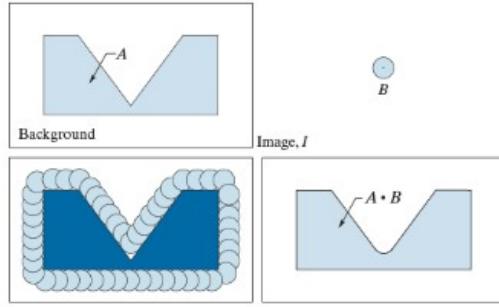
Chiusura

La chiusura dell'insieme A attraverso B è **definita come l'esecuzione in sequenza delle operazioni di dilatazione ed erosione**.



La chiusura rende anch'essa generalmente più omogenei i contorni di un oggetto ma, al contrario dell'apertura, **riempie le piccole interruzioni**.

Geometricamente, la chiusura aggiunge ad A quei punti del background che non sono coperti dalla traslazione di B lungo il bordo esterno di A.



Esempio di apertura e chiusura

L'apertura e la chiusura possono essere utilizzate per il filtraggio del rumore.

Consideriamo ad esempio l'impronta digitale affetta da rumore che si trova nella figura A.

Applicando l'operazione di erosione, il rumore esterno viene eliminato, in quanto il rumore è più piccolo dell'ES, ma quello interno aumenta di dimensione. Per questo motivo è necessaria una dilatazione successiva, in modo da recuperare la dimensione originale ed eliminare il rumore interno. Così facendo abbiamo eseguito un'operazione di apertura, data dall'erosione seguita dalla dilatazione.



L'operazione di apertura ha causato però l'interruzione di alcune linee dell'impronta. Si esegue, quindi, una dilatazione a partire dall'immagine ottenuta dall'apertura, in modo da ripristinare la continuità delle linee. Dopo l'operazione di dilatazione, le linee risultano ingrossate e, per questo motivo, si può applicare un'operazione di erosione per ripristinare gli spessori originali.



Ne consegue che andiamo ad eseguire una chiusura, data dalla dilatazione seguita dall'erosione, dopo aver eseguito un'operazione di apertura.

Trasformazioni hit or miss

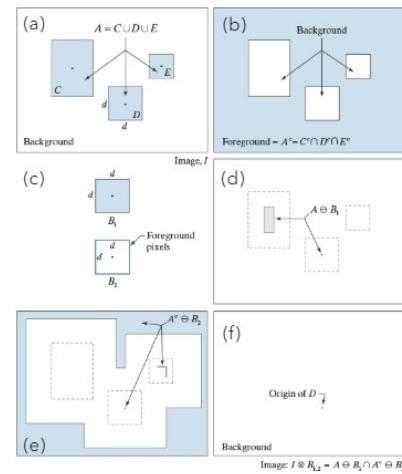
La trasformazione hit or miss **permette di individuare oggetti disgiunti all'interno di un'immagine**, ossia oggetti separati da almeno un pixel di background.

L'elaborazione è basata su due elementi strutturati B_1 e B_2 , con il primo che avrà la forma dell'oggetto da individuare ed il secondo che invece viene utilizzato per determinare le forme nel background.

Supponiamo di voler trovare la posizione d'origine dell'oggetto D in un'immagine I. La trasformazione hit or miss dell'immagine I è definita come tutti i punti z tali che B_1 sia contenuto in A e B_2 sia contenuto nel complemento di A, che equivale all'intersezione tra l'erosione di A attraverso B_1 e l'erosione del complemento A^c attraverso B_2 .

TRASFORMAZIONI HIT OR MISS

- Supponiamo di voler trovare la **posizione** dell'origine dell'**oggetto D** in I
- a) Immagine che consiste di un **foreground** che è l'**unione** A di un insieme di **oggetti**, ed un sfondo
- b) **Immagine** con il suo foreground definito come A^c
- c) ES per determinare l'oggetto D
- d) **erosione di A tramite B_1**
- e) **erosione di A^c tramite B_2**
- f) **Intersezione** di (d) ed (e) che mostra la posizione dell'origine di D

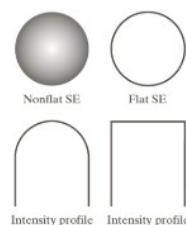


Morfologia in scala di grigio

Le operazioni morfologiche possono essere estese anche alle immagini in scala di grigio. In tal caso le immagini devono essere viste come funzioni $f(x,y)$, e non insiemi, come anche gli elementi strutturanti, $b(x,y)$.

Esistono due tipi di ES:

- **flat**, se i valori assunti sono solo 0 o 1;
- **non flat**, se i valori assunti crescono verso il centro e decrescono verso i bordi.



Erosione e dilatazione attraverso ES flat

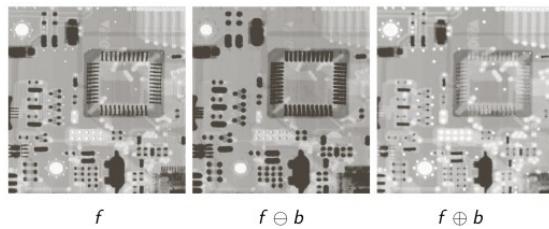
L'**erosione** di una funzione f attraverso un elemento strutturante flat b è **definita**, per ogni posizione (x,y) , come il valore minimo dell'immagine nella regione coincidente con b , quando l'origine di b si trova in (x,y) .

$$[f \ominus b](x, y) = \min_{(s, t) \in b} \{f(x + s, y + t)\} \quad \text{Simile a correlazione spaziale}$$

La dilatazione di f attraverso un elemento strutturante flat b , invece, è definita, per ogni posizione (x,y) , come il valore massimo dell'immagine nella finestra indicata da b riflesso, quando l'origine di quest'ultimo (b riflesso) si trova in (x,y) . Inoltre, visto che b riflesso è uguale a $-b$, si può definire la dilatazione anche in termini di b , come di seguito.

$$\begin{aligned}[f \oplus b](x, y) &= \max_{(s, t) \in \hat{b}} \{f(x + s, y + t)\} \\ &= \max_{(s, t) \in b} \{f(x - s, y - t)\}\end{aligned} \quad \text{Simile a convoluzione spaziale}$$

L'operazione di **erosione** rende le immagini più scure, mentre al contrario l'operazione di **dilatazione** rende le immagini più luminose.



Apertura e chiusura attraverso ES flat

Le operazioni di apertura e chiusura sono analoghe al loro corrispettivo binario, quindi l'**apertura** di f attraverso b è definita come l'esecuzione in sequenza delle operazioni di erosione e dilatazione

$$f \circ b = (f \ominus b) \oplus b,$$

mentre la **chiusura** di f attraverso b è definita come l'esecuzione in sequenza delle operazioni di dilatazione ed erosione.

$$f \bullet b = (f \oplus b) \ominus b$$

Anche in questo caso valgono le proprietà di dualità per cui il complemento della chiusura di f attraverso b è uguale all'apertura del complemento di f attraverso b riflesso e il complemento dell'apertura di f attraverso b è uguale alla chiusura del complemento di f attraverso b riflesso.

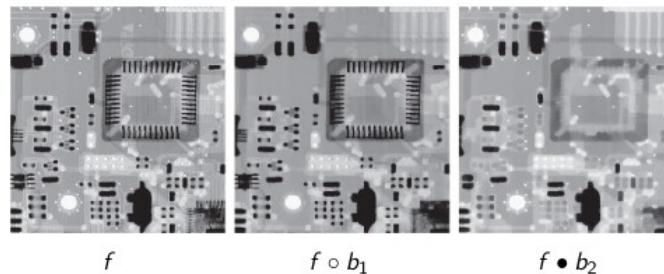
$$\begin{aligned}(f \bullet b)^c &= f^c \circ \hat{b} \\ (f \circ b)^c &= f^c \bullet \hat{b}\end{aligned}$$

Valgono inoltre diverse proprietà sia per l'apertura che per la chiusura, che vediamo elencate di seguito.

$$\begin{aligned}
 f \circ b &\leq f \\
 \text{if } f_1 &\leq f_2, \text{ then } f_1 \circ b \leq f_2 \circ b \\
 (f \circ b) \circ b &= f \circ b
 \end{aligned}$$

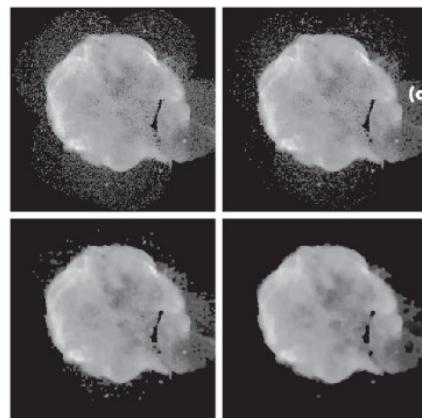
$$\begin{aligned}
 f &\leq f \bullet b \\
 \text{if } f_1 &\leq f_2, \text{ then } f_1 \bullet b \leq f_2 \bullet b \\
 (f \bullet b) \bullet b &= f \bullet b
 \end{aligned}$$

Le operazioni di apertura hanno come effetto quello di **diminuire l'intensità dei dettagli luminosi in modo proporzionale alla dimensione dell'ES**, mentre le **operazioni di chiusura** mantengono inalterati i dettagli luminosi ma, al contempo, attenuano i dettagli scuri in modo proporzionale alla dimensione dell'ES.



Smoothing e gradiente morfologico

Le operazioni morfologiche possono essere utilizzate anche per eseguire smoothing



e per ottenere risultati simili a quelli che si ottengono con il gradiente, quindi una specie di individuazione degli edge.

