

## Lab 1 Elim.

### INTRODUZIONE OPEN CV

Libreria OS per fornire un supporto alla CV nel 1999.

Inizialmente, per CV questa libreria intendeva funzioni di basso livello per fornire uno standard per l'elim a basso livello. Con la diffusione di questa libreria, sono state aggiunte altre funzioni e per CV si è iniziato a intendere algoritmi complessi che si avvicinano all'idea di interpretare il contenuto delle immagini.

Nelle ultime versioni c'è un modulo per il ML specifico con l'obiettivo di sfruttare la sinergia tra apprendimento automatico e accesso alle info delle immagini per realizzare algoritmi di apprendimento sulle immagini.

La CV consiste nella trasformazione di immagini in una **decisione** o in una nuova rappresentazione:

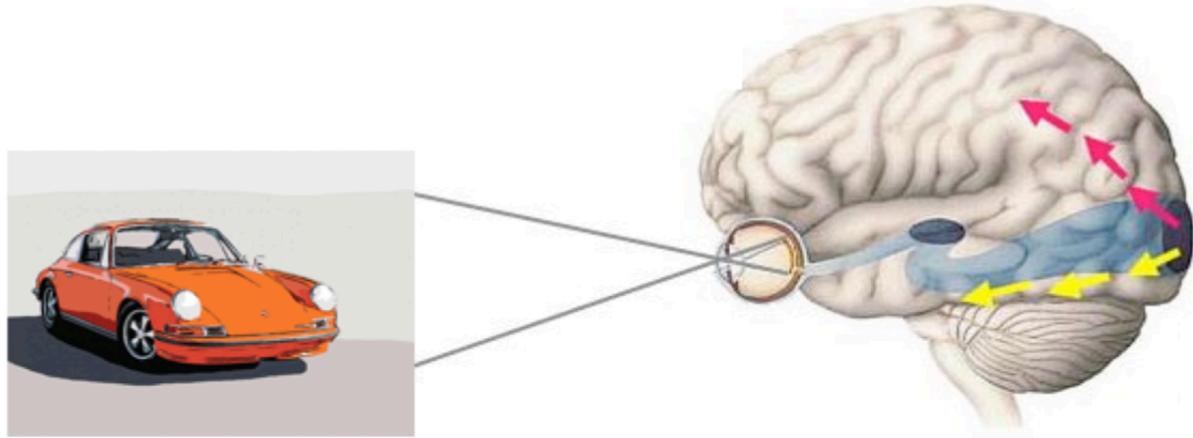
- Decisione: c'è una persona o un'auto nella scena
- Nuova rappresentazione: rimuovere imperfezioni img.

Vogliamo cercare di dare a un programma una caratteristica tipica dell'essere umano: poter vedere e interpretare il contenuto dell'ambiente in cui si trova. Dei task che all'uomo sono semplici, non sono così immediati per una macchina.

### ESEMPIO DI INTRODUZIONE

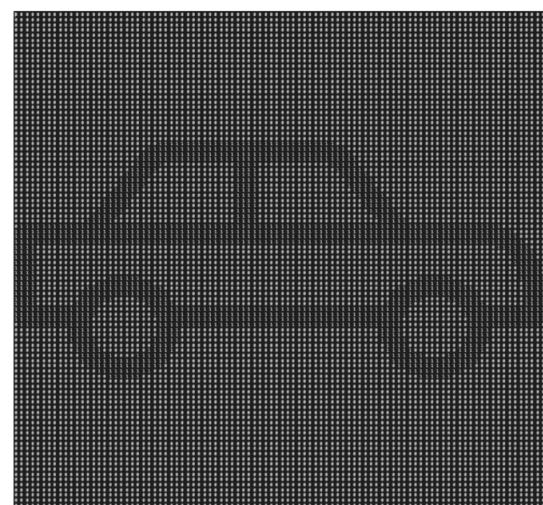
La macchina è la scena che stiamo osservando e il nostro dispositivo di acquisizione sono gli occhi che catturano l'immagine della scena che osserviamo. Questa scena è elaborata dal cervello che riesci a interpretare il contenuto. Come facciamo a sapere che quella è un'auto? Grazie alla

nostra esperienza e alle peculiarità dell'oggetto, estraendo caratteristiche comuni: 4 ruote, fari, siluette dell'oggetto...



Un oggetto è riconoscibile in un'immagine perché riesco ad individuare il suo contorno, una discontinuità tra la parte dell'immagine che sto osservando e la parte che la circonda (es riesco a individuare il faro perché != dalla carrozzeria).  
L'obiettivo è individuare le **discontinuità** dell'immagine, in modo da trovare tutti gli oggetti della scena e interpretare l'immagine.

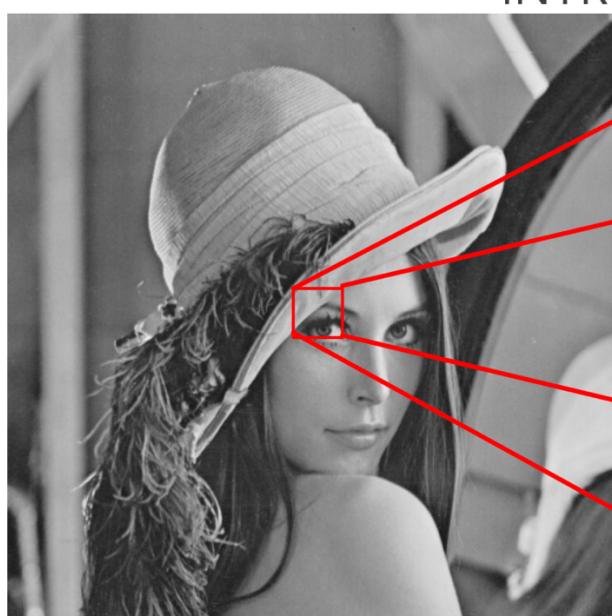
Il nostro computer vede le immagini come matrici di 0 (background) e di 1 (foreground).



Dato che il nostro computer vede le immagini come 0 e 1, un singolo pixel non dà info sulle discontinuità, ci serve una regione di pixel per capire in che contesto siamo. Se abbiamo discontinuità siamo in una regione *edge*, se abbiamo una regione di tutti 0 o 1 parliamo di una regione *flat*. Ci interessano entrambe le regioni, distinguendo dove ci sono discontinuità o meno.

Per capire dove ci sono discontinuità in un'immagine, dobbiamo scorrere tutte le regioni. Dopo lo scan di una regione, avremo un valore che trasforma l'immagine dal contenuto dei pixel a un altro contenuto che indica se quella regione ha una discontinuità o meno. L'output di questa azione è un'altra immagine con valori che indicano se ci sono discontinuità oppure no.

La situazione si complica se abbiamo immagini a scala di grigio dove i valori dei pixel vanno da 0 a 255 e non più da 0 a 1. Posso rappresentare sfumature, dettagli e scene più complesse. Questa complessità si traduce per il nostro programma in un'informazione più completa ma più difficile da trattare. In particolare, per ogni singolo pixel ho un valore tra 0 e 255.



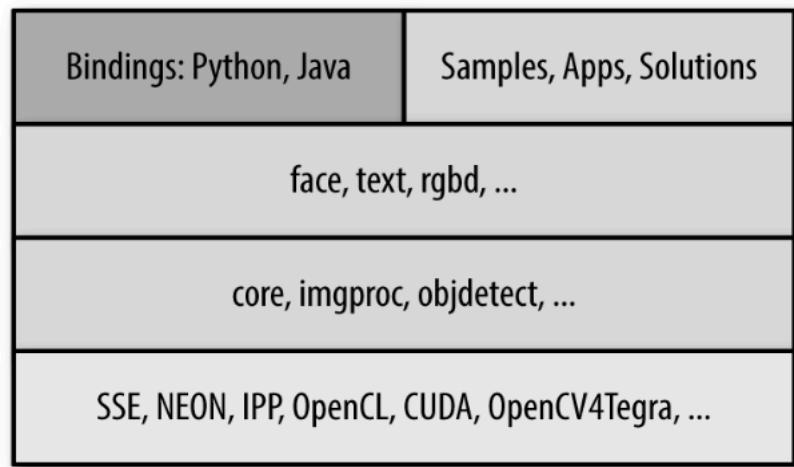
117	107	120	172	125	144	150	156	148	163	142	125	144
101	95	95	155	151	84	94	92	110	59	76	24	108
96	102	84	88	103	84	87	94	65	46	1	103	152
84	51	84	107	73	83	102	74	75	27	123	145	177
57	45	87	100	77	108	79	55	92	125	136	178	186
43	85	76	48	84	108	66	100	128	141	191	204	183
49	59	55	72	89	59	112	92	132	179	212	181	188
59	48	57	75	51	95	91	138	163	181	187	172	183
82	48	59	36	58	106	109	145	187	188	165	163	176
96	49	20	72	109	134	153	167	183	177	159	162	173
107	67	31	112	158	161	152	166	187	187	170	166	168
72	83	100	144	181	192	157	166	184	182	162	159	171
47	98	148	154	163	192	181	168	168	171	166	168	170
80	133	163	166	154	182	198	157	133	139	156	173	177
118	168	183	178	180	186	178	142	115	107	105	127	162
137	183	198	158	180	159	106	121	146	139	95	76	106
162	183	205	195	139	51	62	54	66	55	40	116	139
173	203	184	153	24	43	72	25	76	159	106	80	111
180	197	170	90	92	74	98	80	128	220	170	102	114
200	163	114	135	141	123	103	135	142	167	173	140	118
169	93	147	139	154	139	124	133	134	132	162	152	121
66	130	138	149	159	163	170	144	171	167	163	153	136
111	130	130	151	154	172	181	172	190	177	165	145	141
129	127	137	141	166	165	170	187	168	160	182	144	143

Anche qui dobbiamo definire se nella nostra regione c'è discontinuità o meno. Avremo bisogno di soglie per capire se un'area è grossomodo uniforme (ci spostiamo di pochi livelli di grigio) oppure abbiamo aree con sbalzi livelli di grigio con ad esempio 100 livelli di grigio, questo vuol dire che c'è una grossa discontinuità. Non esistono algoritmi che dicono se c'è o meno discontinuità ma ci sono quindi delle soglie che specificano le situazioni.

---

## STRUTTURA DI OPENCV

Open CV ha una struttura a livelli. Al di sotto del SO abbiamo le interfacce dei linguaggi supportati. Nel livello successivo ci sono le funzioni ad alto livello, poi quelle a basso livello e all'ultimo livello ci sono le ottimizzazioni hw.



## HEADER OPEN CV

Per accedere alle funzioni di OPENCV dobbiamo includere degli header.

La libreria base è *opencv2*.

*Imgproc* contiene le funzioni di più basso livello.

Per includere tutte le funzioni della libreria:

- `#include "opencv2/opencv.hpp"`
  - Consente di accedere a tutte le funzioni della libreria

Funzioni o classi specifiche devono essere incluse...

## PRIMO PROGRAMMA OPEN CV

Questo programma apre un'immagine e mostra il contenuto.

Dichiariamo un main c++. Successivamente dobbiamo leggere un'immagine. Il namespace delle funzioni è *cv*.

- La funzione per leggere è *imread* dove con primo argomento è il path dell'immagine e il secondo specifichiamo come aprire l'immagine (colore, scala di grigio o normale usando -1).
  - Il valore di ritorno di questa funzione è l'immagine che è di tipo *cv::Mat* che è la matrice binaria o bianco e nera o a colori.  
*cv::Mat* è una classe che generalizza il concetto di immagine e contiene info che vanno oltre la semplice matrice.
- Col metodo *empty* verifichiamo se abbiamo caricato l'immagine.
- *Namedwindow* crea una finestra dove visualizziamo l'immagine caricata. Per usarla dobbiamo dargli un nome (example 1) che è il nome di quel contenitore, il secondo

argomento è la dimensione della finestra, nell'esempio la macro è auto size.

- Per mostrare l'immagine uso la funzione ***imshow*** con primo argomento il nome del contenitore e come secondo l'oggetto da visualizzare.

Una caratteristica che useremo spesso è che ***imshow*** crea un contenitore temporaneo se non abbiamo creato un contenitore con ***namedwindow***.

- ***Waitkey*** mette in pausa l'esecuzione finchè l'utente non clicca qualcosa, se non avessi questo, il programma non aspetterebbe. È fondamentale per osservare l'output!
- ***Destroy*** distrugge l'immagine, cioè chiude la finestra.

```
#include <opencv2/opencv.hpp> //Include file for every supported OpenCV function

int main( int argc, char** argv ) {
    cv::Mat img = cv::imread(argv[1],-1);
    if( img.empty() ) return -1;
    cv::namedWindow( "Example1", cv::WINDOW_AUTOSIZE );
    cv::imshow( "Example1", img );
    cv::waitKey( 0 );
    cv::destroyWindow( "Example1" );
    return 0;
}
```

OpenCV gestisce vari formati: PNG, JPEG, PPM...

## Lab 2 Elim

Vedremo qualche funzionalità di OpenCV e un esercizio.

### IMREAD

Legge un'immagine: cioè assegna un'immagine a un oggetto `cv::Mat` che rappresenta l'immagine su cui andremo a lavorare. OpenCV gestisce diversi tipi di immagine, l'associazione è fatta sia con l'estensione che leggendo i primi byte dell'immagine. Primo argomento è il path mentre il secondo specifica come aprire l'immagine, ecco alcune macro:

- *IMREAD\_COLOR*: forza l'apertura a colori. Ogni pixel dell'immagine è rappresentata con 3 valori (*rgb*). Dobbiamo immaginare l'immagine come se fosse composta da 3 sotto immagini: una con soli valori rossi, una con blu e una con verde. Se vediamo le coordinate 0,0 di tre piani colore, avremo i 3 valori che rappresentano il primo pixel. Se inserisco questa macro per immagini binarie o a scale, comunque openCV creerà 3 matrici ma sarà uno spreco poiché replica 3 volte la stessa immagine.
- *IMREAD\_GRAYSCALE*: alloca un solo piano. Se l'immagine è a colori la trasforma in scala di grigio.
- *IMREAD\_ANYCOLOR*: interpreta il colore.
- *IMREAD\_ANYDEPTH*: interpreta la profondità. I bit che servono per rappresentare il colore o il grigio è la *profondità*.
- *IMREAD\_UNCHANGED*

### IMWRITE()

Salva sul filesystem l'immagine passata come secondo argomento

In base all'estensione del nome determina il tipo di codifica che deve essere usata. Il primo argomento è il nome, il secondo è il cv::Mat.

---

## OPENCV: STRUTTURE DATI

La rappresentazione dei dati è divisa in 3 categorie:

- Tipi di base: sono i tipi del linguaggio c++ (int, float...). In questa categoria rientrano anche dei dati strutturati di piccole dimensioni come vettori o matrici predimensionati (non allocati dinamicamente) e delle primitive geometriche
- Helper objects: oggetti che usano i tipi di base per essere definiti: oggetti di tipo range ecc...
- Large array types: c'è la classe cv::Mat che rappresenta l'immagine. Allocando un'oggetto cv::Mat rappresentiamo le immagini.

La libreria internamente usa la STL e le classi vector.

## OPENCV: CV::VEC

I fixed vector class sono dei vector di dimensione prefissata di piccole dimensioni. OpenCV fornisce degli alias per creare i vector: la sintassi prevede l'utilizzo del nome della classe (Vec), la dimensione fissata e un tipo (byte, word, signed, int, float, double)

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

Esempio Vec2i crea un vettore di 2 int.

---

## CV::MATX

Serve a creare matrice di dimensione fissata di piccole dimensioni.

L'alias ovviamente vuole 2 parametri per righe e colonne.

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

Se voglio una matrice più grande non posso usare gli alias.

---

## CV::MAT

È la classe più importante. La maggior parte delle funzioni prende o restituisce CV::MAT.

CV::MAT non è solo un array di dati ma contiene info accessorie che dipendono dal tipo di dati che stiamo memorizzando (colori, scala di grigio...). Queste info ci permettono di accedere correttamente i dati.

CV::MAT rappresenta array densi dove ogni posizione della matrice è allocata e c'è un valore memorizzato.

CV::SPARSEMAT è per le matrici grandi ma sparse.

## I DATI IN CV::MAT

I dati sono organizzati in modo *ruster*: i dati sono memorizzati per riga, in maniera sequenziale. Ogni cella rappresenta un pixel col suo valore. All'interno dell'oggetto CV::MAT avremo la prima riga, poi la seconda ecc... l'immagine è memorizzata quindi per righe in un vector monodimensionale.

Nel caso di immagini a colori, ogni pixel è rappresentato da 3 valori: l'origine in cui openCV memorizza i colori è *bgr*. Abbiamo quindi 3 valori per ogni pixel (riga colonna). Poiché vanno rappresentati in sequenziale, abbiamo tutti i pixel accodati

0.0 0.0 0.0 / 0.1 0.1 0.1 / ... ... 0. m 0. m 0. m / 1.0 1.0 1.0 / 1.1 1.1 1.1 / ... ... 1. m 1. m 1. m

Poiché in base al tipo di immagine devo accedere in maniera diversa alle immagini, l'accesso al vector è differente.

Per memorizzare queste info che determinano l'accesso ai pixel, in CV::MAT sono memorizzate delle metainformazioni che indicano quale tipo di dato è memorizzato nell'immagine:

Abbiamo:

- Flags
- Dimensioni
- Righe e colonne
- Data: puntatore all'area di memoria in cui sono memorizzati i dati.

Un altro campo fondamentale è l'array **step**: è l'offset da sommare per arrivare al pixel richiesto.

Nel caso di un array multidimensionale, ho gli indici per tutte le dimensioni. Ogni singolo indice deve essere spiazzato in base al tipo di dato della matrice: scala di grigio spiazzo il pixel di 1 valore per accedere al successivo, se ho rgb di 3 ecc... a partire da *mtx.data* (*puntatore al primo elemento*) devo sommare  $step[0]*i_0$  (indice della prima dim \* indice dello shift).

$$\& \left( mtx_{i_0, i_1, \dots, i_{N_d-1} N_d} \right) = mtx.data + mtx.step[0]*i_0 + mtx.step[1]*i_1 + \dots + mtx.step[N_d-1]*i_{N_d-1}$$

Ad esempio, se ho un'immagine *bidimensionale*:

$$\&(mtx_{i,j}) = mtx.data + mtx.step[0]*i + mtx.step[1]*j$$

Se voglio accedere all'elemento [1,1] della matrice 3x3: in step 0 avrò 3 \* numero di canali (1) che viene moltiplicato per l'indice della riga, lo stesso in step 1 per j.

## CV::MAT ALLOCAZIONE

Per creare l'immagine ho due possibilità:

- Creo una variabile vuota e alloco.
- Uso un costruttore per dichiarare l'immagine e allocarla.

Nel caso in cui creiamo un'immagine vuota successivamente dobbiamo usare ***create()*** per inizializzarla.

- Per quanto riguarda il **tipo** bisogna specificare il tipo fondamentale ed il numero di canali  
 $\text{CV}_{\{8U,16S,16U,32S,32F,64F\}C\{1,2,3\}}$
- CV\_32FC3 corrisponde ad un array di float a 32 bit con tre canali



Per creare l'immagine posso o definire la variabile **CV::MAT m** e poi usare **m.create()**.

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

cv::Mat m;

// Create data area for 3 rows and 10 columns of 3-channel 32-bit floats  
m.create( 3, 10, CV\_32FC3 );

Oppure uso un costruttore **CV::MAT** e definire **m**.

**cv::Mat m( 3, 10, CV\_32FC3, cv::Scalar( 1.0f, 0.0f, 1.0f ) );**  
**Cv::Scalar** è una classe che rappresenta un oggetto scalare. Sono 3 valori perché ho 3 canali.

## CV::MAT COSTRUTTORI

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&amp; 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&amp; s );</code>	Two-dimensional arrays by type with initialization value (size in sz)

Guardando l'immagine:

- 4° Costruttore = creo un'immagine e lo inizializzo con dati esterni. Il vector step è determinato da righe, colonne e tipo. L'array data deve essere consistente col tipo di immagine che sto dichiarando. Per definire lo *step* viene usata AUTO\_STEP.
- 5°: creo un oggetto size che contiene il numero di righe e colonne.

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

Posso creare oggetti multidimensionali. Non confondere matrice multidimensionale con matrice a più canali: la matrice che rappresenta un'immagine rgb ha 3 canali, gli array multidimensionali invece creano delle matrici in cui ho più piani separati, ogni singolo pixel è contenuto in una dimensione definita.

Per usare matrici multidimensionali devo dichiarare il numero di dimensioni, ho un array in cui in ogni posizione dichiaro la dimensione relativa e il tipo.

```
cv::Mat(  
    int ndims, const int* sizes,  
    int type  
)
```

# CV::MAT COSTRUTTORI DI COPIA

Constructor	Description
<code>cv::Mat( const Mat&amp; mat );</code>	Copy constructor
<code>cv::Mat( const Mat&amp; mat,           const cv::Range&amp; rows,           const cv::Range&amp; cols  );</code>	Copy constructor that copies only a subset of rows and columns
<code>cv::Mat( const Mat&amp; mat,           const cv::Rect&amp; roi  );</code>	Copy constructor that copies only a subset of rows and columns specified by a region of interest
<code>cv::Mat( const Mat&amp; 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&amp; expr );</code>	Copy constructor that initializes <code>m</code> with the result of an algebraic expression of other matrices

Posso usare costruttori di copia per creare altre matrici o prendere solo una parte di essa.

---

## CV::MAT ALTRE FUNZIONI

CV::MAT fornisce metodi statici per creare array di uso comune:

Function
<code>cv::Mat::zeros( rows, cols, type );</code>
<code>cv::Mat::ones( rows, cols, type );</code>
<code>cv::Mat::eye( rows, cols, type );</code>

- *Zeros()*: crea una matrice di zeri.
- *Ones()*: crea una matrice di 1.
- *Eye()*: crea matrice di identità.

## CV::MAT ACCESSO AGLI ELEMENTI

Dopo aver definito come creare le matrici, dobbiamo accedere agli elementi.

Il metodo più semplice è usare il metodo templato `at<>()`. Nel parametro devo specificare il tipo contenuto e nelle parentesi gli indici di righe e colonne.

- 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)  
)
```

Nel caso ho più canali:

```
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]  
)
```

---

## CV::MAT ACCESSO AGLI ELEMETI

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

`pt` è un oggetto di tipo point inizializzato con coordinate x,y.

Per dimensioni superiori a 3 devo fornire un vector di indici.

## CV::MAT ACCESSO ALLE RIGHE

Col metodo `ptr<>()` possiamo accedere a tutti i pixel di una riga.

## CV::MAT ITERATORI

Abbiamo due iteratori:

- `MatIterator<>`: Posso modificare i valori in cui accedo.
  - `MatConstIterator<>`: Non posso modificare.
- 

## CV::MAT ACCESSO A BLOCCHI

Possiamo accedere a un sottoinsieme dell'array e accedere a blocchi. Quando uso questi metodi, viene allocata un nuovo oggetto MAT ma i dati non sono copiati, fanno solo riferimento all'immagine originale. Quando modifco i pixel del blocco, modifco anche quelle dell'immagine originale.

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>

- `rowRange`: ho un range di righe.
- `colRange`: range di colonne.
- `Diag`: estraggo la diagonale.
  - I valori estratti non sono nuovi ma fanno riferimento all'immagine originale.

## CV::MAT OPERAZIONI ALGEBRICHE

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 <code>DECOMP_LU</code> )
<code>m0.t();</code>	Matrix transpose of <code>m0</code> (no copy is done)
<code>m0&gt;m1; m0&gt;=m1; m0==m1; m0&lt;=m1; m0&lt;m1;</code>	Per element comparison, returns uchar matrix with elements 0 or 255

Per effettuare la somma le due matrici devono avere stessa DIM.  
Stesso per la sottrazione.

Ci sono anche operatori logici, prodotti vettoriali e scalari.

## CV::MAT UTILITY

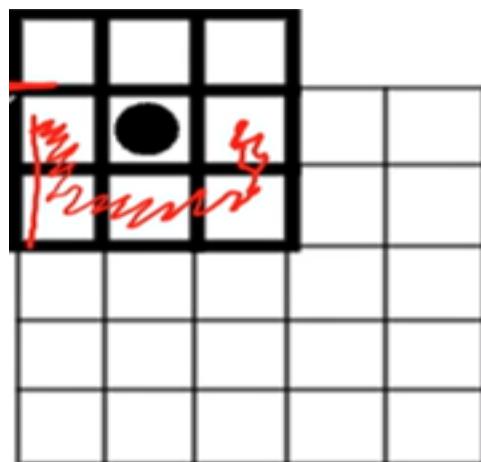
Example	Description
<code>m1 = m0.clone();</code>	Make a complete copy of <code>m0</code> , copying all data elements as well; cloned array will be continuous
<code>m0.copyTo( m1 );</code>	Copy contents of <code>m0</code> onto <code>m1</code> , reallocating <code>m1</code> if necessary (equivalent to <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 <code>mask</code> are copied
<code>m0.convertTo( m1, type, scale, offset );</code>	Convert elements of <code>m0</code> to type (e.g., <code>CV_32F</code> ) and write to <code>m1</code> after scaling by <code>scale</code> (default 1.0) and adding offset (default 0.0)
<code>m0.assignTo( m1, type );</code>	Internal use only (resembles <code>convertTo</code> )
<code>m0.setTo( s, mask );</code>	Set all entries in <code>m0</code> to singleton value <code>s</code> ; if <code>mask</code> is present, set only those values corresponding to nonzero elements in <code>mask</code>
<code>m0.reshape( chan, rows );</code>	Changes effective shape of a two-dimensional matrix; <code>chan</code> or <code>rows</code> 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 <code>s</code> at the end
<code>m0.push_back( m1 );</code>	Extend an $m \times n$ by $k$ rows and copy <code>m1</code> into those rows; <code>m1</code> must be $k \times n$
<code>m0.pop_back( n );</code>	Remove <code>n</code> rows from the end of an $n \times n$ (default value of <code>n</code> is 1) <sup>a</sup>

Una *mask* è un'immagine con la stessa DIM dell'immagine di input in cui ho i pixel 1 o 0. Se voglio estrarre un'area che è un cerchio o due cerchi: i pixel che voglio estrarre sono messi a 1.

## CV::MAT PADDING

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

Vogliamo contestualizzare il valore che stiamo analizzando in un *intorno*. Le elaborazioni tengono conto di 1 pixel e del suo intorno selezionare (ad esempio per capire se c’è una variazione di intensità). Dopo aver selezionato 1 pixel, vado a considerare il suo intorno, ad esempio un 3x3. Se il pixel selezionato si trova sul bordo, alcuni pixel del suo intorno potrebbero essere esterni all’immagine:



Se volessi accedere a questi elementi creerei un *seg fault*. Per arginare il problema, possiamo utilizzare il padding che aggiunge righe e colonne extra all'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

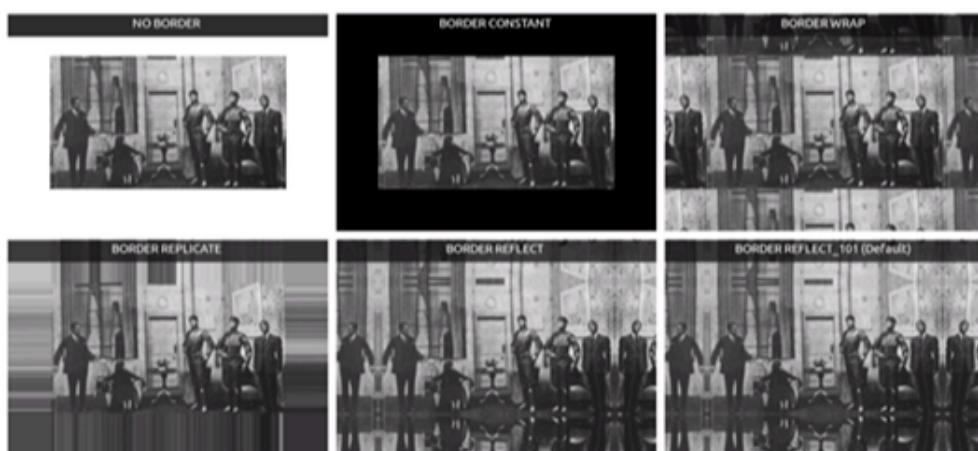
0	0	0	0	0	0	0
0						0
0						0
0						0
0						0
0						0
0						0
0	0	0	0	0	0	0

Poiché vado a riempire questi pixel con degli 0, l'operazione è detta 0-padding. A prescindere dal pixel da elaborare, tutti avranno un intorno ben definito.

In OpenCV c'è una funzione per questo: ***copyMakeBorder*** a cui passiamo l'immagine di input, l'output, quante righe aggiungere a sx, dx, sopra e sotto, e *border type* che definisce il tipo di bordo come viene creato. Se *border type* è *constant* vuol dire che tutto l'intorno è impostato a un valore costante, ad esempio 0.

***Borderwrap***: l'img si chiude su sé stessa a sx, dx, sopra e sotto.

```
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
);
```



***Replicate***: l'ultima riga, colonna e prima riga colonna vengono copiate.

***Reflect***: come se ci fosse uno specchio.

***Reflect\_101***: se consideriamo il bordo dell'immagine, l'ultima colonna, essa sarà a fianco a una sua copia quando facciamo *Reflect*. La 101 evita queste due colonne uguali vicine.

## Lab 3 Elim

Il filtraggio permette a dei valori di essere bloccati ed alcuni di essere visualizzati. È come un filtro che blocca e fa passare qualcosa. Se riusciamo a far decidere cosa far passare o meno stiamo modificando delle immagini, ottenendo delle *trasformazioni*. A fronte di blocchi e passaggi avremo diverse operazioni di filtraggio.

Siamo interessati a due tipi di variazioni che cercheremo di bloccare o far passare:

- Alte frequenze: repentinii cambi di valori nei pixel.
- Basse frequenze: variazioni intensità dei pixel graduate.

Gli effetti che vogliamo produrre usando il filtraggio sono 2:

- Repentinii (alte frequenze)
- Graduali (basse frequenze)

Queste due variazioni corrispondono a due profili che ritroveremo spesso:



Riuscivamo a vedere il faro (Lez 1) perché aveva diversi tipi di intensità. Questi repentinii cambi di intensità fanno individuare gli oggetti nella scena.

Potrei non avere un cambio repentino ma un cambio di livelli di grigio più graduale (foto 2). Il passaggio da 0 a 1 impiega un certo numero di pixel per realizzarsi.

Il nostro obiettivo è sia cercare di smussare i repentina cambi di intensità, sia riuscire a ricavare un punto in cui possiamo dire che è avvenuto il passaggio da una parte scura a una chiara. Quando acquisiamo un'immagine digitale infatti potremmo avere dei contorni con errori di acquisizione che rendono difficili operazioni citate sopra, dette filtraggio.

Se considerassimo un singolo pixel, il suo contenuto informativo sarebbe basso perché se guardo il singolo valore non sto osservando cosa succede nel suo intorno!

Se prendo un pixel  $x$  e  $z$  non riesco a distinguerli. Se considero  $x$  e il suo intorno noto che la sua regione ha una determinata peculiarità (stesso colore, stessa intensità...), se considero un pixel  $z$  che si trova sul bordo e la sua regione, vedo che quel pixel si trova su un *edge (bordo)*.

Gli edge sono essenziale per l'elaborazione dell'immagine a basso livello perché permettono di dare i contorni agli oggetti.

Nella foto 1 è molto facile denotare i cambiamenti.

Se considero lo stesso pixel e il suo intorno nella foto 2, non è mai costante la sua regione e non avrò mai una netta variazione di intensità nel suo intorno, sono in una situazione difficile: c'è una variazione di intensità nell'intorno ma è graduale. Dovremo usare tecniche che permettono di capire, in base alle variazioni, se mi trovo su un bordo o meno.

Poiché stiamo osservando un intorno, è come osservare un'area più o meno grande e posso avere quindi più o meno informazioni. Ci sono diversi approcci per “tirare” fuori info da questo secondo caso, che è più vicino alla realtà rispetto a quello dalla foto 1.

---

## FILTRAGGIO SPAZIALE

Le tecniche di filtraggio spaziale operano sui pixel di un'immagine prendendo in considerazione i valori di intensità di un intorno (*neighborhood*).

*«I livelli di grigio sono costanti o ci sono repentini cambi di intensità? »*

Applicando il filtraggio ottengo una nuova immagine che esalta o riduce i repentini cambi di variazione.

Per effettuare questa trasformazione dobbiamo considerare un pixel, stabilire un intorno e definire una regola di trasformazione. La regola di trasformazione dice come vado a modificare i pixel che si trovano all'interno dell'immagine osservando l'intorno di ognuno di essi.

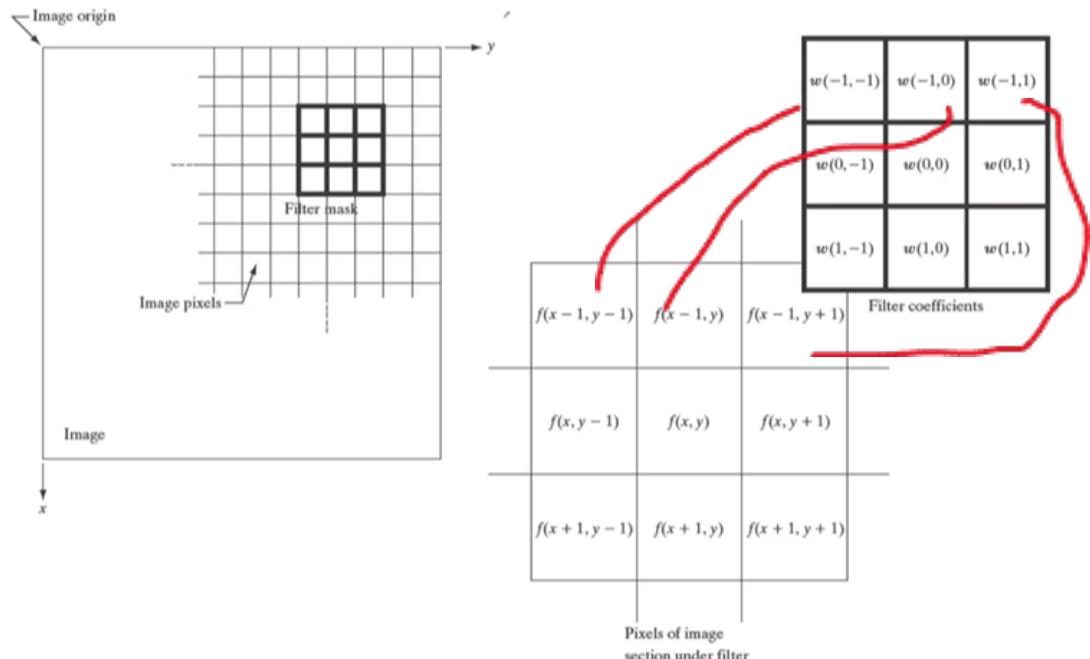
Definiamo un intorno come una matrice che si sovrappone centrata sul pixel considerato. Se considero un intorno  $3 \times 3$  di ogni pixel, sto creando una matrice  $3 \times 3$  in cui il pixel centrale è sovrapposto al pixel da elaborare. Quando la matrice si sovrappone sul pixel, le altre 8 posizioni si sovrapporranno al suo intorno. Devo riuscire a definire la regola di trasformazione inserendo dei valori nella matrice, così che i valori della matrice  $3 \times 3$  combinati coi valori dell'immagine, danno in output una trasformazione.

La funzione di trasformazione è quindi una matrice  $[n \times m]$  che di solito è una matrice dispari poiché dovendo lavorare su solo pixel e il suo intorno, avendo una matrice dispari ho righe e colonne uguali a sinistra e destra e sopra e sotto.

Nel momento in cui la funzione di trasformazione (di due variabile poiché ogni pixel ha una coordinata x e y) è definita come una matrice, posso fare una combinazione lineare tra i valori della

matrice e l'immagine e facendo ciò ottengo un unico valore che corrisponde al valore di output della trasformazione.  
Ecco perché è chiamato *filtraggio lineare*.

## FILTRAGGIO LINEARE



La maschera scorre su tutti i pixel dell'immagine. Ogni volta che lo centriamo su un pixel, ogni livello di intensità dell'immagine è sovrapposto al suo corrispondente nel filtro. Per avere in output un nuovo valore, effettuo un prodotto puntuale (ogni elemento per il suo corrispondente) ed effettuo la somma di tutti i valori.  
Sto effettuando quindi una combinazione lineare dei valori di input dell'immagine con quelli del filtro.

In generale abbiamo bisogno di un filtro (o kernel, maschera) che è una matrice di una dimensione dispari all'interno della quale inseriamo i pesi che determinano la trasformazione. Questi pesi sono moltiplicati per i corrispondenti valori dell'input. Si sommano e si ottiene il valore in output, ovvero l'immagine trasformata in base ai pesi:

- Il **pixel** nell'**immagine filtrata**,  $g(x,y)$ , è ottenuto come **combinazione lineare** dei pixel nell'immagine originale,  $f()$ , in un intorno di  $(x, y)$ :

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

Questa operazione è detta *correlazione*: progressivo scorrimento di una maschera e ogni volta che si cambia pixel, si fa l'operazione ripetuta per ogni  $x, y$ .

Un'altra operazione è la *convoluzione*: come la correlazione ma il filtro è ruotato di  $180^\circ$ . Dobbiamo immaginare che la matrice gira di  $180^\circ$  e il pixel centrale è fisso. Vengono fatte due rotazioni di  $90^\circ$ : nella prima rotazione la prima riga diventa l'ultima colonna.

---

## CORRELAZIONE 1-D

Effettuando l'operazione di correlazione di un filtro con un segnale unitario discreto, otteniamo una rotazione di  $180^\circ$  del filtro. Se i pesi del filtro fossero tutti uguali, una rotazione di  $180^\circ$  non la noterei.

---

## CONVOLUZIONE

Effettuare una convoluzione di un segnale unitario discreto (segnale con tutti 0 e un solo 1.) produce una copia esatta del filtro.

Per ottenere la convoluzione nel dominio spaziale dobbiamo preruotare il filtro di  $180^\circ$  ma se i pesi sono tutti uguali non ce ne accorgiamo poiché avremo sempre la stessa cifra.

- 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

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

---

## RAPPRESENTAZIONE VETTORIALE

Nel caso in cui non c'è un peso associato a una posizione o la posizione dei coefficienti non è importante (come se ogni peso pesasse uguale), possiamo esprimere l'operazione in forma vettoriale.  $w$  è il kernel che deve essere rappresentato in forma *stacked (righe una dietro l'altra)*. La matrice dei pesi è trasformata in un vettore.

$w_1$	$w_2$	$w_3$
$w_4$	$w_5$	$w_6$
$w_7$	$w_8$	$w_9$

---

## CORRELAZIONE IN OPENCV

- In OpenCV è possibile effettuare la correlazione usando la funzione **cv::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  
)
```

Per effettuare la correlazione in OPENCV possiamo usare la funzione *filter2D()*.

Filter2D() prende l'immagine di input, di destinazione, la profondità di output e il kernel (di tipo input array, cioè una matrice dove inseriamo i valori della trasformazione), gli ultimi 3 parametri sono lasciati di default e definiscono come posizionare il filtro sull'immagine.

---

## CONVOLUZIONE

Per effettuare le rotazioni c'è la funzione *rotate* che prende in input l'immagine di input (il filtro) e in output abbiamo il filtro ruotato e il 3 parametro è una macro.

- Per effettuare la convoluzione è necessario ruotare il filtro di 180°

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

- rotateCode:
  - ROTATE\_90\_CLOCKWISE
  - ROTATE\_180
  - ROTATE\_90\_COUNTERCLOCKWISE

---

## SPECIFICA DEL FILTRO

Per creare un filtro lineare è necessario specificare i coefficienti. Si applica la media come abbiamo visto.

Possiamo avere una funzione di trasformazione: non scrivo i valori della matrice ma ho una funzione parametrica che restituisce un valore in base alle coordinate della maschera.

Infine possiamo avere filtri non lineari dove si specifica solo l'intorno su cui applicheremo determinate operazioni. Sono filtri basati su statistiche d'ordine.

## FILTRI DI SMOOTH

I filtri di smoothing o blurring smussano il contenuto dell'img.  
Otteniamo una sfocatura che è dovuta dal fatto che stiamo  
perdendo i dettagli più fini dell'immagine.

Si perdono dettagli, si evidenziano però gli oggetti più grandi e si  
rimuove il rumore.

Lo smoothing viene spesso usata per rimuovere il rumore di  
acquisizione, cioè le imperfezioni che possono essere viste come  
oggetti ma che in realtà non lo sono.

---

## FILTRI DI SMOOTH LINEARI

- Filtri media: pesi tutti uguali.

Filtro box  $\frac{1}{9} \times$

1	1	1
1	1	1
1	1	1

- Filtri media ponderata: al centro abbiamo il peso maggiore poiché è dove ho interesse a capire cosa succede nel suo intorno. I 4-adiacenti definiscono i 4-intorno: sono i 4 pixel più vicini al centrale e pesano meno del centrale ma più di quelli agli angoli che definiscono la m-adiacenza. Peso più i pixel vicini al centrale e più mi allontano e meno pesano.

$\frac{1}{16} \times$

1	2	1
2	4	2
1	2	1

Filtro media ponderata



## EFFETTI DI SMOOTHING

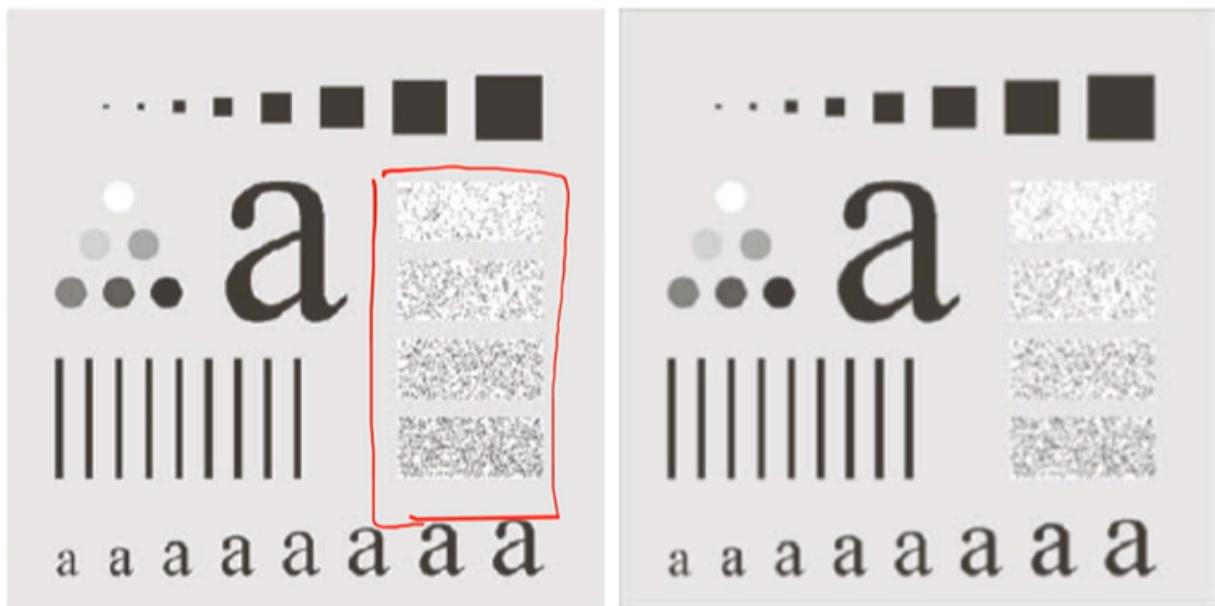


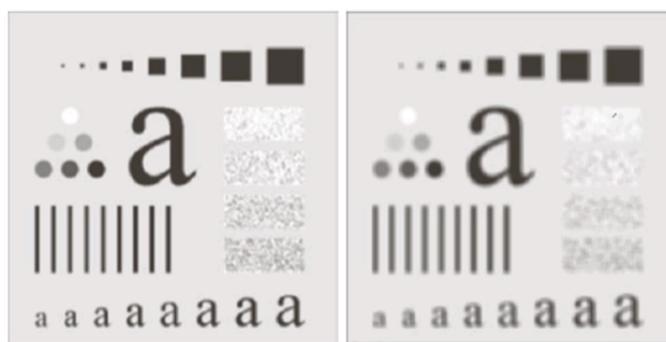
Immagine originale

Filtro media 3x3

L’oggetto contornato in rosso è una parte dell’immagine che rappresenta il rumore *salt and pepper* che potrebbe sembrare un oggetto ma non lo è.

Applicando lo smoothing già col 3x3 appiattiamo il rumore. Vediamo che i bordi dell’oggetto però sono sempre più sfocati... Gli oggetti chiari su sfondo chiaro sono poco percepibili.

Se aumento la dimensione del filtro, la sfocatura diventa sempre più forte:

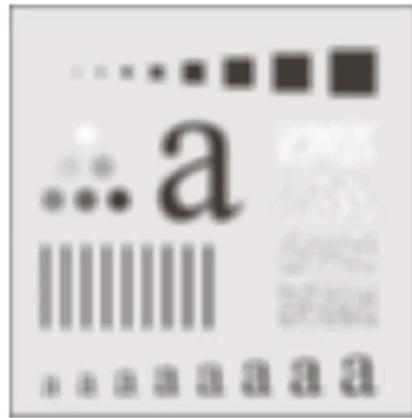


Filtro media 5x5

Filtro media 9x9

Il rumore si è appiattito ma i bordi sono sempre meno distinguibili.

È chiaro che stiamo perdendo i dettagli più fini, ma evidenzio gli oggetti più grandi perché sono quelli che resistono di più ai continui smussamenti.



Filtro media 15x15



Filtro media 35x35

Anche se non dovessi riconoscere la *a* nell'immagine, so che c'è qualcosa che possiamo estrarre, a differenza di altre figure che non sono distinguibili. Da queste info possiamo estrarre oggetti più grandi, sapendo che gli oggetti più piccoli li sto perdendo.  
Al crescere del filtro vado ad ampliare la parte da appiattire.

## SMOOTH IN OPENCV

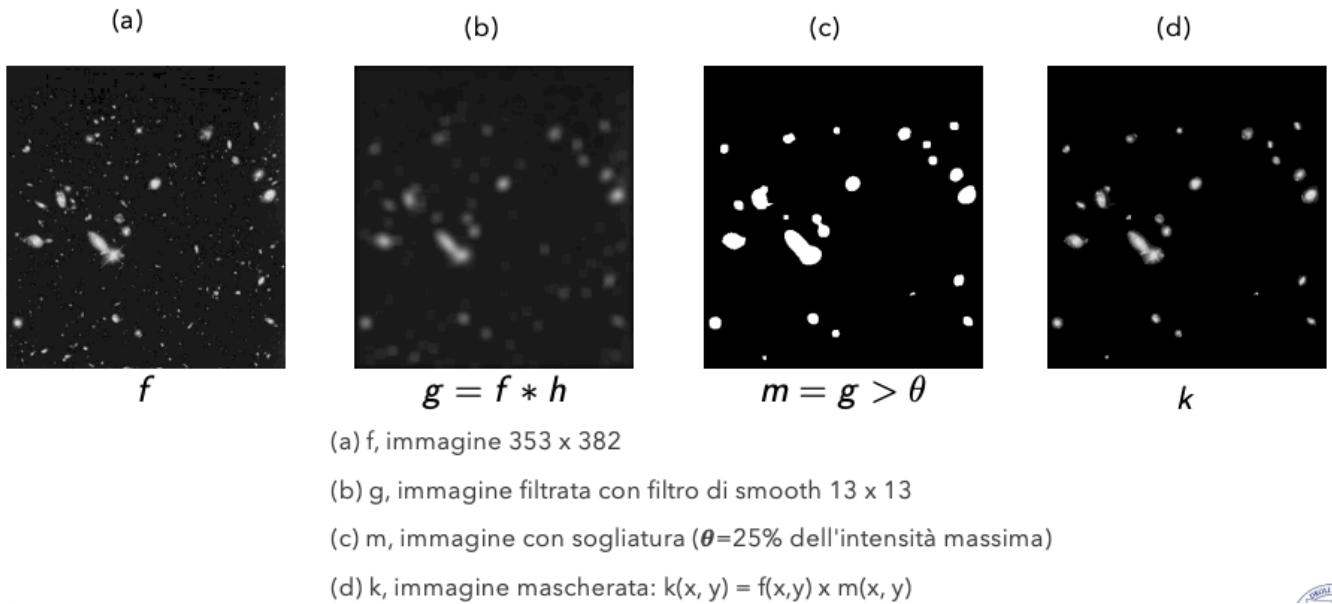
C'è la funzione *blur* e *boxFilter* che è la generalizzazione di blur.

- In OpenCV esistono diverse funzioni di smoothing

```
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
```

```
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
```

## ESEMPIO: RIMOZIONE DI DETTAGLI



Per capire quali oggetti sono *sopravvissuti* al filtro, applico una *soglia (thresholding)*. La soglia che viene applicata è del 25% dell'intensità massima dell'immagine, grossomodo 150.

Per ogni pixel dell'immagine, se il suo valore è maggiore di 150, lo metto ad 1, altrimenti lo metto a 0.

Scorro quindi l'immagine, per ogni pixel, se il suo valore è  $\geq 150$ , nell'immagine  $c$  è un pixel 1 oppure a 0.

Abbiamo creato una *maschera binaria*. Se nell'immagine  $c$  ho zero significa che quel pixel è al di sotto della soglia e ha un livello di grigio basso e quindi lo scartiamo, se è ad 1 vuol dire che ci interessa.

Immaginiamo di sovrapporre  $c$  con l'immagine di input, facciamo un prodotto puntuale. Se faccio un prodotto di un valore di grigio per 0, in output ho zero: non sto trasportando in output quel pixel.

Se invece faccio il prodotto per 1 vuol dire che porto in output quell'area che ha superato la soglia.

Facendo lo smoothing e mascherando i pixel, abbiamo estratto le aree che hanno superato l'operazione di filtraggio di smoothing.

## THRESHOLDING OPENCV

- Per effettuare la sogliatura (thresholding) in OpenCV

```
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  
);  
  
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  
);
```

IST

*Threshholdtype* è un valore che ci fa ottenere un effetto diverso in base al valore scelto:

Threshold type	Operation
cv::THRESH_BINARY	$DST_I = (SRC_I > thresh) ? MAXVALUE : 0$
cv::THRESH_BINARY_INV	$DST_I = (SRC_I > thresh) ? 0 : MAXVALUE$
cv::THRESH_TRUNC	$DST_I = (SRC_I > thresh) ? THRESH : SRC_I$
cv::THRESH_TOZERO	$DST_I = (SRC_I > thresh) ? SRC_I : 0$
cv::THRESH_TOZERO_INV	$DST_I = (SRC_I > thresh) ? 0 : SRC_I$

- Binary: Se supera la soglia la mette a MAXVALUE o a zero.

## FILTRAGGI NON LINEARI BASATI SU STATISTICHE D'ORDINE

- La risposta di questi filtri consiste
  - Ordinare i pixel contenuti nell'intorno definito dal filtro
  - Sostituire il valore del pixel centrale con un valore dell'insieme ordinato
- Esempi
  - Filtro mediano (sostituzione con il valore mediano)
  - Filtri Max e Min
  - Filtri basati su percentile

## La media non è una statistica d'ordine.

- $f = [100, 120, 98, 99, 110, 255, 100, 200, 10]$
- 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

Se sostituisco al pixel centrale il minimo, in output avremo un'immagine più scura. Di contro il massimo un'immagine più chiara.

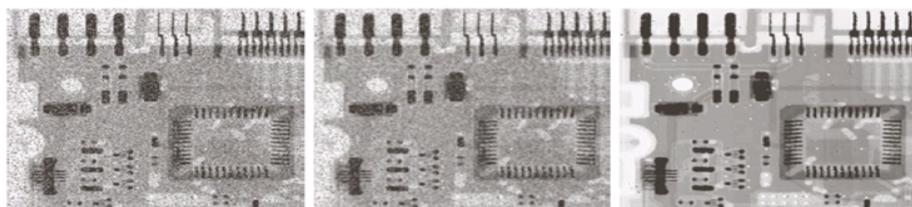
## ES: FILTRI MEDIANI

I filtri *median*i sono efficaci in presenza di rumore ad impulso.

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

(b) Immagine filtrata con filtro media  $3 \times 3$

(c) Immagine filtrata con filtro mediano  $3 \times 3$



IST

(a)

(b)

(c)

Col filtro mediano ottengo un risultato molto interessante perché riesco ad eliminare quasi tutto il rumore.

## FILTRO MEDIANO IN OPENCV

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

## Lab 4 → ELIM

Vediamo un'altra tecnica del filtraggio spaziale che consiste nello **sharpening**, cioè cercare di esaltare i contorni (**edge**).

Se da una parte l'operazione di *smooth* riduce i picchi dell'immagine che sono associati ai rumori, lo **sharpening** li esalta.

L'operazione consiste di definire un **kernel** (matrice dispari) e avere un numero di righe uguali sopra e sotto e colonne *sx* e *dx*.

---

Il valore risultante, cioè la risposta del kernel, andrà a sostituire il pixel centrale nell'immagine di output.

## SHARPENING

**Sharpening vuol dire *evidenziare le transizioni di intensità*.**

Vogliamo riconoscere un oggetto poiché è distinto grazie ai suoi bordi. Trovare i bordi, ci permette di capire dove si trova l'oggetto.

Per problemi di acquisizione, gli edge potrebbero non essere ben definiti e lo sharp esalta i bordi e li mette in risalto. È l'opposto dello *smoothing*.

Gli operatori di filtri di sharp usano il concetto di *derivata discreta* perché la derivata ci dà una misura della quantità di variazione tra un determinato insieme.

Se abbiamo 2 pixel vicini con valori diversi tra loro, la pendenza tra la loro differenza ci dirà la differenza di intensità tra i pixel: maggiore è più sarà l'evidenza dell'edge in quel punto.

Vogliamo sfruttare le derivate per ottenere una risposta proporzionale alla variazione di intensità (maggiore intensità, maggiore risposta.)

- In area di intensità costante vogliamo una variazione uguale a zero: due pixel vicini allo stesso livello.
  - Una risposta forte la si vuole quando l'intorno è centrato su un edge.
-

## DERIVATA PRIMA DI UN'IMMAGINE

La derivata prima è il concetto principale per questa operazioni. La derivata è discreta e useremo quindi un'approssimazione che conservi le proprietà della derivata prima:

- Deve essere uguale a 0 dove l'intensità è costante
- Deve essere  $\neq 0$  ogni volta che ho transizione di intensità
- Costante sulle rampe in cui la transizione di intensità è costante.

Per implementare queste proprietà usiamo un'approssimazione, cioè differenziazione spaziale: la derivata è data dall'elemento successivo meno l'elemento precedente.

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

$f(x)$  e  $(f_{x+1})$  sono due pixel uno dietro l'altro e la loro differenza ci dice la loro variazione di intensità.

---

## DERIVATA SECONDA DI UN'IMMAGINE

È implementata con le differenze centrali: eseguo una differenza in avanti e una indietro.

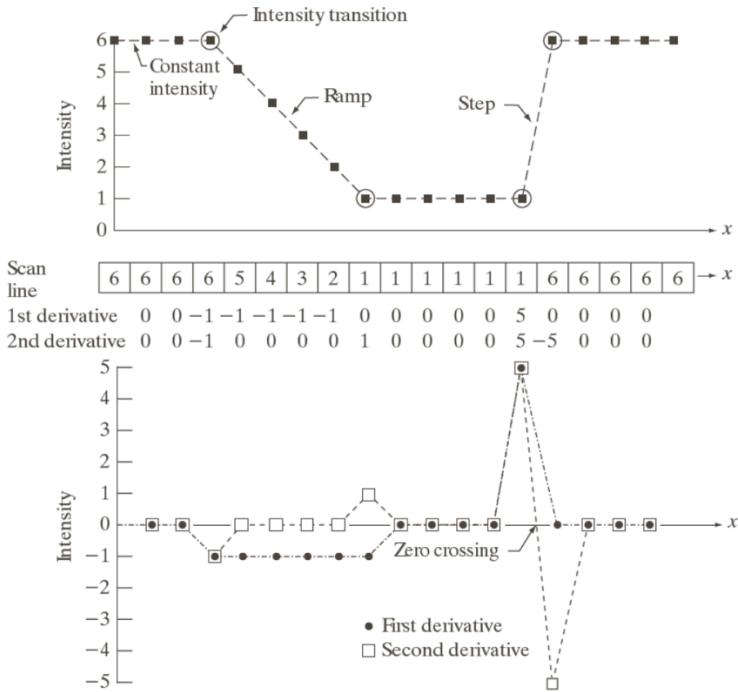
Poiché vogliamo che la derivata seconda è zero dove l'intensità è costante, se considero pixel avanti e poi indietro considero 2 volte il pixel centrale 2 volte, al pixel centrale quindi devo sottrarlo al successivo e al precedente  $(2(f_x) + (f_{x-1}))$ .

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

- Soddisfa le seguenti **proprietà**:

1. È uguale a zero dove l'intensità è costante
2. È diverso da zero all'inizio di un passo (o rampa) di intensità
3. È uguale a zero sulle pendenze costanti delle rampe

## ESEMPIO: DERIVATA DISCRETA



IST

## LAPLACIANO

È la somma delle derivate secondo lungo l'asse x e y.

È il primo operatore che useremo.

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

La risposta del filtro è indipendente dalla direzione dell'edge ed è detto **isotropico**.

A parità di variazione di intensità, la direzione dell'edge è **persa**.

- Fisso prima la colonna e mi muovo sulla riga precedente e successivo

- Fisso poi la riga e mi muovo sulla colonna precedente e successiva.
  - In un'immagine digitale, le derivate seconde rispetto a  $x$  e  $y$  sono calcolate come:
 
$$\frac{\partial^2 f}{\partial x^2} = f(x+1, y) - 2f(x, y) + f(x-1, y)$$

$$\frac{\partial^2 f}{\partial y^2} = f(x, y+1) - 2f(x, y) + f(x, y-1)$$
  - Quindi, il Laplaciano risulta:
- $$\nabla^2 f(x, y) = f(x+1, y) + f(x-1, y) + f(x, y+1) + f(x, y-1) - 4f(x, y)$$

Questi filtri devono essere a somma zero perché in aree costanti devono dare risposta zero.

Questo filtro è isotropico di  $90^\circ$ : al centro vale -4, ai bordi 1 e alle diagonali 0.

0	1	0
1	-4	1
0	1	0

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

Filtro Laplaciano invariante alle rotazioni di  $90^\circ$

*Se sommo tutto deve fare 0 poiché è costante ( $-4+1+1+1+1=0$ ).*

Come è usato il laplaciano per fare *sharpening*?

Abbiamo l'immagine di input, sommiamo il laplaciano dell'immagine calcolato pesato con una certa costante  $c$ . se  $c$  è vicino a zero, non ha nessun effetto, man mano che  $c$  aumenta, gli edge vengono sempre più esaltati: solo dove ci sono gli edge aggiungo valori più chiari.

## LAPLACIANO IN OPENCV

Prende in input l'immagine, l'output e la profondità più dei valori di default.

- Kernel size: 3 implementa laplaciano a 45°. 1 laplaciano a 90°

```
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  
)
```

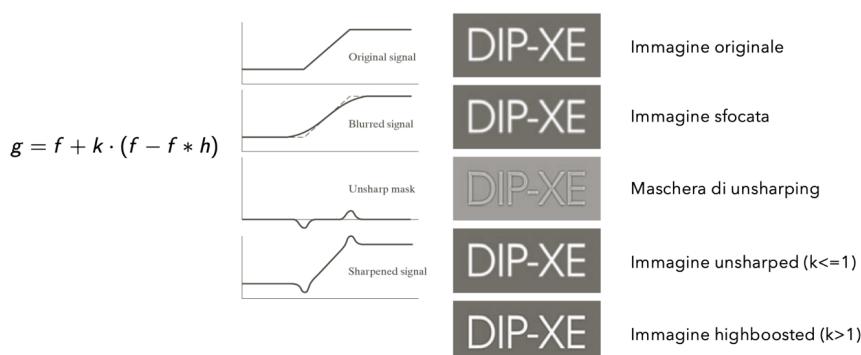
---

## UNSHARP MASK

È una tecnica di grafica più che una tecnica di elim perché presuppone che le immagini siano perfette.

L'idea è prendere l'immagine, eseguire un passo di smoothing e ottengo quindi la stessa con bordi smussati. Dall'immagine originale sottraggo quella smussata e di fatto esalto dei picchi che vado a sommare all'immagine originale.

Ottengo quindi un'immagine più nitida.



## GRADIENTE

Per quanto riguarda la derivata prima, implementa l'approssimazione del gradiente (un vettore delle derivate parziali). Quando ho un'immagine faccio la derivata *dell'asse x e y* e sono le componenti del vettore **gradiente**.

In quanto vettore, il gradiente ha un'info in più: la direzione.

Ci interessa la **magnitudo** del gradiente: più è lungo più la variazione di intensità è maggiore. La magnitudo è calcolata come la radice quadrata della somma delle componenti al quadrato.

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

$$M(x, y) \approx |g_x| + |g_y|$$

## FILTRI DI SOBEL

Considerano lungo l'asse x tutto 0 e calcolano la variazione rispetto alla riga superiore e inferiore.

- Operatori di Sobel:

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

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

$g_x:$	<table border="1"><tr><td>-1</td><td>-2</td><td>-1</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>1</td><td>2</td><td>1</td></tr></table>	-1	-2	-1	0	0	0	1	2	1
-1	-2	-1								
0	0	0								
1	2	1								

$g_y:$	<table border="1"><tr><td>-1</td><td>0</td><td>1</td></tr><tr><td>-2</td><td>0</td><td>2</td></tr><tr><td>-1</td><td>0</td><td>1</td></tr></table>	-1	0	1	-2	0	2	-1	0	1
-1	0	1								
-2	0	2								
-1	0	1								

Il valore -2 nel filtro di *Sobel* dà un effetto di smoothing per una migliore individuazione dell'edge.

## SOBEL IN OPENCV

- In OpenCV la funzione che implementa il filtro di Sobel è

```
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  
>;
```

- `xorder` e `yorder` servono a specificare l'ordine della derivata lungo la direzione x e y

Restituisce l'immagine filtrata con gli edge saltati.

## Lab 5 → Elim

Cercheremo di formalizzare come percepiamo il colore noi esseri umani e come lo descriviamo e come la libreria Open CV e le librerie di grafica in generale rappresentano queste caratteristiche degli oggetti per essere elaborate.

---

### COLORE

Il colore è un descrittore (una proprietà di un oggetto) molto forte per identificare ed estrarre gli oggetti o dettagli dalla scena. Il colore serve quindi a capire quali sono e dove si trovano gli oggetti.

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

1. Full-color: colori acquisiti da un sensore full-color.
2. Falsi colori: colori assegnati a particolari valori di intensità.

### PERCEZIONE DEL COLORE

Dobbiamo capire come gli esseri umani percepiscono il colore perché abbiamo un'esperienza che ci consente di interpretare i colori e in realtà quello che usa la libreria è simile a come osserviamo i colori e diversa da come li percepiamo. I colori sono rappresentati con  $r g b$  che mescolati danno un insieme di colori. Quando vedo una sedia di un colore non ragioniamo in termini di  $rgb$  ma in termini di colore unitario.

Newton scoprì che facendo passare un raggio di luce bianco attraverso un prisma, questo scomponiva la luce acromatica in uno spettro di colori che va dal viola al rosso.

I colori colpiscono gli oggetti che incontrano che riflettono una parte della lunghezza d'onda e assorbono l'altra.

Il colore riflette la lunghezza d'onda e assorbe le altre, ecco perché vediamo un oggetto di un determinato colore.

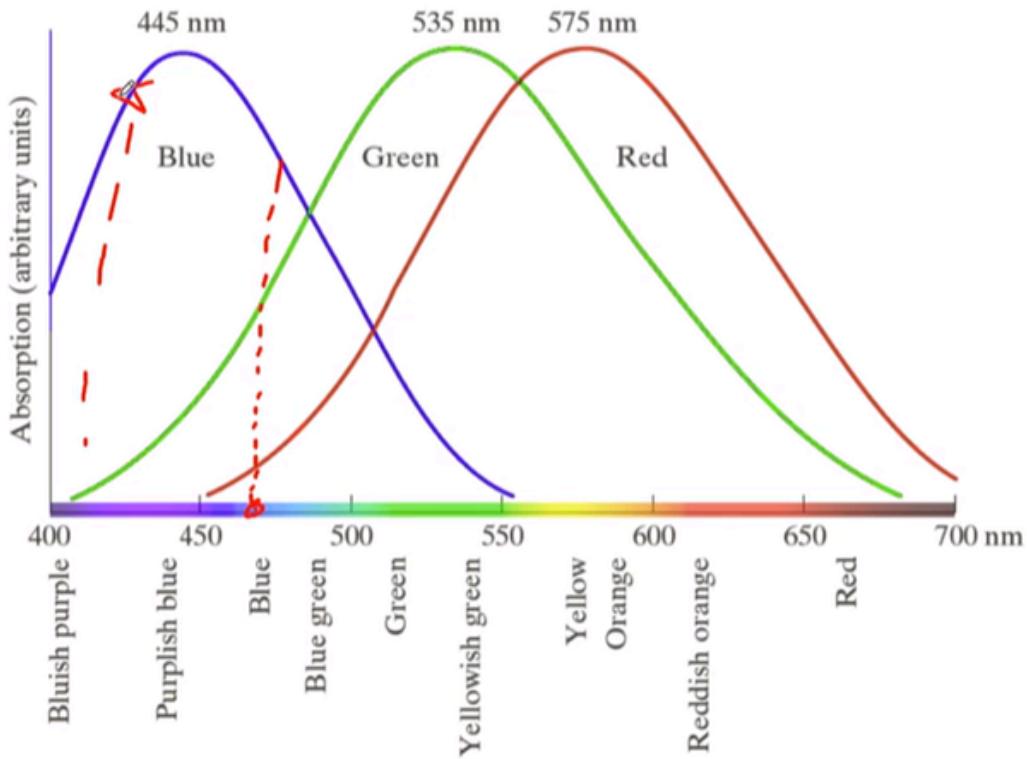
Se riflette tutte le lunghezze d'onda l'oggetto è bianco, se invece sono assorbite tutte l'oggetto è nero.

Come fa l'uomo a osservare un colore?

- La parte di luce riflessa colpisce dei *filtr* (*coni*) che all'interno dell'occhio sono divisi in 3 gruppi che percepiscono solo la luce in una determinata lunghezza d'onda. Il **65%** è sensibile alla luce rossa (molti filtri in grado di percepire il rosso). Il **33%** il verde e solo il **2%** la blu che sono la parte minore ma sono anche i più **sensibili**.

Rosso, verde e blu sono 3 colori che devono essere mescolati per dar vita ad altri colori.

Quello che vediamo come colori rgb è una diversa eccitazione dei coni che ci permettono di vedere diversi colori e tonalità.



I *colori primari della luce* (rgb) sono mescolati per dar vita a colori secondari della luce:

- Magenta (rosso + blu).

- Ciano (verde + blu).
- Giallo (rosso + verde).

I colori primari dei pigmenti, che assorbono o riflettono i colori primari della luce, corrispondono ai secondari della luce mentre i colori secondari dei pigmenti sono i primari della luce.

Se ho un colore del pigmento magenta significa che assorbo una quantità di magenta e rifletto il verde.

---

## CARATTERIZZAZIONE DEL COLORE

Ciò che caratterizza un colore sono queste 3 caratteristiche:

1. Luminosità: quanto il colore è chiaro o scuro.
2. Tonalità: associata alla lunghezza d'onda dominante.
3. Saturazione: misura la purezza della tonalità (quantità di bianco mescolato alla tonalità). Un colore puro non ha il bianco. Più bianco metto più la saturazione scende.  
All'infinito più aggiungo bianco più arrivo al bianco che è insaturo.

*Tonalità e saturazione* insieme sono dette **cromaticità**.

La quantità di rosso ,verde e blu necessarie per formare un colore sono dette *valori tristimolo (XYZ)*.

Un colore viene specificato mediante i **coefficienti tricromatici**

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

## DIAGRAMMA DI CROMATICITA'

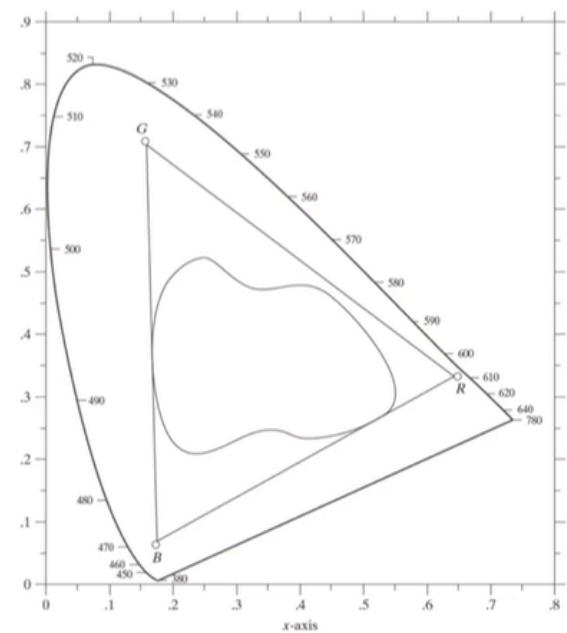
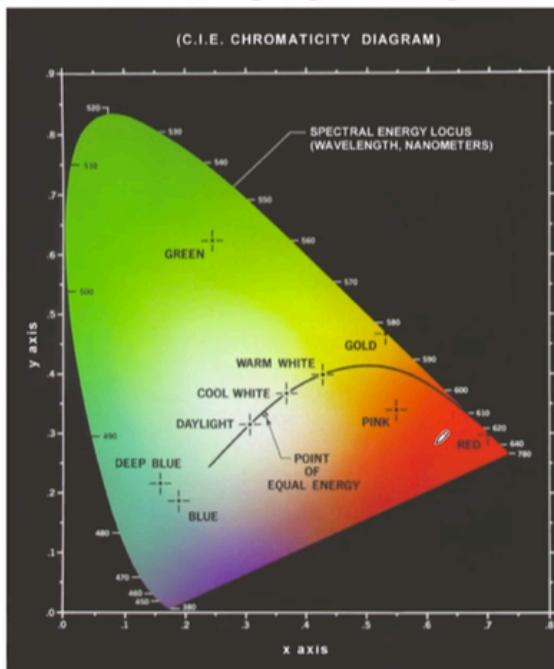
Un altro modo per specificare il colore è mediante il diagramma di cromaticità che mostra la composizione del colore in funzione di x (rosso) e y (verde).

I valori rgb che vengono usati dai monitor e schermi dà luogo a un triangolo che contiene tutti i colori che riusciamo a osservare attraverso essi.

Usando questi 3 valori non riusciamo a descrivere tutti i valori che possiamo vedere, è un limite di rgb.

All'interno del triangolo c'è un'altra area che rappresenta i colori rappresentabili sulle stampanti.

## DIAGRAMMA DI CROMATICITÀ



ST

Sul bordo ci sono i colori completamente saturi (senza bianco, tinta pura).

Presi 2 punti sul diagramma, il segmento che unisce i due punti definisce tutti i possibili colori che si possono ottenere fissato il terzo colore.

La curva nera si chiama *black body curve* che è usata per definire il calore di un colore.

## MODELLI COLORE

Dal diagramma di cromaticità che rappresenta le quantità di rgb nella rappresentazione di un colore, dobbiamo capire come nelle librerie di grafica si rappresenta un colore.

Un colore è rappresentato da un modello colore definito da un sistemi di coordinate rgb in cui un sotto spazio ne definisce tutti i possibili colori che posso rappresentare in quello spazio colore. Se ho rgb ho un sistema i coordinate cartesiane tridimensionali, il sotto spazio in cui posso rappresentare il colore è un *cubo*.

I modelli colori più usati sono:

- RGB.
- CMY e CMYK per stampanti (ciano, magenta, giallo, k→nero)
- HSI

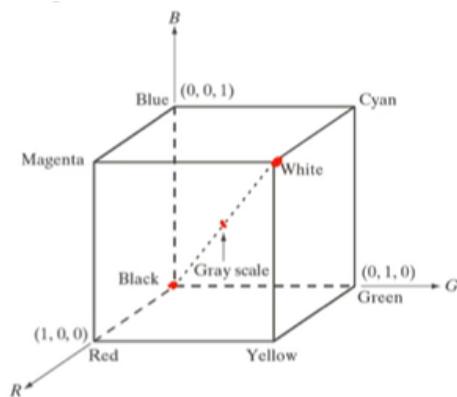
Questi spazi colore sono detti percettivamente uniformi.

## RGB

Spazio tridimensionale con asse r g b. il sotto spazio sono i cubi.

Per convenzione le componenti variano tra 0 e 1.

[0,0,0] è nero [1,1,1] è bianco. L'asse che collega il nero col bianco è la scala di grigi, le 3 componenti lungo l'asse hanno eguali valori.



Sugli spigoli abbiamo rosso, verde e blu e poi i colori secondari che si trovano all'incrocio tra 2 primari. All'interno del cubo ci sono tutti i possibili colori.

Le immagini rappresentate nel modello RGB sono formate da 3 immagini, una per ogni colore primario.

Ognuna delle 3 immagini è un'immagine a 8 bit, per cui la profondità del pixel RGB è 24 bit (*3 piani \* 8 bit*).

---

## CMY – CMYK

Ciano, magenta e giallo sono colori secondari di luce o primari di pigmenti.

Le conversioni da RGB a CMY e viceversa sono date dalle seguenti relazioni:

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

Cioè il complementare del RGB.

Uguali quantità di pigmenti primari non producono il *nero puro*, per cui nei dispositivi di stampa viene aggiunto un quarto colore, il nero (K).

---

## HSI

Sono gli spazi colori più interessanti perché sono più vicini al modo in cui l'uomo cerca di descrivere il colore.

Sono caratterizzati dalla tonalità, la saturazione e la luminosità.  
È un modello ideale per l'elaborazione delle immagini ma è costoso.

---

## DA RGB A HSI

Le immagini nelle librerie sono rappresentate con RGB e HSI però sono più vicini alla nostra rappresentazione. Dobbiamo trasformare quindi il colore da RGB a HSI.

RGB sono 3 colori e HSI sono invece 3 caratteristiche. Devo trovare un modo per convertirle!

### COME FARE:

Per determinare l'intensità a partire da un punto colore RGB, bisogna far passare un piano perpendicolare all'asse di intensità che contenga il punto colore.

Il punto di intersezione darà il valore di intensità (se il punto colore è più vicino al nero vedremo il colore più scuro).

La saturazione è data dalla distanza del punto colore dall'asse di intensità: i colori saturi (senza bianco) sono quelli sui bordi e più lontani al bianco. Il colore insaturo è il bianco. Più il punto è vicino all'asse più è saturo e viceversa.

Consideriamo un punto colore nel cubo, creo un triangolo che unisce il punto colore col nero e col bianco: immaginiamo di far ruotare il triangolo intorno all'asse bianco-nero. In base al punto colore, possiamo ottenere diverse sezioni. Per determinare la tinta prendo come riferimento il rosso e calcolo le tinte come un angolo rispetto all'angolo zero che è il rosso.

---

## ELABORAZIONI FULL-COLOR

Ci sono 2 metodi full-color:

- Ogni componente viene elaborata separatamente e combinata con le altre.

- Vengono elaborati i pixel colore (tutte le componenti sono elaborate insieme).

Smoothing e sharpening se le applico alle singole componenti o sul vettore colore, il risultato è lo stesso. Non tutte le operazioni però portano a questo risultato: se lavoro sul vettore potrei avere un risultato diverso.

---

### SMOOTHING RGB

Lo smoothing di un'immagine in scala di grigio si può realizzare con un'operazione di filtraggio spaziale e una maschera opportuna. Il valore di ogni pixel viene sostituito con la media dei valori dei pixel nell'intorno definito dalla maschera.

### SMOOTHING HSI

Se si utilizza la rappresentazione HSI, è possibile eseguire lo smoothing solo sulla componente intensità lasciando inalterate la tonalità e la saturazione.

### SHARPENING RGB

- Lo sharpening si realizza tramite l'uso del laplaciano
- Il valore del laplaciano di un vettore è definito come un vettore le cui componenti sono uguali al valore laplaciano delle componenti scalari del vettore di input

### SHARPENING HSI

Se si utilizza la rappresentazione HSI, è possibile eseguire lo sharpening solo sulla componente intensità lasciando inalterate la tonalità e la saturazione.

---

## OPENCV:: IMREAD

### ■ La funzione **imread()**

- determina automaticamente il tipo di file (BMP, JPEG, PNG, PPM, ...)
- alloca la memoria necessaria per la struttura dati (**cv::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 <sup>a</sup>	no

## SPAZI COLORE

- La funzione **cvtColor()** serve per modificare lo spazio colore con cui è rappresentata un'immagine, mantenendo lo stesso tipo
- L'array in **input** può essere a **8 bit, 16 bit unsigned o 32 bit floating point**
- L'array in **output** ha la **stessa dimensione** e la stessa **profondità** dell'array in **input**
- Il tipo di conversione è determinato dall'argomento **code**
- L'ultimo argomento è il **numero di canali** dell'immagine di output

```
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')  
)
```

## Lab 6 → Elim

Segmentare un’immagine significa partizionare le regioni dell’immagine che sono disgiunte.

L’obiettivo è individuare delle regioni i cui pixel sono accomunati da qualche condizione che ci interessa.

Nel momento in cui estraiamo una regione dell’immagine, posso usare delle tecniche di classificazione per capire il contenuto di essa. La qualità della segmentazione è fondamentale per determinare la qualità degli step successivi: più sono preciso a trovare la regione di interesse più facilita i lavori successivi.

Lavoreremo su immagini a scala di grigio, è sufficiente ciò perché il colore è un’info in più e complica l’elaborazione.

Se segmentiamo solo usando la scala di grigio, posso ricavare una maschera a partire dall’algoritmo di segmentazione, mettendo in AND la maschera con l’immagine posso estrarre la parte dell’immagine.

La segmentazione non è un task determinato, dipende da ciò che vogliamo estrarre dall’immagine...

Possiamo testare i nostri algoritmi rispetto al classificatore umano e capire quanto bene stia lavorando l’algoritmo.

Se si sfruttano le discontinuità, la segmentazione sarà guidata da bruschi cambiamenti di intensità, ad esempio gli *edge*:

- Canny
- Harris
- Hough

Se si sfruttano le similarità, si raggruppano pixel simili in base a criteri di similarità:

- Sogliatura
- Region growing
- Split and merge

- Clustering

---

## FONDAMENTI

Denotiamo con  $R$  la regione occupata dall'immagine.

La segmentazione dell'immagine consiste nel partizionare  $R$  in  $n$  sottoregioni:

- L'unione delle sottoregioni ritorna l'immagine reale.
- $R_i$  è un insieme连通的  $i$ .
- Le regioni devono essere disgiunte
- Tutte le regioni hanno un predicato  $Q$  vero
- Applicando il predicato  $Q$  all'unione di 2 regioni, il risultato deve essere falso

## ADIACENZA, CONNETTIVITA' E REGIONI

Adiacenza rispetto ai valori in un insieme  $V$ :

- Due pixel  $p$  e  $q$  seguono la 4 adiacenza se  $q$  appartiene al 4-intorno di  $p$  e hanno  $p$  e  $q$  valori in  $V$ .
  - **8-adiacenza:** due pixel  $p$  e  $q$  con valori in  $V$   $q \in N_8(p)$
  - **m-adiacenza:** due pixel  $p$  e  $q$  con valori in  $V$

Se esiste un percorso,  $p$  e  $q$  sono connessi.

Tutti i pixel connessi formano una componente connessa.

La regione è un insieme di pixel connessi.

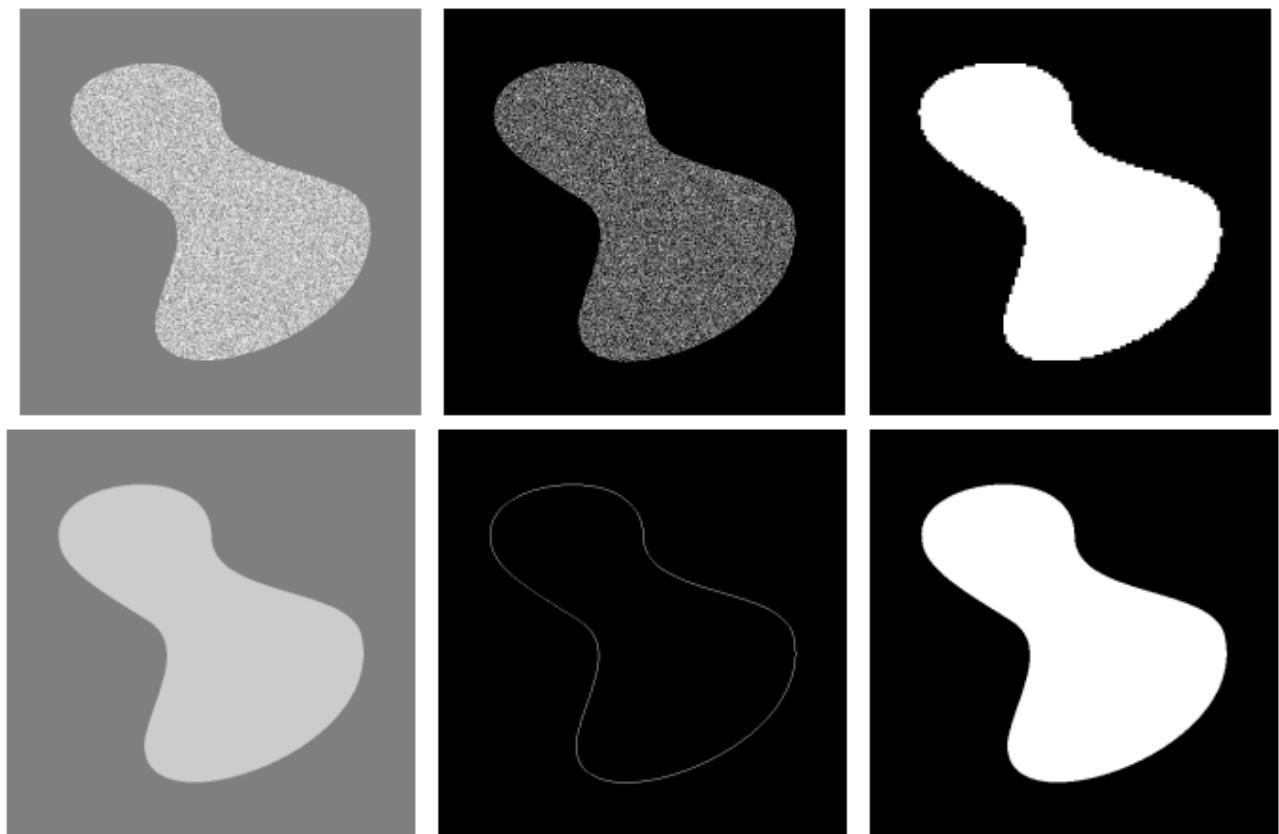
## SEGMENTAZIONE EDGE BASED

Vogliamo trovare i bordi. In un'immagine in scala di grigio, cerchiamo la discontinuità sfruttando il concetto di *intensità di un pixel*, col concetto di derivata: vediamo i pixel in cui ho una transizione da una zona più chiara a una più scura e viceversa. I pixel di discontinuità si trovano sul bordo degli oggetti.

Se abbiamo situazioni dove non possiamo sfruttare la derivata, possiamo sfruttare la similarità dei pixel: considero un pixel col suo intorno e definisco una regola: *se nell'intorno del pixel il valore di grigio non varia per 10% o n% intorno al pixel centrale, allora fanno parte della regione.*

Se verifico l'intorno di ogni pixel rispetto alla derivata, nel caso di immagini con rumore elevato, non ho una risposta puntuale: so che in quel blocco di pixel c'è una variazione più o meno forte.

[la prima ha rumore, si usa similarità, la seconda uso la derivata].



## SEGMENTAZIONE EDGE BASED

Gli *edge* sono sequenze di pixel di edge, il pixel di edge ha una forte variazione di intensità.

- **Edge**: insiemi di pixel di edge, ovvero pixel in cui si presenta una repentina variazione di intensità
- **Linee**: segmenti di edge in cui l'intensità ai lati della linea è minore o maggiore dell'intensità dei pixel della linea
- **Punti**: linee di lunghezza e larghezza pari a 1 pixel

## DERIVATA PRIMA DI UN'IMMAGINE

La derivata prima è uguale a zero in aree di intensità costante, sarà diversa da zero in caso di transizione (rampe) e sarà costante sulle rampe in cui la transizione di intensità è costante.

## DERIVATA SECONDA DI UN'IMMAGINE

Con la derivata seconda abbiamo introdotto la seguente formula:  
*“a due volte il pixel centrale sottraiamo il pixel precedente e il successivo”*.

- Soddisfa le seguenti **proprietà**:
    1. È uguale a zero dove l'intensità è costante
    2. È diverso da zero all'inizio di un passo (o rampa) di intensità
    3. È uguale a zero sulle pendenze costanti delle rampe
- 

## PROPRIETA' DELLE DERIVATE

- La derivata prima sulla rampa è caratterizzata da una doppia risposta.
- In presenza di edge a rampa, la derivata prima produce edge spessi mentre la derivata seconda produce edge sottili
- La risposta della derivata seconda in presenza di punti isolati è più forte rispetto a quello della derivata prima
- Sia su edge a rampa che su quelli a gradino, la derivata seconda ha segni opposti.

- Il segno della derivata seconda può essere usato per determinare se un edge è una transizione chiaro/scuro (derivata seconda negativa) o viceversa (positiva).
- 

## LAPLACIANO

Ricordiamo che il laplaciano è la somma delle derivate seconde parziali. Il laplaciano è usato per lo sharpening (calcoliamo il laplaciano e lo sommiamo all'immagine originale pesato con una costante per esaltare gli edge).

Il laplaciano lo abbiamo visto in 2 forme:

- Invariante alle rotazioni di 45°.
- Invariante alle rotazioni di 90°.

0	1	0
1	-4	1
0	1	0

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

Filtro Laplaciano invariante alle rotazioni di 90°

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°

Questo filtro è chiamato **isotropico**: invariante alle rotazioni.

---

## PUNTI ISOLATI

Con il Laplaciano non troviamo i bordi ma le discontinuità solamente! Possiamo notare una cosa: quando applichiamo il Laplaciano, la risposta sarà più debole rispetto a quando si applica il laplaciano sul bordo.

Per usare il Laplaciano come *edge detector* uso una soglia sulla risposta del filtro: applico il laplaciano e conservo l'edge solo se è maggiore di una certa soglia  $T$ .

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

- dove  $g$  è l'immagine di output,  $T$  è una soglia non negativa ed  $R$  è la risposta del filtro

**« c'è il valore assoluto perché il laplaciano dà due valori che potrebbero avere segno discorde »**

---

## LINEE

Nel caso di linee, che assomigliano agli edge a gradino o a tetto, la risposta del Laplaciano è doppia che deve essere gestita: o prendiamo il valore assoluto o seleziono solo i valori positivi.

Se prendo il valore assoluto, entrambi potrebbero superare la soglia quindi in uscita potrei trovare 2 risposte. L'edge in presenza di linee di una derivata seconda se prendo il valore assoluto è una doppia risposta dove non so precisamente dove passa l'edge e ho degli edge più spessi.

In alternativa possiamo non considerare la risposta negativa e prendiamo solo la positiva: l'edge è più sottile ma stiamo comunque considerando solo l'attacco della rampa, la localizzazione dell'edge lo stiamo perdendo.

---

## LINEE DIREZIONI SPECIFICHE

Il filtro Laplaciano è isotropico, cioè la risposta è indipendente dalla direzione. Per individuare rette con direzioni specifiche dobbiamo usare una “batteria” di filtri.

Usando queste batterie possiamo estrarre delle direzioni specifiche. L'output non sarà mai preciso, c'è sempre una fase di sogliatura necessaria.

---

## MODELLO DI EDGE

I modelli di edge sono classificati in base ai profili di intensità:

- Edge a gradino: transizione tra 2 livelli di intensità ad una distanza ideale di 1 pixel.
- Edge a rampa: edge sfocati e rumorosi appaiono come una transizione graduale e non netta come nel caso precedente.
- *Roof edge*: associato al bordo di una regione ha una base determinata dallo spessore e dalla sfocatura della linea.

Abbiamo visto che applicando la derivata seconda ho una doppia risposta in ingresso e in uscita dall'edge. Se prendo quindi i valori assoluti ho 2 risposte, se ne prendo 1 mi perdo l'altra parte. Devo cercare di capire dove, tra il positivo e il negativo, avviene il passaggio tra zero e uno [*zero crossing*] perché in quel punto, con elevata probabilità, passa il vero edge.

---

## EDGE A RAMPA CON RUMORE

Abbiamo detto che la derivata seconda ha una risposta più precisa e forte rispetto alla prima. Avere una risposta forte vuol dire che in presenza di rumore esalta anche le variazioni dovute ad esso.

Per ridurre il rumore possiamo applicare lo *smoothing*.

Applicando questo filtro appiattiamo i valori vicini, riducendo il rumore nella risposta delle derivate. La regola è: quando facciamo edge detection, il passo di smoothing ci sarà sempre.

---

## INDIVIDUAZIONE BASATA SUL GRADIENTE

Per quanto riguarda la derivata prima abbiamo introdotto il concetto di *gradiente*.

**Il gradiente è un vettore formato dalle derivate parziali e punta nella direzione di massima variazione.**

Puntando nella direzione di max variazione si troverà in direzione *ortogonale* rispetto all'edge ( $90^\circ$ ).

La magnitudo è l'intensità della variazione ed è la radice quadrata della somma delle componenti al quadrato.

Come facciamo a trovare la direzione del gradiente? A partire dalle due componenti possiamo trovare la direzione id max variazione come l'arcotangente di seno su coseno.

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

Così ho l'angolo alfa che rispetto all'asse x mi determina la direzione del vettore gradiente.

Le due componenti del vettore gradiente sono calcolati con i filtri di *Sobel*.

#### INDIVDUAZIONE BASATA SU GRADIENTE

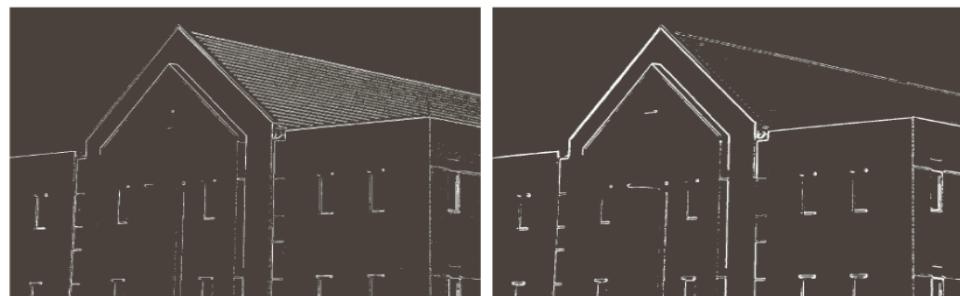
- Per enfatizzare gli **edge diagonali**, è necessario utilizzare le mascher specifiche




---

#### GRADIENTE E THRESHOLDING

- Per ottenere **immagini gradiente** più "pulite", è possibile utilizzare anche la tecnica del **thresholding**, ovvero utilizzare solo i pixel la cui risposta supera una determinata soglia



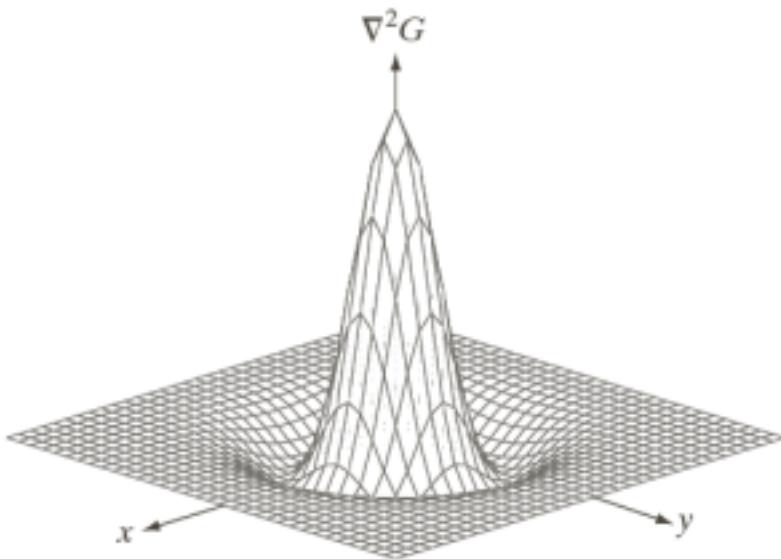
## FILTRO LOG

Un algoritmo che sfrutta la tecnica di smoothing e il concetto di derivata seconda è il filtro *Log*. L'idea è: devo fare smoothing, per estrarre gli edge applico il laplaciano e quindi faccio il laplaciano della funzione Gaussiana (campana), il filtro che ottengo lo applico a tutta l'immagine e ottengo nell'intorno di ogni pixel il doppio effetto di smussare e trovare gli edge contemporaneamente.

È la derivata seconda della Gaussiana rispetto a x + quella rispetto a y.

La funzione gaussiana è determinata da *sigma* che è l'ampiezza della campana (più piccola più smoothing è ridotto e viceversa).

La forma di questo filtro è tipo un *sombrero*.



Quello che otteniamo come filtro da applicare all'immagine è un filtro 5x5.

L'effetto è di smussare e poi applicare il laplaciano per estrarre gli edge.

Per ottenere questo filtro come regola empirica, la dimensione del filtro *n* deve essere 6 volte la deviazione standard:

- La dimensione del filtro  $n$  deve essere scelta pari al più piccolo intero dispari maggiore o uguale  $6\sigma$
- Es.:  $\sigma = 4 \ n = 25$

[ricordiamo che vogliamo filtri dispari perché possiamo centrarli].

Per ottenere questo filtro usiamo *getGaussianKernel* in *OCV*  
In OpenCV è possibile utilizzare la seguente funzione

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

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

Una volta applicato questo filtro, in ogni caso manteniamo il problema del laplaciano, cioè la doppia risposta.

Per eliminare il rumore dobbiamo fare 2 operazioni:

- Scegliere qual è il pixel che rappresenta l'edge. Ricordiamo che il laplaciano ci dà la doppia risposta (pos, neg) e in mezzo ai due valori ci sarà il picco che corrisponde al vero edge. **Lo zero crossing** è il punto in cui il laplaciano passa per lo zero e va verso i valori negativi o viceversa.  
Centreremo il nostro edge nel punto centrale in cui la derivata attraversa lo zero.
- Una sogliatura che determina cosa passa o meno

## FILTRO DOG

A causa di rumori e acquisizione, potremmo non trovare esattamente lo zero. Nel caso reale non potrebbe esserci lo zero e quindi introduciamo un filtraggio dell'immagine di output del LOG in cui andiamo a studiare l'intorno di ogni pixel per capire se c'è questa variazione e in che direzione si realizza.

Consideriamo un intorno  $3 \times 3$  centrato nel nostro pixel, considerando di aver già usato LOG, devo posizionare l'edge nel punto in cui si attraversa lo zero. Vogliamo trovare una coppia di pixel in cui ho una differenza che superi una certa soglia. Non è necessario che attraversi lo zero ma vogliamo solo che calcolando la differenza tra 2 pixel adiacenti, sia superiore a una soglia. Questo algoritmo è chiamato **Dog** (*difference of Gaussians*).

## Lab 7 -> Canny e Harris

### CANNY EDGE DETECTOR

L'algoritmo di Canny è uno dei migliori per individuare gli **edge**. Ha una formulazione fortemente teorica: l'idea parte da un'analisi del rilevamento degli edge e su questa ne costruisce un algoritmo ottimale rispetto a 3 obiettivi fondamentali:

1. Basso tasso di errore: avere un basso numero di falsi positivi, cioè l'algoritmo deve sbagliare poco (se dice che un punto è un edge ma non lo è) e di falsi negativi. Falso positivo vuol dire che l'algoritmo trova gli edge ma non sono reali...
2. Punti di edge ben localizzati: gli edge rilevati devono essere vicini ai punti reali.
3. Risposta puntuale per edge singolo: restituire un solo punto di edge per ogni punto di edge.

Canny definì la derivata prima della Gaussiana come un buon approssimatore di *edge detector*.

Per avere una risposta dal filtro in tutte le possibili dimensioni dobbiamo applicare una batteria di filtri, in modo da avere una risposta per ogni direzione. Abbiamo visto che se sfruttiamo l'approssimazione della derivata prima lungo gli assi e calcoliamo il gradiente, sappiamo che esso punta verso la massima variazione. Ciò significa che l'edge che passa per quel punto sarà perpendicolare al punto di massima variazione.

---

### FUNZIONE GAUSSIANA

Canny propone di avere la derivata prima di una Gaussiana (serve per fare smoothing). Preso il filtro gaussiano facciamo convoluzione con l'immagine e poi derivata prima lungo i due assi (calcolando il gradiente) col filtro di Sobel e calcolo la magnitudo.

- Sia  $f(x, y)$  l'immagine di input e  $G(x, y)$  la funzione Gaussiana

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

- Si effettua la convoluzione delle due funzioni

$$f_s(x, y) = G(x, y) \star f(x, y)$$

- E successivamente si calcola la magnitudo e la direzione del gradiente

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

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

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

---

L'algoritmo di canny cerca di sfruttare queste info per capire dove si sta sviluppando l'edge.

## NON MAXIMA SUPPRESSION

L'immagine Magnitudo presenta picchi intorno ai massimi locali per cui è necessario sopprimere questi valori. A tal fine si considera un numero finito di orientazioni.

Se consideriamo la magnitudo, trovo che potrò avere degli edge più o meno forti. Se ruotiamo l'immagine per vederla di profilo avrò dei picchi alti. La prima cosa di canny è chiedersi se tutti gli edge che trova sono reali, può essere che vicino a un edge forte potrebbero esserci edge meno forti. Canny considera un intorno 3x3, in un intorno del genere, gli edge che possono passare per il pixel centrale sono: **orizzontali, verticali, + 45° e -45°**.

Consideriamo il pixel centrale: sappiamo che la direzione del gradiente è ortogonale alla direzione dell'edge, se lungo la direzione dell'edge trovo dei pixel che hanno una magnitudo

maggiore di  $x,y$ , significa che il vero edge è quello con la magnitudo maggiore nella stessa direzione dell'edge.

Quindi prendiamo le possibili direzioni del gradiente, sappiamo che l'edge è *ortogonale* rispetto al gradiente quindi vado a verificare se lungo la direzione del gradiente ho dei pixel che hanno una magnitudo più forte, se lo trovo metto a zero il pixel in posizione  $x,y$ , eseguendo una *soppressione dei non massimi* (*NMS*). Considero l'intorno  $3x3$  del pixel  $x,y$ , immaginiamo che il gradiente abbia una direzione orizzontale quindi l'edge sarà in verticale quindi abbiamo 3 pixel (alto, basso e centrale) per vedere se hanno una magnitudo maggiore del pixel in posizione  $x,y$  (quello centrale).

## THRESHOLDING CON ISTERESI

Facendo questa operazione ottengo i picchi relativi ai pixel di edge “sopravvissuti”. Ora dobbiamo decidere quali sono quelli da considerare pixel di edge e quali no, applicando una *soglia*.

Se uso una sola soglia, tutti i pixel che hanno una magnitudo minore verranno messi a zero e gli altri verranno segnati come edge. **Come faccio a decidere questa soglia unica?** Se la soglia si abbasso ho più falsi positivi, se la alzo più falsi negativi...

L'idea di canny è usare **due soglie: una alta e una bassa**.

Tutti i pixel che hanno una magnitudo sopra la soglia alta sono pixel forti e passano, tutti quelli che hanno una magnitudo sotto una soglia bassa si scartano. Usando due soglie si crea una **regione di incertezza** dove ci sono valori dove non sappiamo se sono veri edge o no ma possiamo dire che se un pixel ha un valore maggiore della soglia alta è un edge, se un pixel è compreso tra soglia alta e bassa potrebbe essere un edge; come facciamo a dirlo se lo è effettivamente? Andiamo a vedere il suo intorno: se nel suo intorno trovo pixel di edge forte allora lo considero perché sto immaginando che in quel punto c'è stato un abbassamento di intensità ma è un edge.

Quindi se nell'intorno di un pixel che è compreso tra alto e basso trovo un pixel con valore alto allora promuovo quel pixel a forte, se invece nell'intorno trovo dei valori bassi, vuol dire che è rumore di un non pixel e quindi lo azzero.

CANNY: Ricapitolando:

Applico un filtro gaussiano, calcolo con Sobel la derivata lungo gli assi avendo un gradiente e calcolo magnitudo e orientazione (angolo di fase). Poi NMS: considero l'intorno  $3 \times 3$  del pixel  $x,y$ , vedo la direzione del gradiente e vedo se lungo la direzione dell'edge ho dei pixel che hanno una magnitudo maggiore. Se ne trovo almeno 1,  $x,y$  va a zero perché è un non massimo.

Fatto ciò, si potrebbero creare dei buchi nei possibili edge, per trovare una soglia ottimale ne prendo due: una alta e una bassa. Tutti i pixel con magnitudo minore della bassa vanno a zero e quelli con magnitudo maggiore della soglia alta vanno a 255, per i pixel compresi vado a studiare l'intorno. Se nell'intorno del pixel trovo un pixel forte allora viene promosso a 255, altrimenti viene azzerato.

---

## HARRYS CORNER DETECTION

È un corner detector: gli angoli si formano all'incrocio di pattern di intensità: dobbiamo immaginare edge che si incontrano e creano dei corner.

I corner sono importanti perché definiscono i punti di giunzione degli edge ed è stato visto che sono caratteristiche che si trovano negli oggetti per identificarli. Ad esempio nei video individuare i corner è usato per tracciare gli oggetti che si muovono o per riconoscerli attraverso tecniche di classificazione.

## IDEA

L'**idea** è quella di avere una finestra che scorre sull'immagine intera. Questa finestra è un intorno. Se la finestra si muove in un area di intensità costante la variazione sarà pressoché nulla.

Se invece prendo un edge, la variazione di intensità che trovo nella finestra si sviluppa lungo un'unica direzione che è quella ortogonale al gradiente. Se spostando la finestra trovo una variazione in unica direzione ho trovato un edge. Se invece spostando la finestra prendo un corner vuol dire che si sposta in più direzioni.

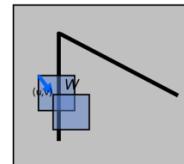
Quindi:

- No variazione: zona flat.
- 1 variazione è un edge.
- Più variazioni è un corner.

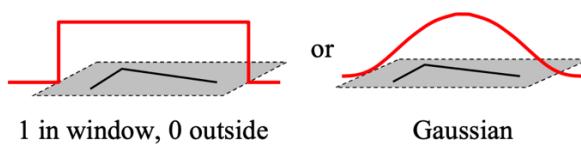
## CARATTERIZZAZIONE DI UN CORNER

- Considerando un piccolo spostamento  $[u, v]$

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



- La funzione **w()** è utilizzata per **assegnare** un **peso** ad ogni pixel dell'immagine



Con

Questa formula calcola la variazione di intensità all'interno della finestra.

La quantità nella parentesi quadra non è altro che il *gradiente*<sup>2</sup>.

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

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

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

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

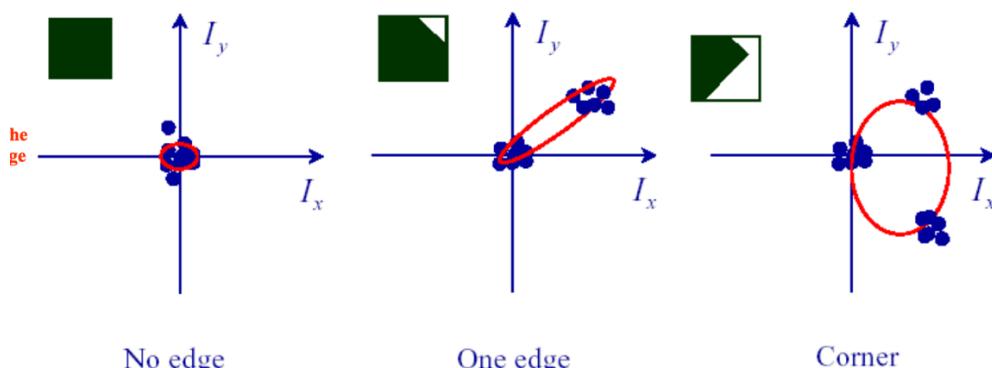
E è simmetrica e quindi posso trasformare la matrice nel prodotto di tre matrici con una SVD:  $[UST^T]$ :

- U (matrice in cui le colonne sono gli autovettori di E)
- S (ha sulla diagonale gli autovalori)
- $U^T$ .

La matrice E indica la variazione di intensità.

Geometricamente gli autovettori di una matrice rappresentano la direzione in cui si sviluppano le variazioni.

- Se le variazioni sono di bassa intensità siamo in zona flat.
- Se invece si sviluppano lungo 1 direzione siamo su un edge.
- Se si sviluppano su più direzioni siamo in un corner.



## OTTIMIZZAZIONE

Calcolare la SVD per ogni pixel costa molto computazionalmente. Notiamo però che i due **autovalori** della matrice E sono legati alla **traccia** e al **determinante** della matrice E.

- Per **ridurre** il **costo computazionale** è possibile utilizzare degli indici più veloci da calcolare

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

- Infatti

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

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

- Se  $R \gg 0$  nell'intorno è presente un **corner**

Invece di calcolare la SVD ci calcoliamo il **determinante** e la **traccia** e tiriamo fuori un **indice** R che è legato agli autovalori della matrice. Se R sarà grande sarà un corner, se è piccolo sarà una flat, perdiamo però la direzione (per sapere se siamo su un edge).

Quindi dopo aver calcolato la matrice E, calcoliamo il determinante e la traccia e calcoliamo R.

- determinante =  $C_{00} * C_{11} - C_{01} * C_{10}$
- traccia =  $C_{00} + C_{11}$
- $R = \text{determinante} - k * \text{traccia}^2$

K ci dice quanto incide la traccia alla fine.

Ricapitolando Harrys:

- Smoothing.
- Derivate parziali lungo gli assi.
- Calcoliamo  $Dx^2$  e  $Dy^2$  che sono le **componenti** della matrice.
- Applichiamo un filtro gaussiano.
- Calcoliamo **determinante** e **traccia** e poi l'**indice**  $R$ .
- Normalizziamo R in [0,255].
- Sogliamo R (sono *corner* tutti i valori che superano una soglia scelta).

## Lez 8 -> HOUGH

Restiamo ancora nell'ambito della segmentazione basata sulla **discontinuità**. Vedremo un algoritmo che non trova le discontinuità ma **usa** le discontinuità trovate con Canny per trovare all'interno dell'immagine delle forme descritte da una funzione caratterizzata da parametri.

Andremo a cercare delle forme descritte attraverso funzioni parametriche.

La trasformata di Hough può essere applicata ogni volta in cui possiamo descrivere una forma con una funzione parametrica ma ci focalizzeremo su rette e cerchi. È possibile trovare configurazioni di pixel che si dispongono lungo rette e circonferenze.

A partire dall'immagine dovremo trovare i punti di edge perché se voglio trovare una retta, i punti appartenenti ad essa li vediamo perché sono *edge*. Troveremo quindi i pixel che corrispondono agli edge, di tutti questi punti vogliamo trovare quelli che si dispongono lungo rette o circonferenze. La trasformata di **Hough** ci serve per questo.

Rispetto agli operatori passati, che erano **puntuali** poiché consideravano un intorno, La trasformata di Hough è un operatore **globale** perché non opera su un intorno di ogni pixel ma analizza le sue proprietà globali: se definisco un punto di edge non posso definire una retta o una circonferenza solo con questo. Se prendo un solo punto, potrebbe appartenere a infinite rette o circonferenze!

È chiaro che dovremmo trovare globalmente tutti i punti che appartengono a rette, circonferenze, ecco perché è *globale*.

Essendo un operatore globale non distingue un singolo cerchio ma li trova tutti, useremo *raggio e circonferenza* per sogliare l'immagine.

## IN BREVE HOUGH:

Data un'immagine, estraggo gli edge con canny, questo è l'input dell'algoritmo.

**[domanda d'esame]** Perché devo lavorare sugli edge? Se non ho discontinuità (*flat*) non abbiamo informazioni quindi devo vedere le discontinuità (possibili forme nell'immagine) e tirare fuori solo quelle che interessano, cioè rette e cerchi.

Dopo aver trovato gli edge voglio trovare quali pixel si dispongono lungo rette o circonferenze. Mediante una soglia sceglieremo quali rette o circonferenze prendere. Nonostante gli edge si dispongano con errori, la trasformata di Hough riesce a trovarli.

---

## HOUGH PER RETTE

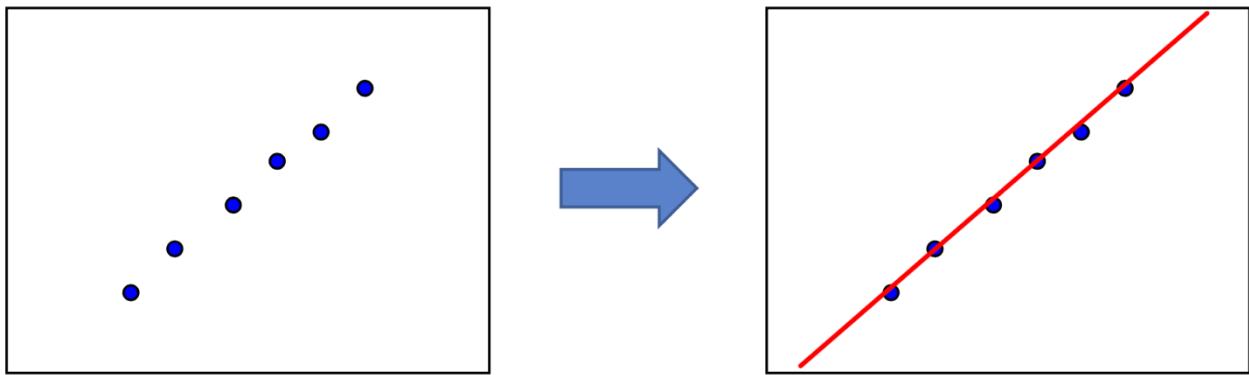
Per applicare la trasformata di Hough dobbiamo partire con **un'equazione parametrica**.

Partiamo con la forma canonica:  $y=mx+b$ . **m e b identificano univocamente una retta.** Faccio variare  $x$ , dati  $m, b$  ricavo  $y$ . Nella realtà implementeremo un'altra forma parametrica della retta che è quella *polare*:

$$\rho = x \cos \theta + y \sin \theta,$$

Dove la retta è definita da ***rho***: distanza della retta dall'origine degli assi e ***theta***: rho forma con l'asse delle x un angolo che è theta. Questi due parametri identificano univocamente una retta.

Partiamo quindi dai punti di edge, vogliamo trovare i parametri della retta che passano per i punti. I punti ovviamente non saranno perfettamente allineati, questo indica l'errore di acquisizione, rumore... nonostante ciò la trasformata di Hough risolve questa problematica.



## SPAZIO DEI PARAMETRI

Una retta nel nostro spazio immagine la descriviamo attraverso la forma canonica. Fissati  $m_0$  e  $b_0$ , al variare di  $x$ , ho la corrispondente  $y$ , così ho tutti i punti della retta. Hough introduce un altro spazio, quello dei **parametri**: gli assi sono parametrizzati da  $m$  e  $b$  (*coefficiente angolare* e *intercetta*): nello spazio immagine le coordinate sono  $x,y$  e nello spazio dei parametri sono  $m,b$ .

Fissati  $m$  e  $b$ , nello spazio dei parametri, queste coordinate corrispondono a **un unico punto**.

Nello spazio dei parametri, un punto (coppia  $m,b$ ) identifica una e una sola retta.

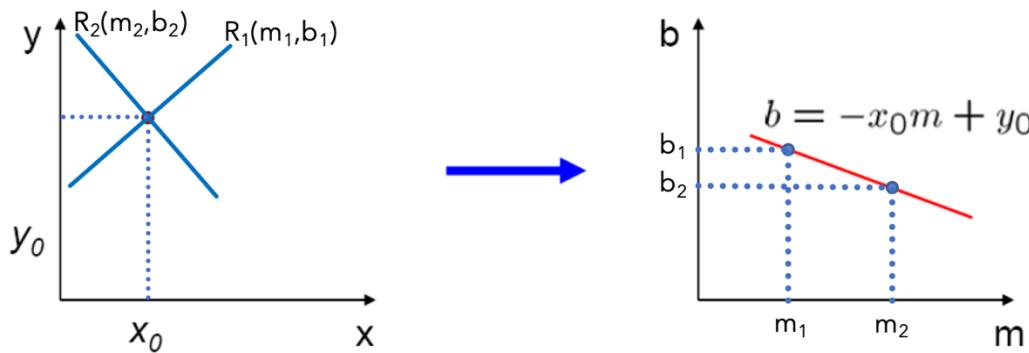
Abbiamo detto che bisogna lavorare sugli edge. Devo partire da un solo punto con coordinate  $x_0, y_0$  che viene dato da Canny. Per un punto passano infinite rette, quindi ognuna di essa sarà univocamente descritta da una coppia  $m,b$  parametri.

## Cosa hanno in comune tutte queste rette?

- Il passaggio per quell'unico punto, cosa succede? Parto dall'equazione canonica e inverto i parametri con le variabili per cui  $m,b$  diventano le variabili e  $x,y$  diventano i parametri perché fissati  $x,y$  (punto in comune), al variare di  $m,b$  ottengo i punti della retta di equazione  $b = -xm + y$ .

Esempio: consideriamo solo 2 rette possibili che passano per il punto:

questi parametri identificano 2 punti, attraverso cui passa la retta  
 $b = -x_0 m + y_0$ .

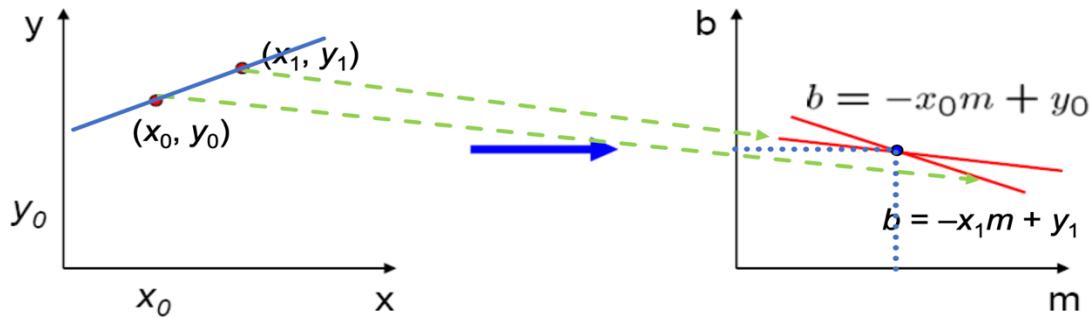


**Domanda di esame:** Perché sappiamo che le infinite rette che passano per l'unico edge si dispongono esattamente lungo una retta nello spazio dei parametri?

Perché tutti passano per il punto fissato  $x_0, y_0$  che diventano coefficiente angolare e intercetta della retta e facciamo variare  $m, b$ . tutti i punti quindi condivideranno stesso coeff e intercetta e quindi tutti giacciono sulla stessa retta.

**Ricordiamo che il nostro obiettivo è trovare le rette:** per identificare univocamente una retta, si necessitano di almeno 2 punti di edge. Consideriamo due punti  $(x_0, y_0), (x_1, y_1)$  attraverso cui passa una sola retta. Un punto di edge nello spazio immagine corrisponde a una retta nello spazio dei parametri, l'altro punto corrisponde all'altra retta. Queste due rette si intersecano in un punto con coordinate  $m, b$  che sono coefficiente angolare e intercetta su cui sono i due punti.

Quindi le rette che passano per i due punti dello spazio immagine si intersecheranno in un punto  $m, b$  nello spazio dei parametri, ecco la trasformata di Hough.

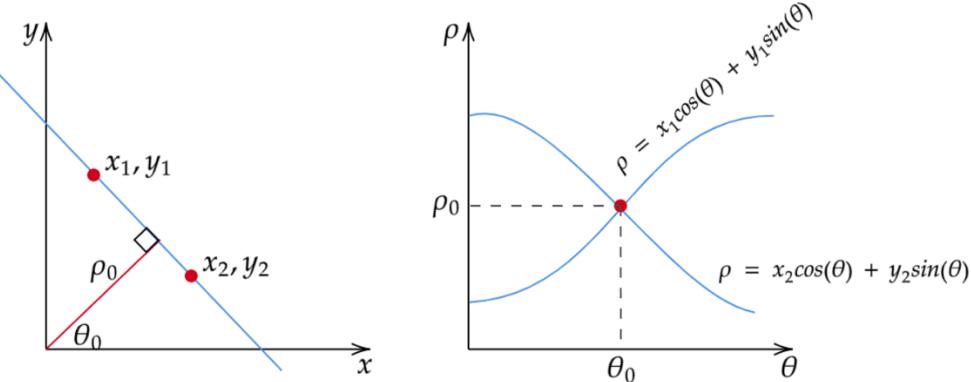


Nel momento in cui la retta è parallela alle ordinate, il coeff angolare va a ***infinito*** quindi non posso rappresentare la retta che va per  $m, b$ . Dovendo implementare questa cosa non posso rappresentare la retta su cui cercare le intersezioni. Si passa quindi alla rappresentazione polare in cui la retta è rappresentata in questo modo:  $\rho$  è distanza da retta dall'origine,  $\theta$  è l'angolo formato da  $\rho$  con le ascisse. Al variare di  $\rho$  e  $\theta$  ho tutte le possibili retta.

La retta nello spazio immagine è caratterizzata da  $\rho$  e  $\theta$ , che diventano i parametri nello spazio dei parametri: avrò un'asse rho e uno theta, il punto di coordinate  $(\rho, \theta)$  corrisponde alla retta nello spazio dell'immagine.

Per un punto di edge passano infinite rette che sono caratterizzate da coppie  $(\rho, \theta)$ . **L'unica differenza:** mentre nello spazio dei parametri le varie coppie  $m, b$  definiscono una retta,  $\rho$   $\theta$  qui definiscono una *sinusoide*.

- Il **punto di intersezione** nello spazio dei parametri individua la **retta** su cui **giacciono i punti** nel spazio **immagine**



## SCHEMA DI VOTO

Per implementare di cosa abbiamo bisogno?

Partiamo dai punti di edge, per ogni punto passano infinite rette, per cui dobbiamo considerare un numero **finito** di rette che è dato dalla variazione del parametro **theta** (ottengo tutte le rette che passano per il punto).

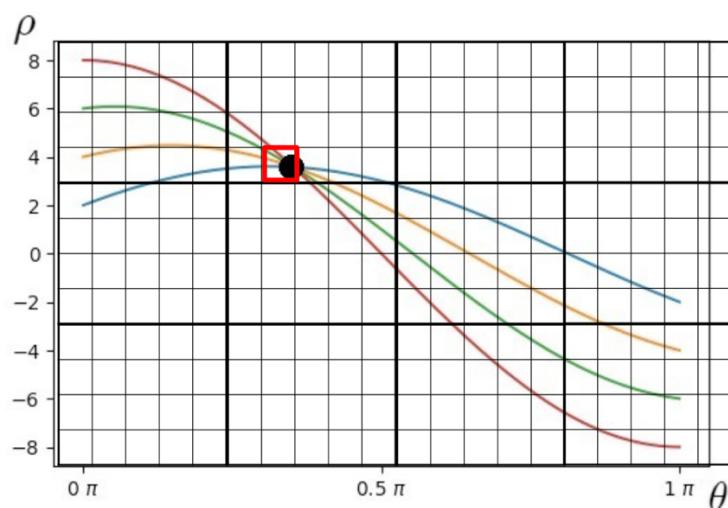
Dovrò avere uno spazio dei parametri in cui *ricordare* tutte le coppie rho theta che passano per il singolo punto.

Quale potrebbe essere un'idea per ricordare queste coppie?

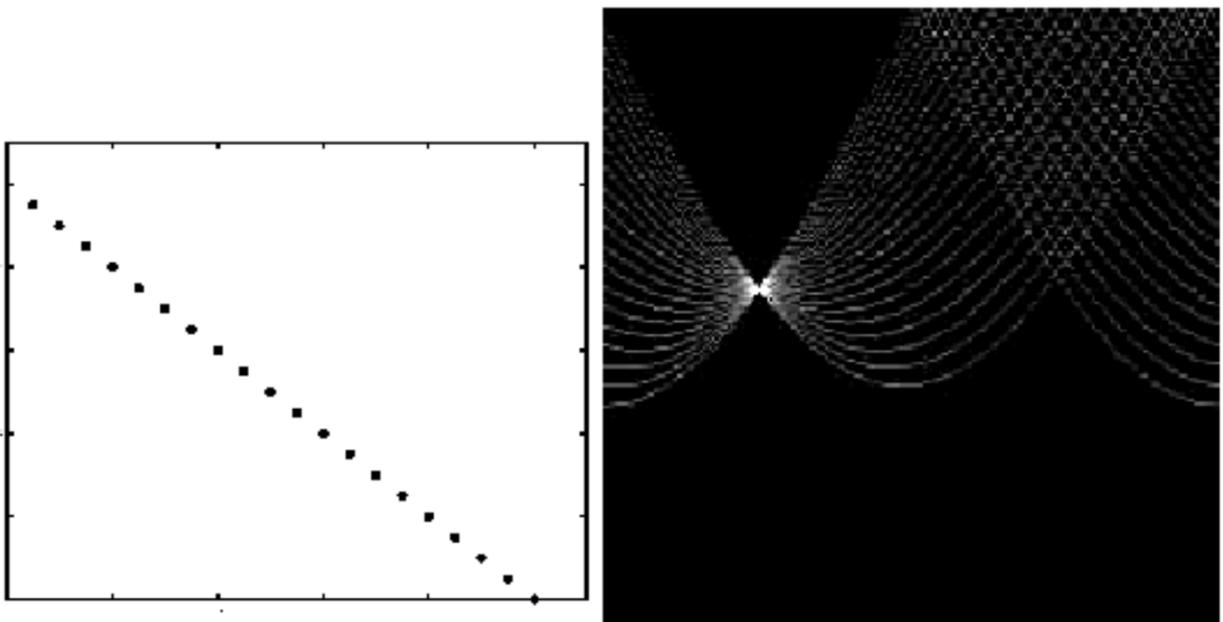
Una **matrice** in cui gli indice di ogni singola cella sono rho e theta. Possiamo **quantizzare** lo spazio dei parametri *rho, theta* (definiamo intervalli per ogni angolo e ogni unità rho) e ogni punto segnerà *il fatto che esiste un punto di edge attraverso cui passa una retta di coordinate rho theta*.

L'1 nella cella indica un punto di edge che sta *votando per la coppia rho theta*.

Questo spazio viene chiamato **spazio dei voti**. Se nello spazio troviamo tanti voti in una cella, significa che ci sono tanti edge che condividono quella retta. Quantizzando vado a recuperare tutti gli errori di rumore. L' algoritmo può essere *parallelizzabile* quindi nonostante la sua complessità computazionale molto elevata può essere migliorato ampiamente.



Ripetendo: Ho dei punti di edge che si dispongono lungo una retta. Ogni punto corrisponde a una **sinusoide**. Quantizziamo con una matrice, ognuno di questi punti vota per le possibili configurazioni. Ogni volta la sinusoide passa per una cella, il punto aggiunge un 1 (vota). Se immaginiamo questo spazio alla fine di procedura di voto, avrò molti picchi più o meno alti. Se lo plottiamo abbiamo il picco bianco dove si incrociano i voti ma anche altri picchi nelle vicinanze che determinano altre intersezioni.



Ricapitolando: **smoothing** per ripulire l'immagine, facciamo Canny per i punti di edge. Fatto ciò, andiamo a calcolare con *ipot* la distanza maggiore che possiamo trovare nell'immagine cioè la diagonale dell'immagine e andiamo a creare il nostro spazio di voti che è lo spazio dei parametri quantizzato. Creiamo uno spazio che ha  $dist^2$  dove  $dist$  sono i possibili valori di  $\rho$ . Poiché  $\rho$  varia tra  $-\rho$  e  $+\rho$ , poiché una matrice non può avere indice negativo, lo alloca con 2 volte  $\rho$  e riporto quelli negativi nell'intervallo positivo. Con 2 for analizzo tutti i pixel di edge che votano per le configurazioni delle rette. Verifico se è un punto di

edge: se così per ogni theta (tra 0 e  $180^\circ$ ) passo da angoli a radianti per seno e coseno. Theta e rho saranno gli indici su cui voteranno e incremento la posizione. Dopo che tutti hanno votato devo estrarre le coppie rho theta più votate: stabilisco una soglia che dipenderà dal numero di pixel di edge. Vado a scorrere lo spazio dei voti, se il numero di voti è superiore a una soglia (ho una coppia rho theta per cui hanno votato più edge e probabilmente avrò una retta) porto theta tra  $-90$  e  $+90$ , calcolo le coordinate x,y del punto che appartiene alla retta. Devo disegnare una retta: prendo il punto x,y come un punto centrale e ho bisogno di due punti: ne prendo uno a  $+dist$  e uno  $-dist$  lungo la direzione della retta (i due estremi) tra questi due punti disegno la linea.  
**[all'esame è importante modificare i valori per ottenere un buon output].**

---

## HOUGH PER CIRCONFERENZE

L'idea è la stessa, cambia la **rappresentazione**.

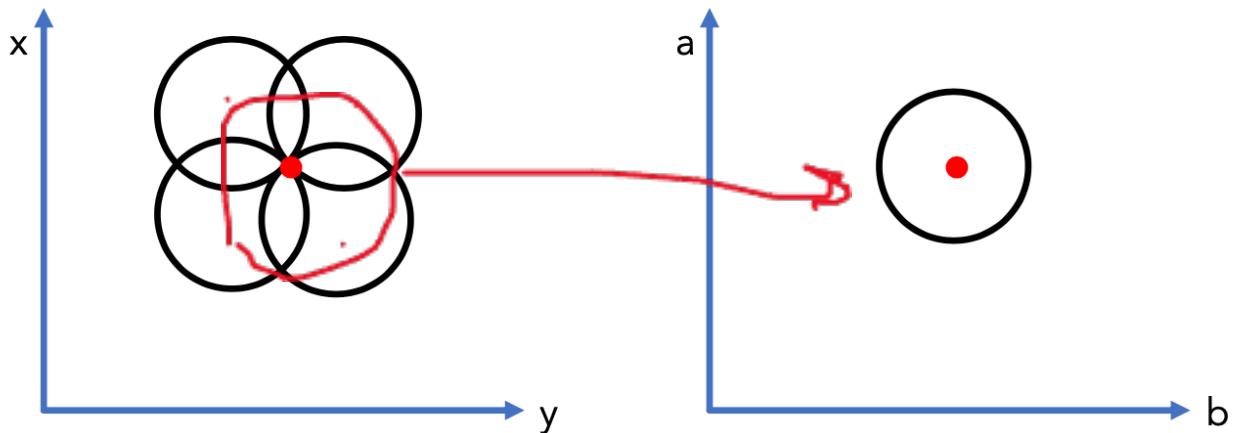
L'equazione parametrica: fissati  $a, b$  (coordinate del centro), fissato  $R$  (lunghezza del raggio), al variare di **theta** tra 0 e  $360^\circ$  trovo le coordinate dei punti che si dispongono lungo la circonferenza.

$$x = a + R \cos \theta$$

$$y = b + R \sin \theta$$

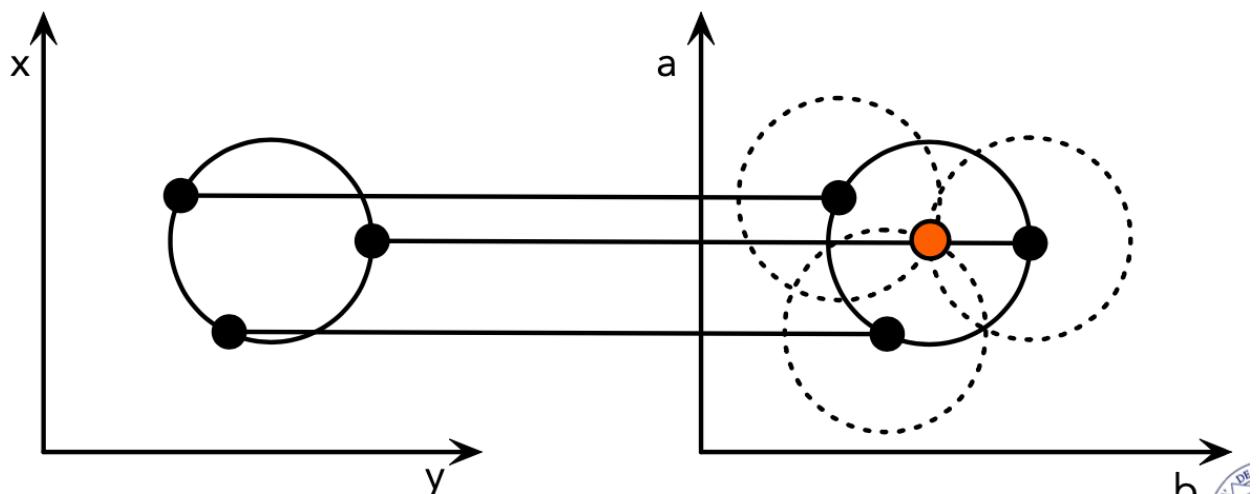
Inizialmente consideriamo lo spazio bidimensionale: fissiamo  $R$ , gli unici parametri che possono variare sono  $a, b$ .

Consideriamo il punto di edge: per un punto di edge passano infinite rette. Un punto di edge, nello spazio immagine corrisponde a una circonferenza nello spazio dei parametri  $a, b$  perché per un punto passano infinite circonferenze ma avendo fissato  $R$  avranno tutte raggio uguale. Se uniamo i punti, si forma una circonferenza di raggio  $R$ .



Il punto di edge voterà per tutte le coppie  $a,b$  delle circonferenze che passano.

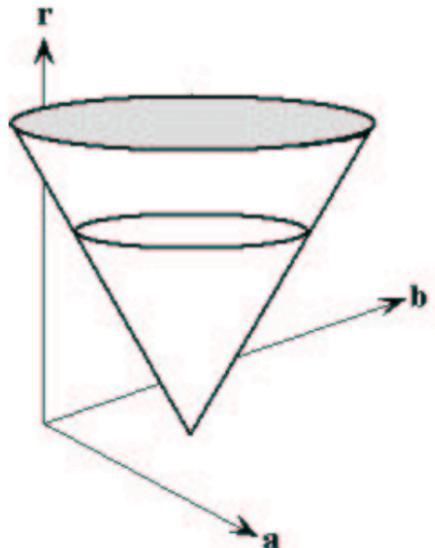
Se considero 3 punti (passa 1 circonferenza), ogni punto corrisponde a una circonferenza:



Queste 3 circonferenze si intersecheranno in un punto che corrisponde alla circonferenza che stiamo cercando.

Quantizzo lo spazio dei parametri ed ogni punto di edge va a votare all'interno delle celle. Nel punto centrale dove si intersecano le circonferenze troverò al maggior numero di voti che corrisponde alle coordinate  $a,b$  del centro delle circonferenze che sto cercando nello spazio immagine.

Ricordiamo che abbiamo fissato  $R$ , se invece  $R$  non è fissato lo spazio dei parametri è **tridimensionale** ( $a,b,R$ ): un pixel di edge dovrà votare per le coppie  $a,b$  al variare di  $R$ .



Sto creando un **cono** ma lavoriamo solo sulla sua **superficie**.

Ogni singolo punto crea un cono nello spazio dei parametri. Un altro punto di edge andrà a creare un altro cono, le cui superfici si intersecheranno in un punto che **determina centro, raggio e le coordinate a,b**.

Codice: facciamo blur e canny e vado a creare uno spazio di voto tridimensionale: la matrice di voti è  $3 \times \text{size}$  dove size contiene righe, colonne e un intervallo **rmin e rmax** (si stabilisce un intervallo di raggi dove cercare le circonferenze). Dopo aver definito lo spazio dei voti tridimensionale: per ogni punto di edge, per ogni possibile raggio, considero gli angoli da  $0$  a  $360^\circ$ , calcolo le coordinate a,b del cerchio nello spazio dei parametri, vado poi a vedere se le coordinate del centro cadono all'interno dell'immagine (sto restringendo l'analisi solo alle circonferenze i cui centri sono visibili nell'immagine) e vado a votare. Infine, dove ho un picco che supera la soglia vado a disegnare il cerchio.

## PASSI DELL'ALGORITMO DI HOUGH PER RETTE

- Inizializzare l'accumulatore H
- Applicare l'algoritmo di Canny per individuare i punti di edge
- Per ogni punto  $(x, y)$  di edge
- Per ogni angolo  $\theta = 0: 180$  calcolare  $\rho = x \cos \theta + y \sin \theta$
- Incrementare  $H(\rho, \theta) = H(\rho, \theta) + 1$
- Le celle  $H(\rho, \theta)$  con un valore maggiore di una soglia  $th$  corrispondono alle rette nell'immagine

## PASSI DELL'ALGORITMO DI HOUGH PER CERCHI

- Inizializzare l'accumulatore H
- Applicare l'algoritmo di Canny per individuare i punti di edge
- Per ogni punto  $(x, y)$  di edge
- Per ogni angolo  $\theta = 0: 360$  e per ogni raggio  $r=R_{\min} : R_{\max}$
- Calcolare  $a = x - r * \cos\left(\theta * \frac{\pi}{180}\right)$  e  $b = y - r * \sin\left(\theta * \frac{\pi}{180}\right)$
- Incrementare  $H(a, b, r) = H(a, b, r) + 1$
- Le celle  $H(a, b, r)$  con un valore maggiore di una soglia  $th$  corrispondono alle circonferenze nell'immagine

## Lez 9 -> SOGLIATURA

È una tecnica di segmentazione molto importante. Abbiamo visto in OpenCV la funzione ***threshold*** che ci permette di sogliare l'immagine (trovare il valore di intensità di grigio attraverso cui discriminiamo gli oggetti in background da quelli in foreground). Consideriamo questa una tecnica di segmentazione perché classifichiamo i pixel appartenenti a una classe o a un'altra.

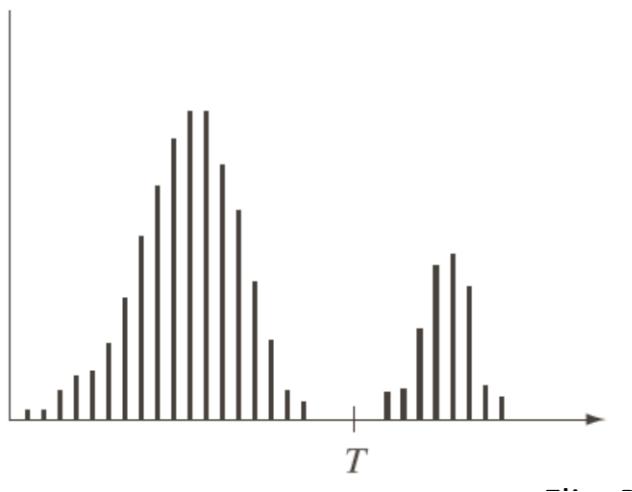
Analizzeremo questa tecnica e i possibili approcci algoritmici, come l'algoritmo di ***OTSU*** che ci permette di trovare una soglia ottimale a partire da un'immagine a scala di grigio.

Per quanto riguarda la sogliatura, andremo a usare valori di intensità locali, al contrario dell'istogramma che ci dà info globali.

L'idea: partiamo dall'istogramma di un'immagine che ci dice per ogni livello di grigio sulle ascisse, quanti pixel presentano quel determinato valore.

Abbiamo una distribuzione bimodale (due picchi) che rappresentano due parti di livelli di grigio nella nostra immagine: una parte più scura e una più chiara.

Con l'istogramma non studiamo la posizione ma solo i valori. Dobbiamo trovare una soglia  $t$  che permette di separare i pixel scuri dai chiari.



## TIPI DI SOGLIATURA

- Globale: un solo valore di soglia per tutta l'immagine.
  - Variabile: è adattata iterativamente.
  - Locale: la soglia dipende dall'intorno di ogni pixel. Mentre le altre osservano l'intero istogramma, la sogliatura locale prende in considerazione l'intorno, quindi la zona del pixel.
  - Dinamica: si individuano delle zone e all'interno si identifica la soglia.
  - Multipla: si usano più soglie. Quando l'istogramma è multimodale, per una segmentazione ottimale dobbiamo trovare più soglie.
- 

## FATTORI CHE INFLUENZANO LA SOGLIATURA

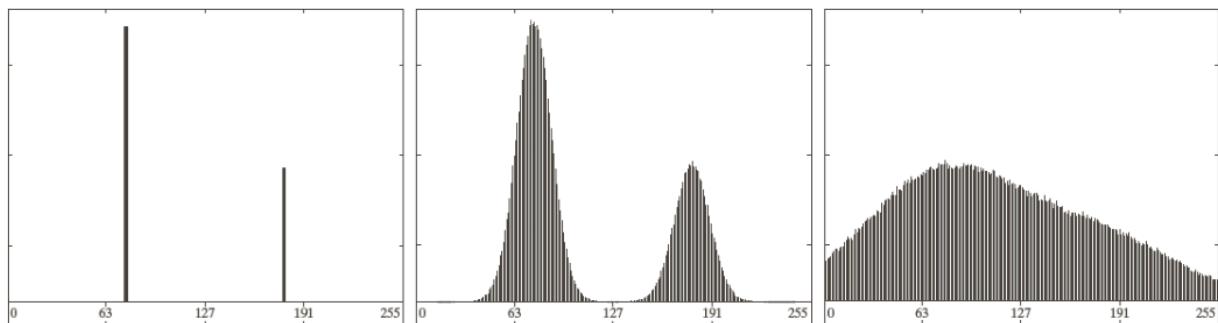
- Distanza tra i picchi: se consideriamo l'istogramma, se ho due picchi ben distanti, qualunque soglia prendo sarà ottimale. Finché i livelli di grigio sono distanziati non c'è problema. Se ho una distribuzione più complessa, è difficile trovare una soglia ottimale.
- Rumore nell'immagine.
- Dimensione degli oggetti rispetto allo sfondo: se ho un oggetto piccolo in foreground rispetto a uno sfondi in bg, il numero di pixel che contribuiscono alla costruzione dell'istogramma dell'oggetto è più piccolo del bg.
- Uniformità della fonte di illuminazione: il cambiamento dell'illuminazione modifica la forma dell'istogramma.

- Uniformità della proprietà di riflettanza nell'immagine.
- 

## RUOLO DEL RUMORE

Consideriamo un istogramma con due picchi (bimodale): sono ben distanziati e quindi troviamo una sogliatura ottimale.

Aggiungiamo del rumore gaussiano con una deviazione di 10 livelli di grigio (rispetto alla media modifichiamo i livelli di intensità in modo che si distanziano da quel valore di 5 livelli di grigio), finchè c'è un rumore insinuo va bene ancora, con 50 livelli di deviazione, l'effetto è non avere più moda.

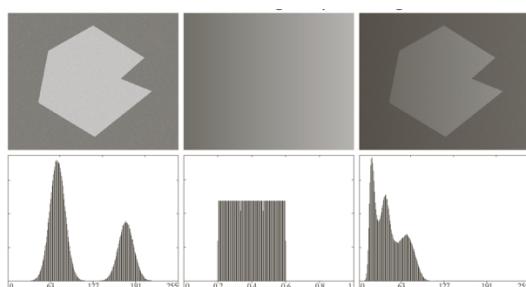


Ovviamente per eliminare il rumore e poi elaborare l'immagine, si fa smoothing.

---

## RUOLO DELL'ILLUMINAZIONE

Partiamo da un istogramma e simuliamo una illuminazione non uniforme (simuliamo una fonte non uniforme su tutto l'oggetto). Otteniamo un istogramma differente a seconda dell'illuminazione



## SOGLIATURA GLOBALE

Cerco una soglia per dividere gli oggetti in background dai foreground.

È un procedimento iterativo finchè non si verifica una condizione di uscita.

1. Scelgo un valore di soglia  $T$ . questo valore separa i livelli di grigio in due classi. Calcolo il valore medio di livelli di intensità a sinistra e destra della soglia ottengo due gruppi  $G_1$  e  $G_2$ .
  2. Calcolo la media  $m_1$  e  $m_2$  nei due gruppi.
  3. Calcolo la media delle medie e ottengo una nuova soglia.
  4. Posiziono  $T$  in una nuova posizione e ripeto il ciclo.  
Funziona bene se ho 2 picchi ben separati
- 

## METODO DI OTSU

Se non ho conoscenza a priori dell'istogramma possiamo usare l'algoritmo di **OTSU**. Questo algoritmo tratta il problema della sogliatura da un punto di vista **statistico**, trovando una separazione ottima di una qualunque distribuzione dei livelli di grigio nell'istogramma.

La separazione ottima minimizza l'errore medio: quando scelgo una soglia, pongo il problema come un problema di **classificazione**, assegnano i pixel a una classe. Quando decido la soglia, potrei assegnare pixel di  $bg$  ai  $fg$  e viceversa, commettendo degli errori che mischiano le carte.

L'algoritmo di OTSU minimizza proprio questo errore appena descritto. Per fare questo, massimizza la varianza interclasse, cioè massimizzare la distanza tra le due classi.

OTSU tratta i valori dell'istogramma come delle probabilità.

Trattare la sogliatura in questo modo comporta una complessità media computazionale.

## PASSI DI OTSU

Calcoliamo l'istogramma: l'istogramma per ogni posizione contiene il numero di pixel che presenta quel livello di intensità. Se dividiamo ogni valore per il numero di pixel dell'immagine ottengo la percentuale di pixel di quei livelli di grigio, ottenendo un valore compreso tra 0 e 1. Se consideriamo l'intera immagine  $M \times N$ , la frazione può essere vista come una probabilità: se prendo un pixel a caso, qual è la probabilità che quel pixel avrà un determinato livello di grigio? (data un'immagine bianca, il pixel 255 ha 100% di probabilità ad esempio).

Così ho l'istogramma normalizzato che è visto come una probabilità. Se vado a sommare tutte le probabilità ottengo 1. Dobbiamo trovare una soglia  $k$  che divide l'istogramma in due classi  $C_1$  e  $C_2$ : la classe dei pixel con intensità minore della soglia e maggiore della soglia. Dato un  $k$  posso calcolare la probabilità che un pixel appartenga a  $C_1$  o a  $C_2$ , sommando tutte le probabilità a sx e dx di  $k$ .

- La **probabilità** che un **pixel appartenga** alla classe  $C_1$  è  $P_1(k) = \sum_{i=0}^k p_i$
- La **probabilità** che un **pixel appartenga** alla classe  $C_2$  è  $P_2(k) = \sum_{i=k+1}^{L-1} p_i = 1 - P_1(k)$

Abbiamo calcolato probabilità che un pixel assuma un determinato valore, senza considerare i valori di intensità.

Se calcolo solo la probabilità, non conosco la forma della distribuzione.

Per introdurre questa info calcolo il valore di intensità media nelle due classi pesando il valore di intensità  $i$  con la probabilità che quel valore si presenti:

$$m_1(k) = \frac{1}{P_1(k)} \sum_{i=0}^k ip_i$$

Se  $p_i$  è zero, il valore medio  $i$  si verificherà mai. Se si presenta con un probabilità 0.1 darà contributo 0.1. Se  $i=1$  il valore medio è  $i$ .

Faccio lo stesso con la classe C2.

Dopo vado a calcolare la media cumulativa di tutti i livelli di grigio fino a  $k$  e la media globale (media di tutti i valori nell'immagine).

Questi elementi servono a calcolare la varianza interclasse (*separare al meglio le due classi dato un k*).

Al crescere delle due medie la varianza interclasse aumenta.

Più i valori sono vicini alla media più la varianza è bassa e viceversa.

Usando OTSU voglio capire quanto la varianza interclasse è diversa rispetto alla varianza globale, vogliamo studiare come si rapporta la dispersione dei pixel nelle due classi rispetto alla varianza globale.

[vedere meglio slide 14-15-16]

#### ALGORITMO DI OTSU

- Calcolare l'istogramma normalizzato dell'immagine
- Calcolare le somme cumulative  $P_1(k)$  per  $k=0, 1, \dots, L-1$
- Calcolare le medie cumulative  $m(k)$  per  $k=0, 1, \dots, L-1$
- Calcolare l'intensità globale media  $m_G$
- Calcolare la varianza interclasse  $\sigma_B^2(k)$  per  $k=0, 1, \dots, L-1$
- Calcolare la soglia  $k^*$ , ovvero il valore  $k$  per cui  $\sigma_B^2(k)$  è massimo
- Calcolare il valore di separabilità  $\eta(k^*)$

L'ultimo punto è facoltativo perché non aggiunge nulla al calcolo della soglia, serve per dare un'indicazione della qualità della separazione, più grande è meglio è e viceversa.

---

## PROBLEMA DEL RUMORE – SMOOTHING

La presenza di rumore può essere complesso da gestire. Un passo di smoothing ci permette di tornare a un istogramma più pulito e ritornare a una distribuzione facilmente separata.

Non sempre ciò accade...

Un problema si presenta quando l'oggetto da segmentare è molto piccolo rispetto al background e lo smoothing non risolve il problema. Se effettuo sogliatura globale o OTSU comunque non risolvo, forse peggioriamo.

Invece di considerare tutti i pixel possiamo usare solo i pixel di edge. Quando prendo i pixel di edge, troverò quei pixel che possono appartenere ai bg o al fg, di fatto eliminiamo il problema del rumore. Quindi nel calcolo dell'istogramma considero solo i pixel di edge.

- Idea: calcolo l'immagine di edge. Individuare un valore di soglia  $T$  riguardo gli edge. Applichiamo la soglia all'immagine di edge ottenendo un'immagine binaria  $g_T$  che è 0 non edge e 1 edge. Calcolo l'istogramma e lo uso per la segmentazione col metodo di OTSU.
- 

## SOGLIE MULTIPLE

Se l'istogramma presenta più mode, una sola soglia non è sufficiente a separare bene. L'idea è considerare lo stesso procedimento di OTSU però avere due o più indici che scorrono l'array, ho quindi più  $k$ . Devo trovare i due  $k$ , nel caso di 3 classi ad esempio, che massimizzano la varianza interclasse.

Al crescere del numero di classi, OTSU inizia a non lavorare benissimo perché spinge le classi centrali verso la media. Ottengo

una matrice triangolare di valori con tutti i valori possibili di k1 e k2.

---

## SOGLIATURA VARIABILE

Questa tecnica consiste nel dividere l'immagine in zone.

Se separiamo in zone otteniamo regioni in cui l'istogramma è ben separato e possiamo applicare bene OTSU.

La soglia cambia in base alla posizione del pixel nell'immagine.

Il problema qui è come dividere l'immagine.

La tecnica di sogliatura variabile visualizza localmente un intorno. Si considera un intorno del pixel e si trova la soglia ottimale in quell'intorno.

---

- Per effettuare la sogliatura (thresholding) in OpenCV

```
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  
) ;
```

## Lez 10 -> REGION GROWING & SPLIT AND MERGE

### SEGMENTAZIONI SULLE REGIONI

Affronteremo due algoritmi che hanno come obiettivo quello di creare delle regioni sfruttando la similarità tra pixel.

Già con la sogliatura accumuniamo i pixel per similarità.

Denotiamo con  $R$  la regione occupata dall'immagine. La segmentazione consiste nel partizionare  $R$  in  $n$  sotto regioni. L'unione di tutte le regioni ci deve ridare l'immagine originale, ogni pixel deve appartenere a una regione. Date due regioni  $R_i$  e  $R_j$  la loro intersezione deve essere vuota.

Ogni regione crea una componente连通的, i pixel che appartengono alla regione devono essere adiacenti e devono presentare un valore di intensità simile secondo una regola.

Dovremo stabilire un predicato  $Q$  che può essere applicato ai pixel di una regione e deve restituire come risultato “*vero*”: ad esempio la varianza non deve essere maggiore di una soglia ecc...

Il predicato applicato a due regioni adiacenti deve restituire *falso* perché se considero un'immagine con 2 regioni, so che il predicato per  $R_1$  e il predicato di  $R_2$  devono dare vero. Se applico il predicato  $Q$  a  $R_1$  unito a  $R_2$  deve dare *falso*, se fosse il contrario vuol dire che le due regioni sono *uguali* e quindi sono 1 sola.

---

### REGION GROWING

L'idea è quella di accrescere delle regioni. Si parte da pixel singoli chiamati *seed* e seguendo una certa connettività (4/8 connessioni) e un predicato, si cerca di aggiungere ai *seed* altri pixel adiacenti. Se provo ad aggiungere un pixel alla regione e il predicato è rispettato, allora si può aggiungere. È come fare un'esplorazione seguendo alcune regole, finché troviamo pixel continuiamo.

Quando abbiamo aggiunto i pixel, a partire da questi cercherò di far accrescere la regione finché c'è possibilità di aggiungerne.

Nel momento in cui la regione non può essere più accresciuta, si passa a un altro *seed*.

**Importante** -> Rilevare i *seed* è fondamentale. La scelta è esterna all'algoritmo di region growing. A partire da un'immagine, applicando delle tecniche come edge detection e sogliatura, si arrivano a pixel reputati importanti, che saranno i *seed*. Per fare questo dobbiamo avere una certa conoscenza dell'immagine in input, da quei pixel poi partirà l'algoritmo.

Se non abbiamo conoscenza dell'immagine, tutti i pixel possono essere seed, finchè non entreranno a far parte di una regione.

Se immaginiamo un'immagine, tutti i pixel potrebbero essere seed: partiamo dal primo pixel e si accresce la regione a partire da esso. Secondo il predicato si fa accrescere la regione, quando non si può espandere, si va ad analizzare a partire l'ultimo pixel di seed i pixel successivi: se sono in una regione non saranno più seed e si procede, se il successivo non è in una regione, avremo il nuovo *seed*. Ciò garantisce che avremo una segmentazione completa perché tutti i pixel saranno scansionati.

È importante decidere un predicato: se scegliamo valori vicini al seed avremo una segmentazione più raffinata e viceversa.

Raggruppare pixel ***non adiacenti*** può generare segmentazioni inconsistenti.

Posso calcolare il predicato sempre rispetto al *seed* di partenza oppure possiamo farlo rispetto al pixel di fronte.

In generale bisognerebbe tener conto di tutti i pixel inseriti nella regione, ovviamente ciò è molto costoso computazionalmente ma si possono fare alcune approssimazioni per avere una media abbastanza fedele (*average running*).

Passi:

abbiamo la matrice dei seed binaria: 0 seed e 1 non seed. Per ogni posizione x,y avremo i valori 1/0. Se non abbiamo questa info, la matrice verrà popolata di 1 o 0 se il pixel è seed o no.

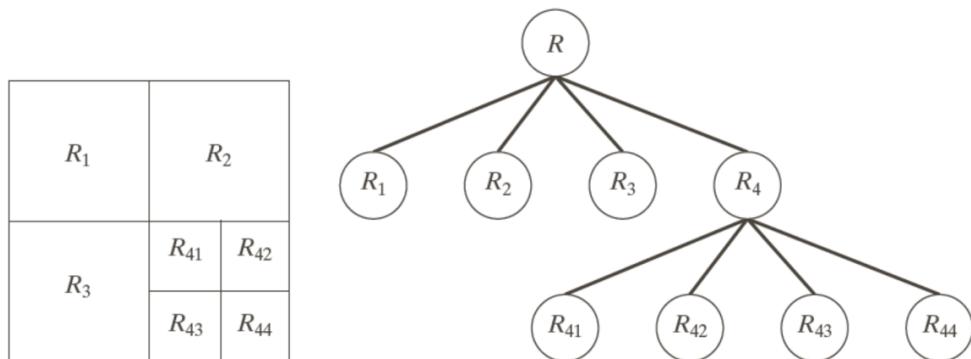
Si stabilisce poi un predicato Q e una connettività e si aggiungono i pixel alle regione seguendo questi fattori appena citati. Se il predicato è falso per quella regione, quel pixel andrà in un'altra regione, finché non avremo una segmentazione completa.

Ovviamente la segmentazione è un problema **mal posto** poiché non sappiamo definire una segmentazione giusta o errata, dipende dai punti di vista e dall'uso che si farà dell'elaborazione.

---

## SPLIT AND MERGE

Usa un'idea simile, consiste nel dividere un'immagine in regioni, verificare il predicato nelle regioni, se il predicato è falso, si continua ad esplorare. Vedremo una divisione in regioni regolari (quadrate): si parte da un'immagine R, si applica il predicato a tutta la regione, se il predicato è falso vuol dire che la regione individuata non risponde alla segmentazione voluta e divido la regione in sotto regioni, useremo una **4 tree che divide R in 4 sottoregioni**.



Nel momento in cui ho finito la divisione, avendo usato una regione regolare, potrei avere due regioni adiacenti che potrebbero essere unite (due regioni unite devono dare predicato falso). Nella fase di *split* dove divido sto usando una proprietà dove il predicato deve essere vero, se è falso vuol dire che ci sono più regioni e vanno divise. Finito la split, l'unione di due regioni adiacenti

potrebbero essere vere, ciò significa che dovrebbero essere incluse in una sola regione: parto quindi con la fase *di merge* dove uniamo regioni piccole adiacenti la cui unione torna un predicato comunque vero.

Fatto ciò, posso ragionare in due modi: o il merge lo faccio *bottom up*, ottenendo una segmentazione più fine ma costosa, oppure un merge *top down*, dove aggregiamo le regioni più grandi.

In generale: divido in quadranti l'immagine.

Nella fase di merge analizzo le regioni adiacenti, applichiamo il predicato all'unione e se è vero le mergiamo, così via finché non possiamo fare ulteriori unioni.

## Lez 11 -> CLUSTERING

Gli psicologi hanno individuato una serie di fattori che consentono alla vista di raggruppare un insieme di elementi.

Vogliamo usare delle caratteristiche come colore e intensità per trovare oggetti e zone di interesse di un'immagine.

Una tecnica che sfrutta queste similarità è la *sogliatura*: quando scelgo una soglia sto dividendo i valori di intensità per similarità. Ora ci interesseremo dei livelli di intensità presi per ogni pixel singolo.

---

### K-MEANS

Il primo algoritmo è il *k-means*.

Immaginiamo uno spazio bidimensionale dove abbiamo dei punti. Ogni punto è individuato da una coppia di coordinate  $x,y$ . Il compito è raggruppare questi punti per *similarità*. Quando parlo di similarità devo definire un *criterio*. Un modo potrebbe essere la *distanza euclidea*: più due punti sono vicini, più sono simili e viceversa. Una volta ottenuti i gruppi, per rappresentarli e avere un modo compatto di visualizzazione, posso trovare il punto centrale di questi gruppi, quello chiamato *centro di massa*.

Prendo le coordinate  $x,y$  di tutti i punti e ne faccio la *media*, stiamo quindi cercando il centro nel gruppo di pixel, possiamo vederlo come il **“rappresentante del cluster”**.

Dobbiamo trovare delle regole che permettono, dato un insieme di punti, di raggruppare pixel in cluster e calcolare un rappresentante.

---

### FUNZIONAMENTO K-MEANS

*Il K-means*: dati i punti in uno spazio bidimensionale, devo decidere il valore  $k$  che definisce il numero di gruppi in cui voglio dividere i punti a disposizione. *Supponiamo k=2*.

L'idea è semplice -> inizialmente, come rappresentante dei gruppi, scelgo due pixel a caso. Avendo scelto i rappresentanti, è

come se avessi già scelto i gruppi, il passo successivo è che ogni singolo punto deve essere associato al gruppo più vicino.

Fatto ciò, calcolo il nuovo rappresentante, e poi posso ripetere l'assegnazione.

Il k-means alterna queste due fasi: calcolo del rappresentante, assegnazione dei punti in base alla vicinanza. Questa procedura iterativa si ferma quando il nuovo rappresentante è molto vicino al precedente.

La funzione che descrive il k-means è:

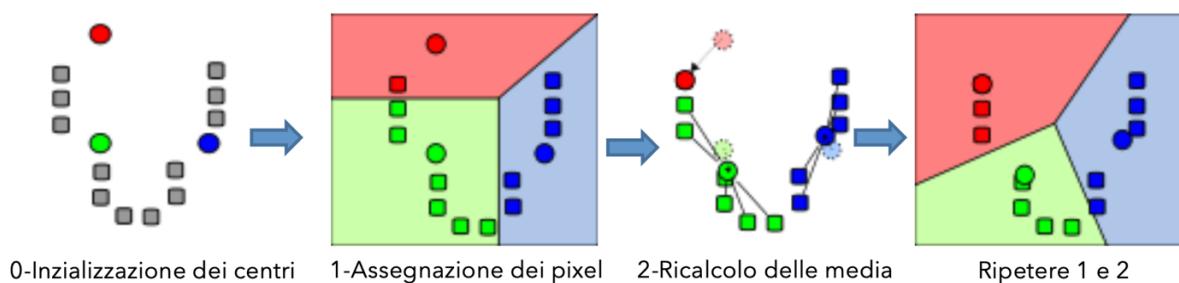
$$\mathbf{c}^*, \boldsymbol{\delta}^* = \underset{\mathbf{c}, \boldsymbol{\delta}}{\operatorname{argmin}} \frac{1}{N} \sum_j^N \sum_i^K \delta_{ij} (\mathbf{c}_i - \mathbf{x}_j)^2$$

Whether  $x_j$  is assigned to  $c_i$

Vogliamo minimizzare le distanze all'interno dei gruppi: *argmin* significa trovare due parametri  $c$  e  $\delta$  che rendono i gruppi più compatti possibile. I due parametri sono le medie ( $c$ ) e l'assegnazione ( $\delta$ ).

La funzione è minimizzata quando la distanza  $(c_i - x_j)$  è piccola, ciò significa che i pixel sono vicini. Inoltre  $d$  vale 0 o 1: se 1 contribuirà oppure viene scartato.

Il k-means riassume il problema *dell'uovo e della gallina*: per trovare i cluster servono i centroidi e viceversa, ecco perché i centroidi vengono presi casualmente all'inizio.



Il k-means ha un comportamento **non deterministico** poiché potrebbe trovare un minimo e arrestarsi, anche se più avanti ci

sarebbe un altro punto di minimo migliore. La soluzione è lanciare più volte l'algoritmo e prima o poi avremo un output migliore.

Ricapitolando, ecco i passi del *k-means*:

1. **Inizializzare** i centri di ogni cluster
2. **Assegnare** ogni pixel al cluster con il centro più vicino
  - Per ogni pixel  $p_j$  calcolare la **distanza** dai **k centri**  $c_i$  ed **assegnare**  $p_j$  al **cluster** con il centro  $c_i$  **più vicino**
3. **Aggiornare** i centri
  - **Calcolare** 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 (ovvero i gruppi non vengono modificati)

## K-MEANS ++

La scelta del  $k$  e dei valori iniziali di  $k$  potrebbe farci cadere il minimo locale che non minimizza globalmente la funzione.

Una possibile soluzione è *k-means++*: il primo punto è *random*, il secondo rappresentato è scelto *più lontano possibile dal primo*, il terzo più lontano di entrambi e così via...

Questo costa  $k * m$  computazionalmente.

**Questa inizializzazione però, consente di scegliere dei rappresentanti distanti e che avranno probabilmente una soluzione ottimale.**

Il *k-means* non vuole componenti connesse perché un livello di intensità potrebbe trovarsi in diverse regioni dell'immagine.

Un altro miglioramento consiste anche nel considerare la *componente spaziale*.

---

## CLUSTERING: MEAN SHIFT

È simile al k-means ma non devo scegliere a priori il parametro  $k$ . Troviamo il numero ottimale di cluster a partire dai valori di intensità o colore nell'immagine.

Consideriamo un istogramma coi valori di intensità: il *mean shift* vuole trovare le *mode* (*i picchi*).

Partiamo da un punto e consideriamo una finestra nell'intorno del valore: calcolo la media in questa finestra.

Quando calcolo la media, si sposta verso i valori più rappresentanti all'interno della finestra, cioè verso la moda di quella finestra. Si chiama mean shift perché la media si sposta. Idealmente, mi avvicino alla moda più forte.

Questo dipende però dall'ampiezza della finestra che stiamo considerando.

Nonostante sia molto più conveniente, mean shift costa di più.