Elaborazione delle immagini – Ferone

Appunti a cura di Fiorentino Michele
- Università degli studi di Napoli “Parthenope”

LEZ 1 – Introduzione a OpenCV (1)

OpenCV è una libreria open source per la computer vision e l'elaborazione delle immagini. L'obiettivo era fornire uno standard per le operazioni su immagini a basso livello, ma col tempo sono state aggiunte sempre più funzioni, tra cui anche un modulo di ML.

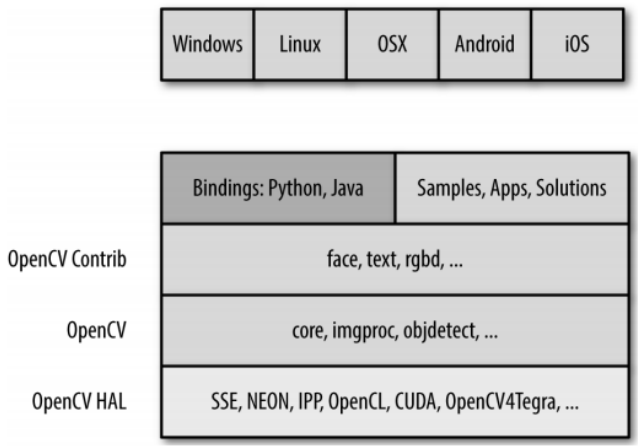
Si vuole dare a un programma la capacità di vedere un'immagine, che tuttavia non è scontato. Un oggetto lo riconosco perché riesco ad individuare un suo contorno, cioè una sua "discontinuità" tra la parte che sto individuando e la parte che lo circonda (es. una ruota rispetto al resto della macchina), e infatti il nostro compito sarà quello di acquisire l'immagine, portarla all'interno del programma, e fare delle elaborazioni per estrarre dei *segmenti* (cerchi, linee, bordi…).

L'algoritmo, ovviamente, è in grado di vedere solo una matrice. Attraverso la discontinuità ne individuo i contorni, che fornirò ad un algoritmo di ML che ne assegnerà un significato.

La discontinuità è ovviamente più difficile da individuare se usiamo una scala di grigi (e non solo il b/n), in tal caso cercheremo di individuare le *piccole* e *grandi discontinuità*, dunque lavoreremo su delle soglie.

OpenCV ha una **struttura a livelli**:

- al di sotto del SO troviamo le interfacce dei linguaggi supportati e le applicazioni;
- nel livello successivo ci sono le funzionalità di alto livello;
- nel livello successivo ancora ci sono le funzioni di basso livello;
- nel livello più basso troviamo le ottimizzazioni hardware. Tra le più importanti OpenCL e CUDA (che permette di usare le GPU).



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

- `#include "opencv2/opencv.hpp"` , tutte le funzioni (useremo all'esame)
- `#include "opencv2/imgproc/imgproc.hpp` , specifiche funzioni per le elim.

Esempio: caricamento e visualizzazione di un'immagine (OpenCV_1_LoadViewImg)

```
#include <opencv2/opencv.hpp>
using namespace cv;

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

Utilizziamo il namespace cv (anche all'esame sarà così).

`imread` si occuperà di leggere l'immagine che passeremo in argv[1]. Notare che per passare un'immagine si deve riportare il percorso dell'immagine stessa (se il percorso contiene spazi, inserirlo fra virgolette ""). -1 demanda alla libreria come caricare l'imm. `imread` restituirà un valore di tipo Mat.
La funzione determina automaticamente il tipo di file (BMP, JPEG, PNG…). Alloca la memoria necessaria per la struttura dati (cv::Mat) che conterrà i dati e viene deallocato automaticamente quando si esce dal contesto.

Controlliamo se l'immagine è stata caricata correttamente (quindi non è vuota).

L'immagine deve essere contenuta all'interno di un contenitore (finestra) altrimenti non saremmo in grado di visualizzarla, dunque creiamo anche una finestra attraverso `namedWindow(name, size)`, alla quale passeremo 1.nome delle finestra e 2.dimensione.

Per mostrare l'immagine usiamo `imshow(nWindow, nImg)`, passando il 1.nome della finestra l' 2. immagine che vogliamo visualizzare.

Usiamo `waitKey()` per visualizzare l'output, altrimenti aperta la finestra andremmo a chiuderla subito dopo. Il focus dove dovremo premere il testo è sulla finestra stessa.

nb: anche se abbiamo un'imm b/n, possiamo forzarla a colori (macro IMREAD_COLOR, 2 arg in imread()), solo che ricopia gli stessi valori per ogni banda dell'RGB.

per *settare gli argomenti* in Code::Blocks, andare in Project -> Set Program's arguments -> (Inserire sotto Program arguments).

LEZ 2 – Introduzione a OpenCV (2)

La funzione **`imread()`** ci permette di assegnare un'immagine ad un oggetto di tipo cv::Mat. Ci chiede il *path* e il modo con cui vogliamo importare l'immagine (possiamo usare delle macro).

```
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_ANY COLOR | cv::IMREAD_ANYDEPTH[a]	no

La funzione **`imwrite()`** permette di salvare sul filesystem l'immagine passata come secondo argomento.
Negli "eventuali parametri" possiamo specificare l'estensione.

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

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

OpenCV utilizza molte **strutture dati**, che possono essere divise in **3 categorie**:

1. **basic data types**: sono creati a partire dalle primitive del C++ e contengono le definizioni di vettori e matrici di piccole dimensioni, e semplici oggetti geometrici (punti, rettangoli).
2. **helper objects**: rappresentano concetti più astratti, come i *range objects*.
3. **large array types**: contengono array e altri contenitori di primitive, come cv::Mat.

OpenCV usa anche la STL, e in particolare la classe **vector**.

Sono molto usati i cv::Vec<>, che sono dei vector di dimensione prefissata.
In generale, ogni combinazione della seguente formula è valida:

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

dove il primo insieme rappresenta la *dimensione*, il secondo rappresenta il *tipo*.
(es. Vec2i è un vettore di 2 int).

In modo simile facciamo per le matrice cv::Matx<>, sempre di dimensione prefissata.
In generale, ogni combinazione della seguente formula è valida:

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

dove i primi due insiemi rappresentano le *dimensioni* e il terzo rappresenta il *tipo*.
(es. Matx22f è una matrice 2x2 di float).

cv::Mat

La classe **cv::Mat** rappresenta la classe centrale dell'intera implementazione C++ della libreria OpenCV. È la rappresentazione dell'immagine.
Non si tratta solo di un array di dati, ma contiene anche delle informazioni accessorie che dipendono dal tipo di dati che stiamo utilizzando, e che mi permettono di accedere correttamente a questi dati.

Tale classe è usata per rappresentare **array densi di un qualsiasi numero di dimensioni** (cioè, posso accedere a qualsiasi valore di riga/colonna).
In alternativa, è possibile usare cv::SparseMat per le matrici sparse.

I dati sono memorizzati in modalità **raster**, cioè attraverso un vector monodimensionale per riga, dove con "step" si intende il pixel successivo.
Nel caso di un'immagine in b/n, lo step comporta lo shift di 1 posizione. Nel caso di un'immagine a colori, lo step comporta lo shift di 3 posizioni (in ordine BGR, per questioni storiche).

Ogni matrice ha diversi **attributi**:

- `flags` (campi di bit) specifica il contenuto;
- `dims` indica il numero di dimensioni;
- `rows` e `cols` indicano il numero di righe e colonne;
- `data` è l'array che contiene i pixel, e si tratta di un puntatore all'area di memoria in cui sono memorizzati i dati.

L'array `step[]` contiene l'offset che devo aggiungere per ottenere il pixel richiesto.

Caso generale:

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

Array bidimensionale:

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

Esaminiamo: *allocazione*, *accesso agli elementi*, *op. algebriche, utilità*, *padding*.

• Allocazione

Posso **allocare** un'immagine in due modi:

1. attraverso il metodo `create()`;
2. attraverso un costruttore.

1. Istanziamo una variabile di tipo cv::Mat, che non ha ancora né tipo né dimensione. Successivamente è possibile allocare memoria utilizzando il metodo **create()** passando come argomenti: il numero di righe, il numero di colonne, e un tipo.

Per il *tipo*, bisogna specificare il tipo fondamentale e il numero di canali:

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

(es. CV_32FC3 corrisponde ad un array di float a 32 bit con 3 canali).

2. Per quanto riguarda il **costruttore**, possiamo distinguere vari tipi di inizializzazione:

Nell'inizializzazione possiamo anche specificare uno Scalar (3 valori float) con la quale inizializzare l'array bidimensionale. Posso creare l'immagine a partire da dati preesistenti, sfruttando un puntatore *data.
Possiamo definire lo step con diverse macro, ad es. AUTO_STEP.

È possibile creare matrici da oggetti di tipo Size, che contiene le dimensioni di righe e colonne (nb: ultime due sono come le prime due, ma usoSize invece di rows e cols).

Constructor	Description
`cv::Mat;`	Default constructor
`cv::Mat( int rows, int cols, int type );`	Two-dimensional arrays by type

`cv::Mat(`
`  int rows, int cols, int type,`
`  const Scalar& s`
`);`
	Two-dimensional arrays by type with initialization value

`cv::Mat(`
`  int rows, int cols, int type,`
`  void* data, size_t step=AUTO_STEP`
`);`
	Two-dimensional arrays by type with preexisting data

`cv::Mat( cv::Size sz, int type );`	Two-dimensional arrays by type (size in `sz`)

`cv::Mat(`
`  cv::Size sz,`
`  int type, const Scalar& s`
`);`
	Two-dimensional arrays by type with initialization value (size in `sz`)

L'altra possibilità è creare *oggetti multidimensionali*. Una matrice multidimensionale non deve essere confusa con una a più canali, in quanto la prima ha proprio più piani separati. Dovrò dichiare il numero di dimensioni.

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

Posso inoltre usare dei *costruttori di copia*, cioè: posso creare una matrice a partire da un'altra matrice.
Posso anche creare una copia partendo da un sottoinsieme della matrice (specifico range di righe e colonne), utile ad esempio se voglio lavorare su un insieme di pixel specifico.

Posso ottenere un sottoinsieme anche attraverso l'oggetto `Rect`: definisco le coordinate del pixel in alto a sinistra, e poi le due dimensioni width e heigth.

`Ranges` ci permette di definire un array di range per array multidim. (non lo useremo).

È anche possibile creare una matrice da espressioni logico/matematiche, e infatti restituiscono una MatExpr (simile a Mat).

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

La classe cv::Mat fornisce anche dei **metodi statici** per creare degli array di uso comune: di *zero*, di *uno*, *identica*.

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

Dunque, due *esempi di creazione* sono:

```
cv::Mat m( 3, 10, CV_32FC3, cv::Scalar( 1.0f, 0.0f, 1.0f ) );
```

Equivalente a

```
cv::Mat m;

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

• **Accesso agli elementi**

Ci sono diversi modi per accedere agli elementi di una matrice. I metodi principali sono:

- locazione;
- iterazione;
- a blocchi.

1. Per l'**accesso diretto**, l'approccio principale è il metodo templato **`at<>()`**.
Nel parametro dovrò specificare il tipo, e nelle parentesi tonde righe e colonne.

Nel caso avessi più colonne, dovrò usare i vettori `Vec` invece del tipo base.

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

È anche possibile accedere ad un'intera riga usando il metodo templato **ptr<>()**, che prende come argomento l'indice della riga e restituisce un puntatore al primo elemento.

2. Per l'**accesso con iteratori**, possiamo usare due iteratori templati:

- **`cv::MatIterator<>`** , permette la modifica:
- **`cv::MatConstIterator<>`** , non permette la modifica.

Inoltre, cv::Mat mette a disposizione i metodi **begin()** e **end()**.

L'iteratore deve essere dello stesso tipo di oggetto contenuto nell'array.

3. Per l'**accesso a blocchi**, che permette di estrarre delle sottomatrici attraverso il metodo **`operator()`**, passando come argomento: due cv::Range (uno per le righe, l'altro per le colonne), un oggetto di tipo cv::Rect.

In questo caso, i valori copiato fanno riferimento sempre alla prima matrice, dunque modificare tali valori è come modificare la prima matrice.

Example	Example
`m.row( i );`	`m.diag( d );`
`m.col( j );`	`m( cv::Range(i0,i1), cv::Range(j0,j1) );`
`m.rowRange( i0, i1 );`	`m( cv::Rect(i0,i1,w,h) );`
`m.rowRange( cv::Range( i0, i1 ) );`	
`m.colRange( j0, j1 );`	`m( ranges );`
`m.colRange( cv::Range( j0, j1 ) );`	

• **Operazioni algebriche**

Un'operazione algebrica restituisce un `MatExpr()`, che può però essere assegnato ad un `cv::Mat`.

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

Example	Description
`m0&m1; m0|m1; m0^m1; ~m0;` `m0&s; s&m0; m0|s; s|m0; m0^s; s^m0;`	Bitwise logical operators between matrices or matrix and a singleton
`min(m0,m1); max(m0,m1); min(m0,s);` `min(s,m0); max(m0,s); max(s,m0);`	Per element minimum and maximum between two matrices or a matrix and a singleton
`cv::abs( m0 );`	Per element absolute value of `m0`
`m0.cross( m1 ); m0.dot( m1 );`	Vector cross and dot product (vector cross product is defined only for 3×1 matrices)

`cv::Mat::eye( Nr, Nc, type );`
`cv::Mat::zeros( Nr, Nc, type );`
`cv::Mat::ones( Nr, Nc, type );`
	Class static matrix initializers that return fixed $N_r \times N_c$ matrices of type `type`

• **Utilità**

`clone` e `copyTo` sono identiche. In questo caso si tratta di una copia vera e propria, dunque non abbiamo più alcun riferimento.

Le *mask* sono utili sevogliamo estrarre delle aree specifiche arbitrarie (es. 1 cerchio), che possono anche essere disgiunte (es. 2 cerchi distinti).

> *Es*. creiamo una maschera della stessa dimensione dell'imm, e segno ad 1 tutti i pixel da estrarre. La copyTo con la maschera effettua una copia dell'immagine ma copiando solo i pixel specificati nella maschera.

La maschera posso usarla anche per impostare lo scalare di certe regioni.

Gli ultimi 3 metodi mi permettono di aggiungere righe e colonne alla matrice.

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

• **Padding**

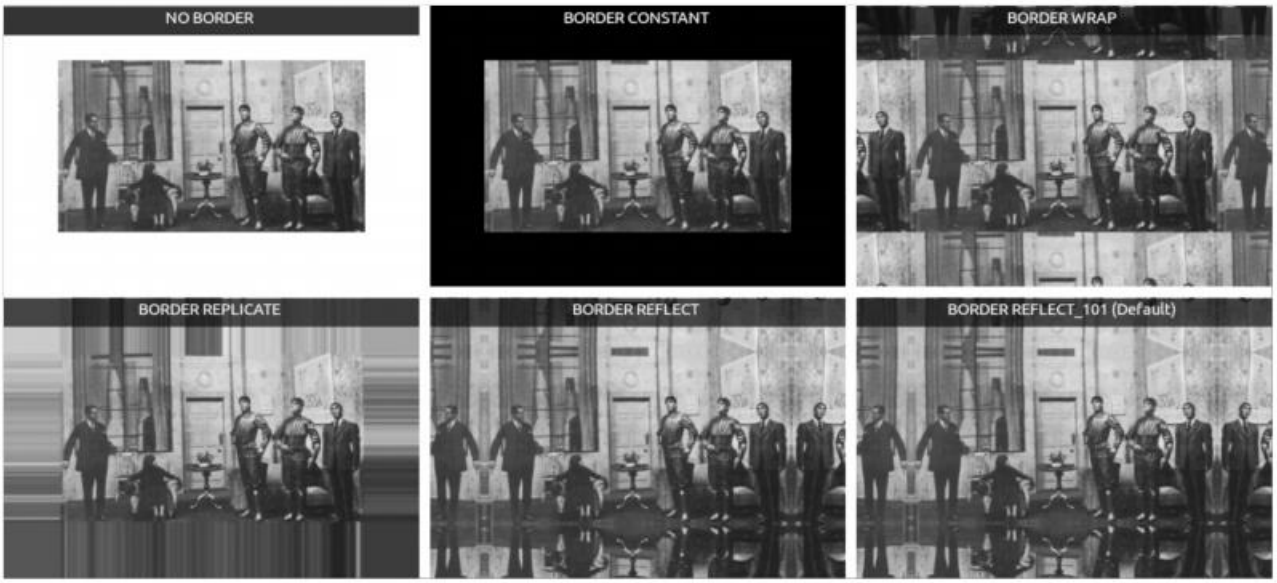
Il padding è un'operazione che consiste nell'aggiungere ad un'immagine righe e colonne extra.

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

Per effettuare il padding, possiamo usare la funzione **`copyMakeBorder()`**.

```
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
);
```

Possiamo distinguere diversi **tipi di padding**:



nb: Border_reflect(1 a 1). evita la copia dell'ultima colonna, ma parte direttamente dalla copia della penultima.

LEZ 3 – Filtraggio spaziale (1)

Il **filtraggio** è una tecnica che consente di far *passare* o *bloccare* alcuni elementi (frequenze).

Ci occupiamo di **due tipi di variazioni** d'intensità dei pixel dell'immagine: *repentini* (alte frequenze) e *graduali* (basse frequenze).

Gli **effetti** del processo di filtraggio sono: l'*attenuazione* (attenua le variazioni) e il *miglioramento dei dettagli* (esalta le differenze).

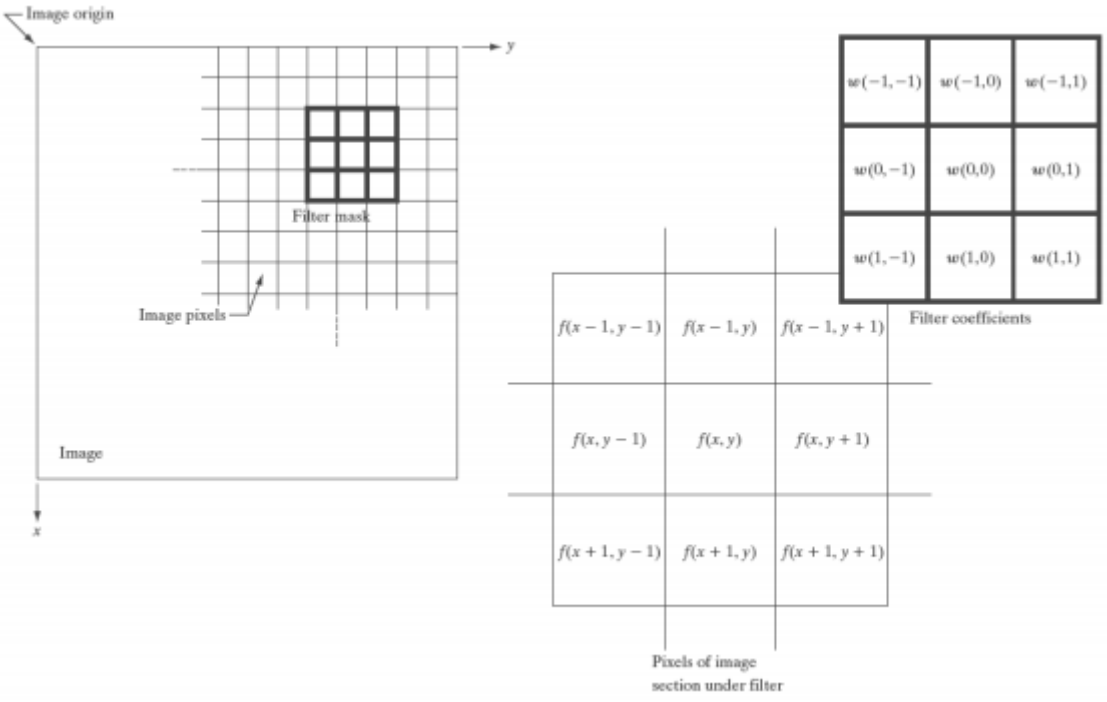
Le operazioni che permettono di smussare od esaltare i gradi di intensità prendono il nome di *filtraggio*.

Le tecniche di **filtraggio spaziale** operano sui pixel di un'immagine considerando il loro intorno (neighborhood).

La *regola di trasformazione* è spesso descritta da una matrice della stessa dimensione dell'intorno, chiamata **filtro** (aka maschera, kernel).
Se la regola di trasformazione è una funzione **lineare** dell'intensità dell'intorno, la tecnica è chiamata **filtraggio lineare spaziale** (altrimenti, non lineare).

Ogni valore dell'immagine è sovrapposto al suo filtro. Effettuo il prodotto puntuale (ie elem per elem), poi effettuo la somma di tutti i valori (vedi dopo, correlazione).

cioè, invece di sommare e dividere per 9 (es. 3x3), possiamo fare che ogni peso *w* valga 1/9, e poi moltiplico ogni valore per 1/9. Infine faccio la somma di tutto.
(Il vantaggio è che potrei impostare manualmente ogni peso).



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):

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

w(s,t) è il filtro, all'interno del quale sono presenti i pesi. Questi pesi vengono moltiplicati per i corrispondenti valori dell'immagine di input *f*.
a e *b* sono le dimensioni del filtro, solitamente una matrice con un numero **dispari** di righe (2a+1) e colonne (2b+1). Dobbiamo andare da [-a,-b] a [a,b] (es. [-1,-1] a [1,1] per una matrice 3x3, o [-2,-2] a [2,2] per una 5x5).
In output otteniamo l'immagine filtrata.

Questa formula ripetuta per ogni pixel prende il nome di *correlazione*.

CORRELAZIONE e CONVOLUZIONE

Con **correlazione** si intende il progressivo scorrimento di una maschera sull'immagine e nel calcolo della somma dei prodotti in ogni posizione.

Con **convoluzione** si intende la correlazione ma dove il filtro viene ruotato di 180°.
Una delle proprietà fondamentali è che la convoluzione di una funzione con l'impulso unitario produce una copia della funzione nella posizione dell'impulso.

La correlazione produce lo stesso risultato, ma l'immagine è ruotata di 180°. Per questo, per applicare la convoluzione: dobbiamo effettuare una *pre-rotazione* del filtro di 180°; dobbiamo applicare la correlazione.

- **Correlazione e Convoluzione 1D**

Prendiamo in considerazione un impulso unitario discreto (1D), così detto in quanto si presenta un solo 1, e un filtro *w*.

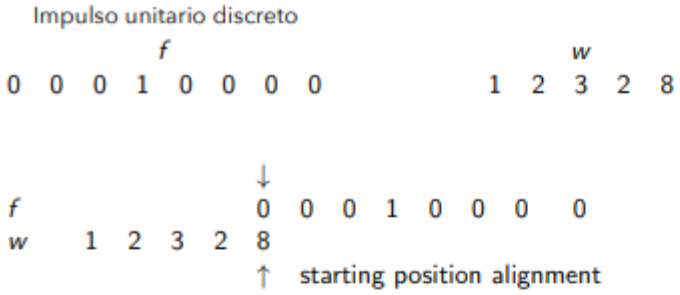
Affinché possa applicare il filtro, dovrò effettuare un zero padding sia a sinistra che a destra del size di w-1, e poi all'inizio dovrò allineare l'ultimo elemento del filtro con il primo elemento dell'impulso.
Ad ogni passo, effettuerò il prodotto w*f.
Infine, effettuerò il cropping (tolgo il passing).

Il risultato sarà il segnale modificato dal filtro, che altro non è che il filtro stesso ma ruotato di 180°.

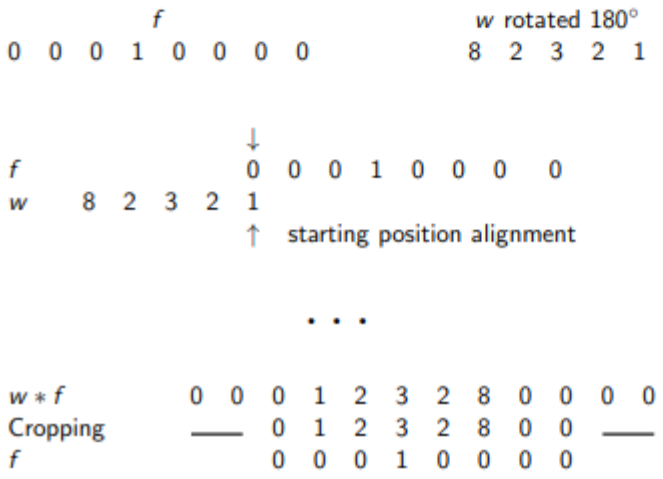
Vien da se che, se vogliamo ottenere il filtro originale dobbiamo effettuare la *convoluzione*. Dunque all'inizio pre-ruotiamo il filtro di 180° e applichiamo normalmente la correlazione.

CORRELAZIONE 1-D

Impulso unitario discreto



CONVOLUZIONE



- **Correlazione e convoluzione 2D**

La formula della correlazione l'abbiamo vista prima.
La formula della *convoluzione* è semplicemente questa formula ma negando *s* e *t*, in quanto così facendo ci troviamo esattamente a 180° rispetto a prima.

Correlazione

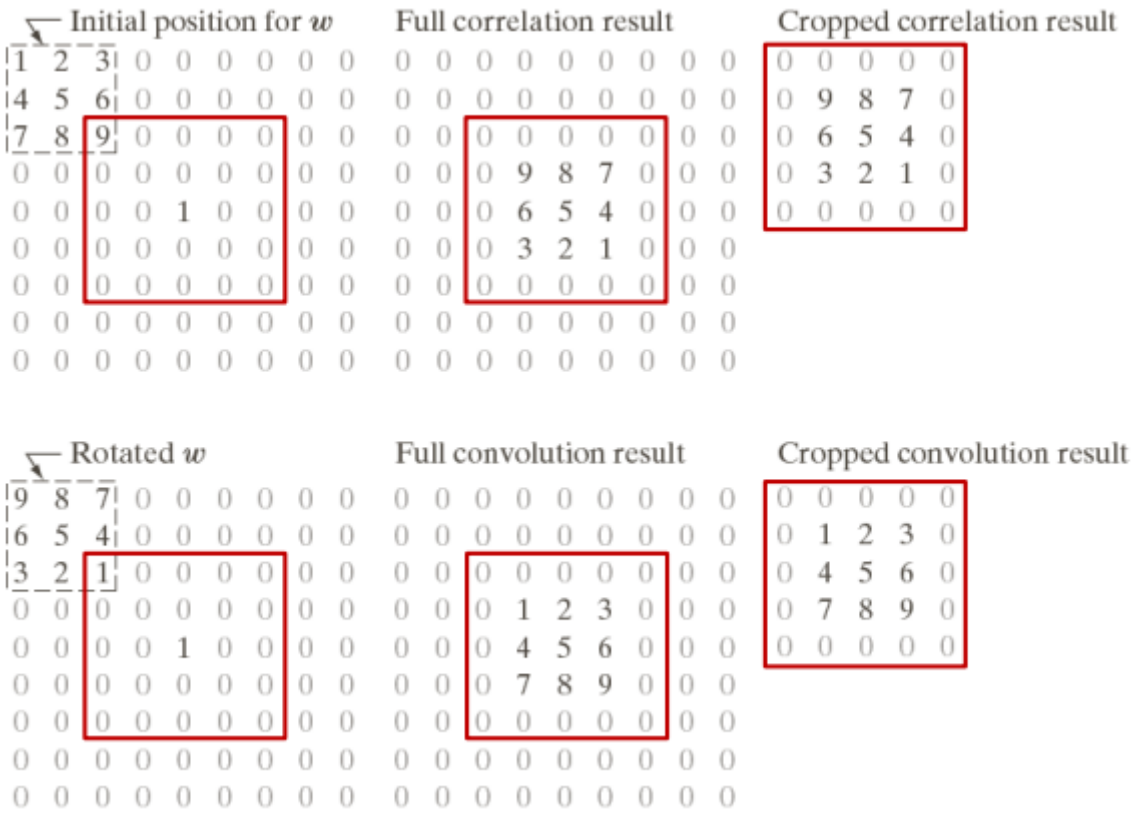
$$g(x, y) = w(x, y) \star f(x, y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s, t)\, f(x+s, y+t)$$

Convoluzione

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

Per poter effettuare queste operazioni, anche in questo caso dovremmo inserire un padding, che nel caso di un filtro 3x3 consiste nell'inserire 2 linee di zero ad ogni lato.

La posizione iniziale del filtro sarà nel primo pixel in alto a sinistra, e lo facciamo scorrere su tutta l'immagine. Il risultato "croppato" sarà il filtro ruotato di 180° nel caso della correlazione, o il filtro originale nel caso della convoluzione.



Rappresentazione vettoriale

Quando la posizione relativa dei coefficienti non è importante, è possibile rappresentare la risposta del filtro usando una rappresentazione vettoriale.

Deve essere specificata una indicizzazione convenzionale.

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

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

L'operazione viene vista come w(trasposto)*z, dunque è un prodotto righe per colonne. (w e z di solito sono vettori colonna).

CORRELAZIONE e CONVOLUZIONE in OPENCV

In OpenCV è possibile effettuare la **correlazione** usando la funzione **cv::filter2D()**. (gli ultimi 3 valori possiamo lasciarli di default).

```
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 **convoluzione** dobbiamo ruotare il filtro di 180° attraverso la funzione **rotate()**.

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

`rotateCode` indica di quanto vogliamo ruotare l'immagine. Abbiamo 3 possibilità:

- ROTATE_90_CLOCKWISE
- ROTATE_180
- ROTATE_90_COUNTERCLOCKWISE

TIPI DI FILTRI

Per creare un **filtro lineare** spaziale è necessario specificarne i coefficienti:

- Specifica diretta:

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

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

- Specifica basata su una funzione:

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

Per i **filtri non lineare**, si specifica solo l'intorno su cui si applicheranno determinate operazioni (es. filtro "max" su un intorno 5x5).

Filtri di smoothing

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

Operano **rimuovendo** i dettagli più piccoli, evidenziando invece gli oggetti più grandi e riempendo piccoli gap (es. interruzioni di linee o curve).

Per la riduzione del rumore, a seconda del tipo, possono essere utilizzati filtri lineari o non lineari.

I **filtri di media** sono filtri di smoothing *lineari*. Sono anche detti *passa basso* (low pass) in quanto enfatizzano le basse frequenze e attenuano le alte frequenze.

Il valore di ogni pixel viene sostituito con la media dei livelli di intensità nell'intorno definito dalla maschera. In particolar modo, distinguiamo due tipi di operatori:

- **operatore di media** (o *box filter*): tutti i coefficienti della maschera sono uguali ad 1;
- **operatore di media pesata**: i pesi sono inversamente proporzionali alla distanza dal centro (dunque il centro avrà peso maggiore, mentre gli angoli sono quelli con peso minore). Ciò riduce gli effetti indesiderati di blurring.

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

1	1	1
1	1	1
1	1	1

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

1	2	1
2	4	2
1	2	1

Filtro media ponderata

Più aumento le dimensioni del filtro, maggiore sarà la sfocatura.
Quello che notiamo è che gli oggetti più piccoli si perdono, mentre quelli più grandi si evidenziano, ed anche se non siamo in grado di riconoscerli sappiamo che in quell'area c'è qualcosa che possiamo estrarre.

In OpenCV esistono diverse **funzioni di smoothing**, fra cui: `blur()`, `boxfilter()` e `GaussianBlur()`.

```
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
);
```

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

`blur()` è la funzione di smoothing più “semplice”.
`boxFilter()` è una generalizzazione di blur, in quanto presenta un parametro in più (ddepth).
`GaussianBlur()` determina i pesi attraverso una funzione gaussiana (una campana 3d). I valori in cima alla campana sono i più alti, e più si scende e più questi diminuiscono. L’ampiezza e l’altezza della campana dipendono dai valori sigmaX e sigmaY (se lasciati a 0, viene calcolato un valore ottimale in modo autonomo).

Effetti

Immagine originale | Filtro media 15x15 | Filtro media 35x35

I **filtri non lineari basati su statistiche d'ordine** sono filtri la cui risposta consiste nel:

- ordinare i pixel contenuti nell'intorno definito dal filtro;
- sostituire il valore del pixel centrale con un valore dell'insieme ordinato.

Ne sono esempi il filtro *mediano*, *max e min*, *basati su percentile*.

Ad esempio,

Dato l'intorno *f* contenuto nel filtro, *ordiniamone* i valori (le intensità). Poi dobbiamo decidere con quale valore dell'insieme ordinato dobbiamo sostituire il pixel centrale (ricordiamo che si tratta del più importante in quanto è quello con peso maggiore).

Sebbene effettuare una *media* potrebbe sembrare una buona idea, in realtà il valore calcolato raramente appartiene all'insieme ordinato. Per questo spesso si preferisce prendere la *mediana*.

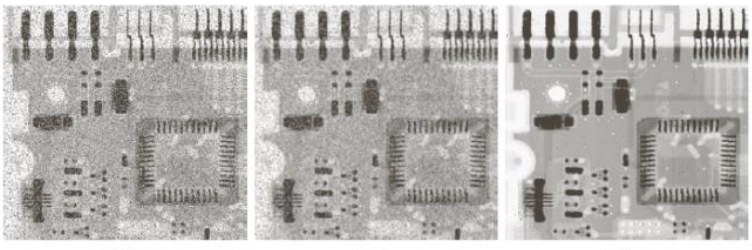
Se sostituisco il pixel centrale con il *minimo* otterrò un'immagine più scura. Viceversa, con il *massimo* otterrò un'immagine più chiara.

- f = [100, 120, 98, 99, 110, 255, 100, 200, 10]
- Ordinato: [10, 98, 99, 100, 100, 110, 120, 200, 255]
- Media: 121
- Mediana: 100
- Min: 10
- Max: 255

10	98	99
100	100	110
120	200	255

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

(a)Immagine originale, corrotta da rumore sale e pepe
(b)Immagine filtrata con filtro media 3 x 3
(c)Immagine filtrata con filtro mediano 3 x 3



(a) (b) (c)

In OpenCV, per applicare il filtro mediano si utilizza la funzione `medianBlur()`.

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

Procedimento per estrarre gli oggetti più importanti

Si vogliono estrarre gli oggetti più importanti da un'immagine (a).

(b) si deve innanzitutto effettuare un'operazione di smoothing, così da perdere gli oggetti più piccoli.

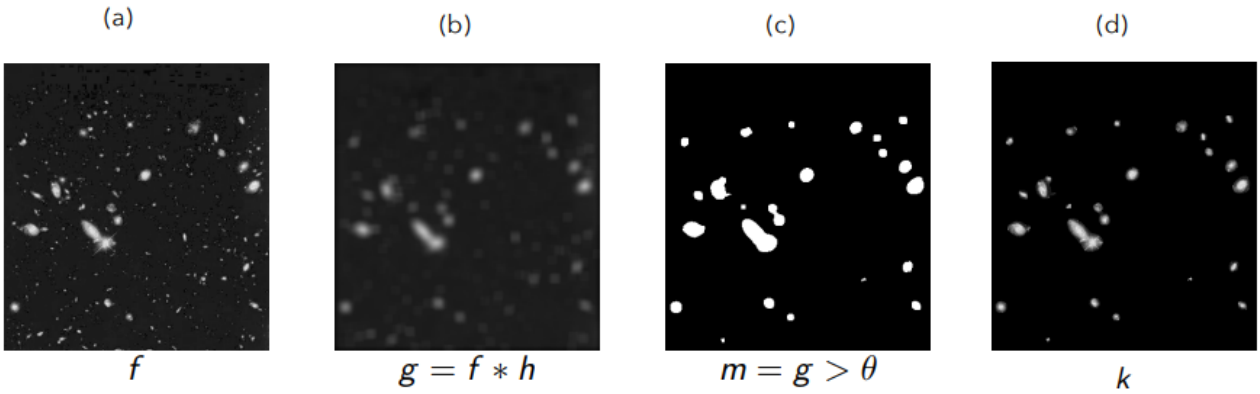
(c) poi si devono individuare le aree sopravvisute. Allora applico una "soglia" θ (es. il 25% dell'intensità massima, cioè 64 su 255). Se l'intensità del pixel è > θ, allora il pixel sarà posto ad 1 (altrimenti a 0). Così facendo creo una maschera.

(d) Sovrappongo la maschera all'immagine di input, ed effettuo un prodotto puntuale. Ovviamente se moltiplico un pixel per 0 otterò 0, e dunque quel pixel non verrà riportato nell'immagine finale. Se fosse 1, viene riportato.

In generale, maggiore è la dimensione del filtro, maggiore sarà la dimensione degli oggetti che andremo ad estrarre.

nb: non confondere 1 e 255, perché il prodotto con 1 mi serve solo per capire quali pixel prendere e quali no.

(Per l'esame, capire la soglia sarà importante. L'immagine potremo obv vederla).



(a) f, immagine 353 x 382

(b) g, immagine filtrata con filtro di smooth 13 x 13

(c) m, immagine con sogliatura (θ=25% dell'intensità massima)

(d) k, immagine mascherata: k(x, y) = f(x,y) x m(x, y)

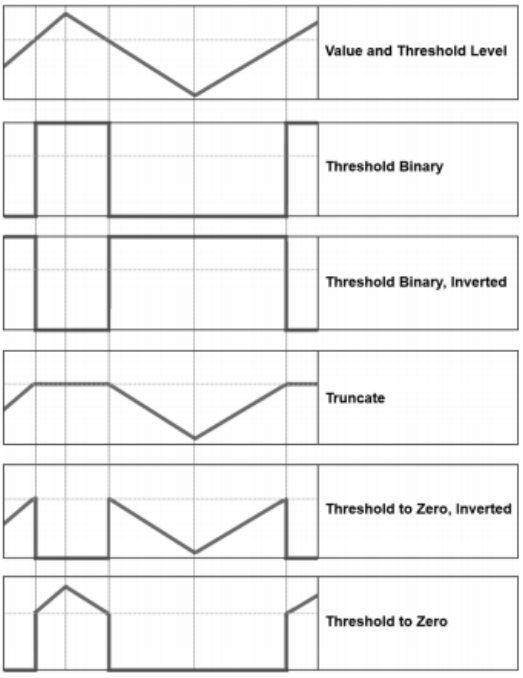
In OpenCV, esistono diverse **funzioni per effettuare la sogliatura** (thresholding):

```
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
);
```

Possiamo distinguere diverse **tipologie** di thresholding. La più importante è quella binaria.

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	$DST_i = (SRC_i > thresh)\ ?\ 0 : SRC_i$



Il thresholding adattivo considera anche l'intorno. L'adaptiveMethod può essere:

- cv::ADAPTIVE_THRESH_MEAN_C
- cv::ADAPTIVE_THRESH_GAUSSIAN_C

blocksize è la dimensione dell'intorno in cui viene calcolata la media pesata.
C è una costante che viene sottratta alla media calcolata.

nb: inoltre consideriamo un altro Threshold type: cv::THRESH_OTSU (metodo di Otsu).

LEZ 4 – Filtraggio spaziale (2)

Sharpening

Il termine **sharpening** si riferisce alle tecniche adatte ad evidenziare le transizioni di intensità. Attraverso il cambio di intensità siamo in grado di individuare i bordi, e più nette sono queste transizioni più sarà definita l'immagine.

Viene utilizzato il concetto di **derivata** (discreta), in quanto ci da la misura della quantità di variazione fra due pixel. Maggiore è la variazione di intensità, maggiore sarà la risposta di questi filtri derivativi.

In aree di intensità costante la variazione sarà nulla, mentre se ci troviamo su un bordo la variazione sarà forte. Più sono piccole le variazioni di intensità, più sarà difficile individuare l'edge in quanto la risposta del filtro derivativo sarà minore.

• Derivata prima di un'immagine

Poiché l'immagine è una funzione discreta, la definizione classica di derivata non può essere applicata. Per questo, è necessario definire un operatore che soddisfi le principali **proprietà della derivata prima**:

1. uguale a 0 dove l'intensità è costante;
2. diversa da 0 per una transizione di intensità;
3. costante sulle rampe in cui la transizione di intensità è costante.

Per implementare queste proprietà usiamo un'approssimazione: la **differenziazione spaziale**, ovvero le *differenze in avanti* (così dette perché sono date dall'elemento successivo meno l'elemento corrente). Si tratta della differenza tra l'intensità dei pixel vicini.

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

• Derivata seconda di un'immagine

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

1. uguale a 0 dove l'intensità è costante;
2. diversa da 0 all'inizio di una rampa (o passo) di intensità;
3. uguale a 0 sulle pendenze costanti delle rampe.

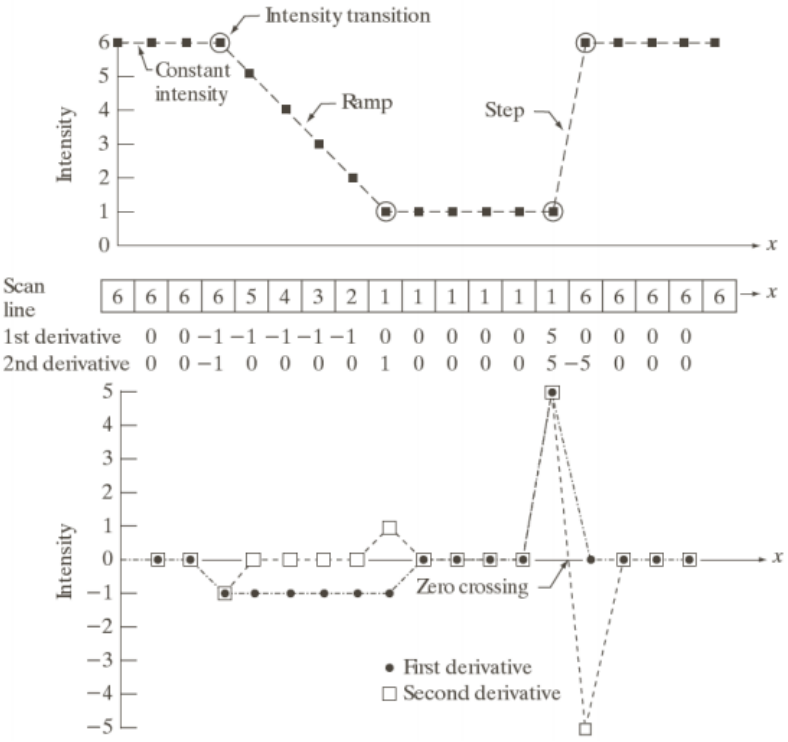
Per implementare queste proprietà utilizzo le *differenze centrali*, ovvero faccio una differenza in avanti e una all'indietro. Tuttavia se ciò è vero sto considerando due volte il pixel centrale (f(x)), e dunque il valore del pixel centrale lo sottraggo a quello successivo e a quello precedente:

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

Nel caso di una intensità uniforme, avremo: 255 – 255*2 + 255 = 0.

Esempio di derivata discreta

Immaginiamo di vedere le intensità di profilo.



Vediamo le risposta della derivata prima e seconda. Sotto abbiamo il profilo delle derivate, sopra il profilo dell'immagine.

All'inizio è costante, poi ho una rampa, di nuovo costante, poi uno *step* (brusco cambio di intensità), e infine di nuovo costante.

Nella *derivata prima*, osserviamo che per tutta la rampa l'intensità rimane costante (-1) e la risposta su tutta la rampa sarà diversa da 0.

Nella *derivata seconda*, osserviamo un picco, uno zero, poi più avanti un altro picco e un altro zero. È riconoscibile proprio per via di questi piccoli.

Questo comportamento lo vediamo anche sullo step.
Notiamo uno "zero crossing", è un punto in cui la retta che passa per i due valori attraversa lo 0.

Operatore: Laplaciano

Il **Laplaciano** implementa una **derivata del secondo ordine 2D**.
In particolare, vogliamo realizzare **filtri isotropici** la cui risposta è indipendente dalla direzione della discontinuità dell'immagine (invarianti per rotazione).

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

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

In un'immagine digitale, le derivate secondo rispetto ad *x* e ad *y* sono calcolate come:

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

Quindi, il Laplaciano risulta:

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

Può essere considerata anche la derivata lungo la diagonale. In tal caso, il pixel centrale sarà considerato 8 volte.

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

In particolare:

filtro Laplaciano invariante alle rotazioni di 90° (f. isotropico a 90°).
Il pixel centrale pesa 4, i pixel laterali pesano 1, quelli diagonali 0.

0	1	0
1	-4	1
0	1	0

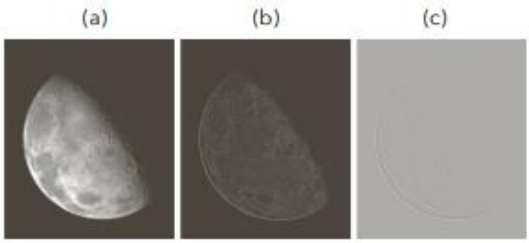
filtro Laplaciano invariante alle rotazioni di 45°. (f. isotropico a 45°).
Il pixel centrale pesa 8, i pixel laterali pesano e diagonali pesano 1.

1	1	1
1	-8	1
1	1	1

Esempio di applicazione di filtro Laplaciano

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

Per effettuare lo sharpening, sommiamo l'immagine al laplaciano dell'immagine per una certa costante *c*. Più *c* si avvicina a -1, più esalto i bordi. Se c=0, non ottengo alcun effetto.



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

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

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

con il filtro a 90° notiamo un miglioramento, ma non risulta essere il filtro migliore siccome prende gli edge solo ai lati (e non è la situazione migliore in quanto ho molte forme circolari, e praticamente perde tutti gli edge diagonali, tipo i crateri).
Infatti con il filtro a 45° abbiamo un miglioramento maggiore, in quanto considera anche le direzioni in diagonale.

Ovviamente se ho un'immagine con angoli squadrati mi conviene usare quella a 90°, anche perché con quella a 45° potrebbe individuare dei contorni che alla fine sarebbero solo fastidiosi.
Invece, se ho a che fare con un'immagine con angoli circolari mi conviene usare quella a 45 gradi.

Se non so che cosa usare (perché non conosco l'immagine) di solito si usa sempre quella a 45 gradi, in quanto in generale da risultati migliori.

In OpenCV, la funzione che implementa il Laplaciano è:

```
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
);
```

se *ksize* = *3*, Laplaciano a 45°. Se *ksize* = 1, Laplaciano a 90°.

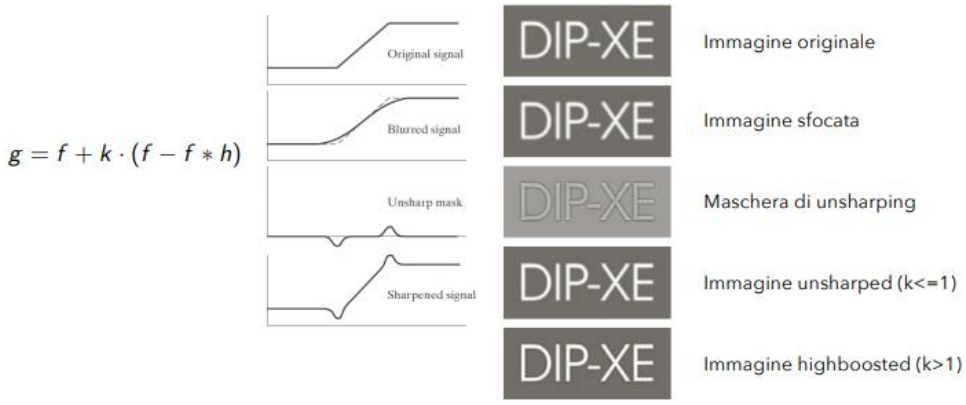
Operatore: unsharp masking

È un metodo di uso comune in grafica per rendere l'immagini più nitide. Si tratta di una tecnica più di grafica che di elim, in quanto suppone che l'immagine non sia affetta da rumore.

L'idea è prendere l'immagine, effettuare lo smoothing, e sottrarre dall'immagine originale quella smussata. Da questa sottrazione ottengo la *sharp mask*:

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

Se l'immagine fosse affetta da rumore, verrebbe esaltato tutto, e il risultato sarebbe un completo caos.



Gradiente

La **derivata prima** viene implementata con un'approssimazione del **gradiente**.

Il gradiente è un **vettore** formato dalle sue **derivate parziali** (in pratica, faccio la derivata rispetto ad *x* e ad *y*, e queste due derivate sono le componenti del vettore gradiente):

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

Essendo un vettore ha due informazioni in più: la direzione e la lunghezza (magnitudo).

La **magnitudo del gradiente**, M(x,y) è un'immagine delle stesse dimensioni di f().
Più la lunghezza è grande, maggiore è la variazione. Per calcolarla dobbiamo fare la radice quadrata della somma delle derivate parziali al quadrato, ma siccome la sqrt è un'operazione onerosa possiamo fare la più banale somma dei valori assoluti.

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

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

il vettore gradiente punta sempre nella direzione di massima variazione (ma per ora ci interessa solo la magnitudo).

Possiamo distinguere diversi **operatori di derivazione**:

a) definizioni base;

b) operatori di **Roberts**: sono filtri 2x2, ma sono difficili da applicare in quanto si preferisce usare filtri con righe/colonne “dispari”. Infatti non si usa molto;

c) opeatori di **Sobel**: sono filtri 3x3 nei quali è anche inglobata un’operazione per ridurre un po’ il rumore (sarebbe il -2 e il 2);

d) opeatori di **Prewitt**: sono filtri 3x3 ma che non includono operazioni per ridurre il rumore (infatti ci sono solo 1 e -1).

a)

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

g_x:

-1	1

g_y:

-1
1

b)

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

g_x:

-1	0
0	1

g_y:

0	-1
1	0

c)

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

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

g_x:

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

g_y:

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

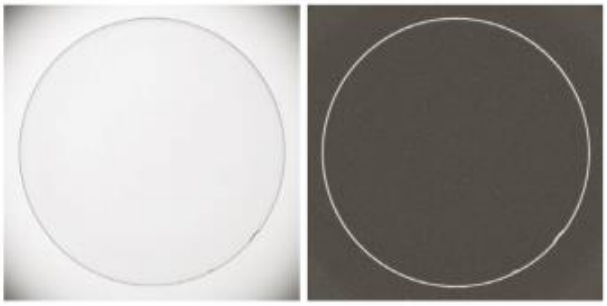
d)

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

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

Esempio Sobel

Il filtraggio di **Sobel** riduce la visibilità di quelle regioni in cui l’intensità cambia lentamente, permettendo di **evidenziare i dettagli** (rendendo l’individuazione dei difetti più semplice per un’elaborazione automatica).



*notiamo come si riescano a notare alcune “imperfezioni” nel cerchio.

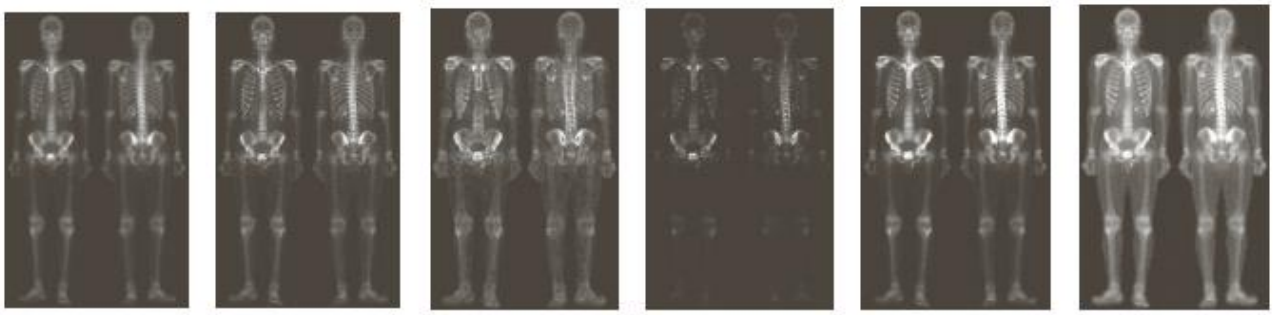
In OpenCV, la funzione che implementa il filtro di Sobel è:

```
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 (la magnitudo sarebbe xorder=1, yorder=1).

Gli altri valori possiamo lasciarli a quelli di default.

nb: In realtà è difficile che venga usata una sola tecnica per migliorare l'immagine, ma spesso viene usata una combinazione di queste



Operazioni utili

- dst = src1 +/- src2
- dst = src * k
- dst = abs(src)
- pow(src,2,dst)
- sqrt(src,dst)

LEZ 5 – Colore

Il **colore** è un descrittore che ci aiuta ad identificare ed estrarre gli oggetti/dettagli da una scena.

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

- **full-color**: i colori sono acquisiti da un sensore full-color (es. fotocamera);
- **falsi colori**: vengono assegnati dei colori a delle immagini non acquisite a colori (in tal caso il colore è un'etichetta che viene assegnata ad una certa intensità).

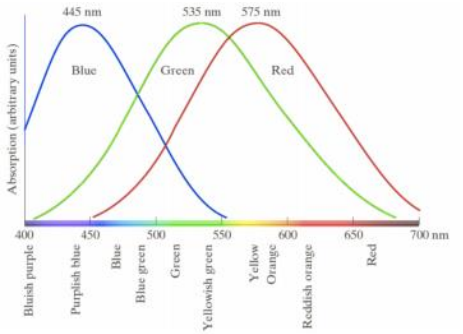
Il motivo per cui **noi percepiamo i colori** è perché questi colori colpiscono gli oggetti, che riflettono una parte della lunghezza d'onda (mentre le altre li assorbono).

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

- coni sensibili alla **luce rossa**, 65%;
- coni sensibili alla **luce verde**, 33%;
- coni sensibili alla **luce blu**, 2% (ma sono anche i più sensibili).

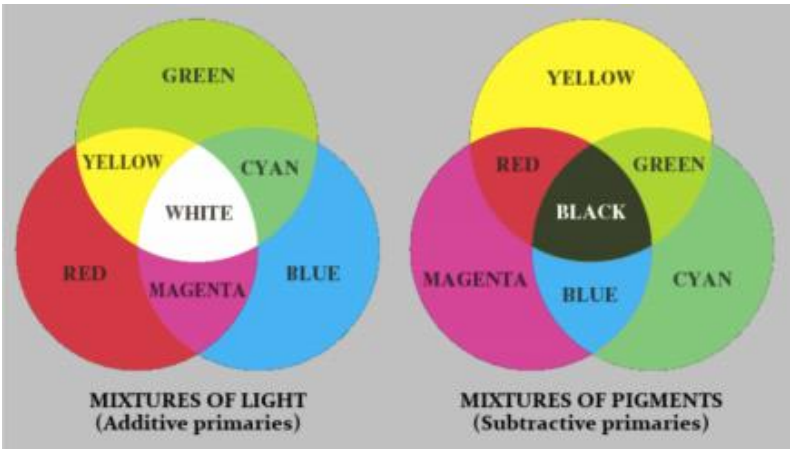
Le caratteristiche dell'occhio consentono di vedere i colori come **combinazioni** variabili dei **colori primari** (RGB, Red Green Blue).

Possiamo ottenere un nuovo colore andando a sommare le frequenze d'onda di queste 3 bande. Anche per questo sono detti *colori additivi*.



I colori primari della luce possono essere **mescolati** per produrre i **colori secondari** della luce: **magenta** (rosso+blu), **ciano** (verde+blu), **giallo** (rosso+verde).

Tali colori prendono anche il nome di **colori primari dei pigmenti**: ogni pigmento assorbe una gamma di colori riflettendo solo quelle che il nostro occhio percepisce. Se aggiungiamo un pigmento sopra un altro, la loro unione **assorbirà sempre più frequenze** rendendo la somma sempre più scura (fino a giungere al nero). Anche per questo motivo sono detti *colori sottrattivi*.



La differenza è che uno è una miscela di *luce*, l'altro è una miscela di *pigmenti*. Notare anche come siano complementari.

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

- **luminosità**: misura l'intensità;
- **tonalità** (Hue): identifica la lunghezza d'onda dominante;
- **saturazione**: misura la purezza della tonalità (ie la quantità di bianco mescolato alla tonalità).

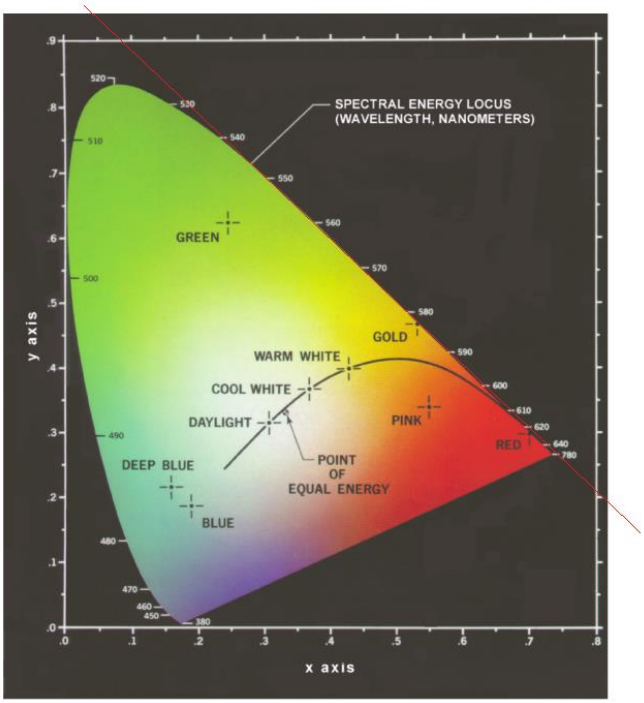
Tonalità e *saturazione* insieme vengono dette **cromaticità**.

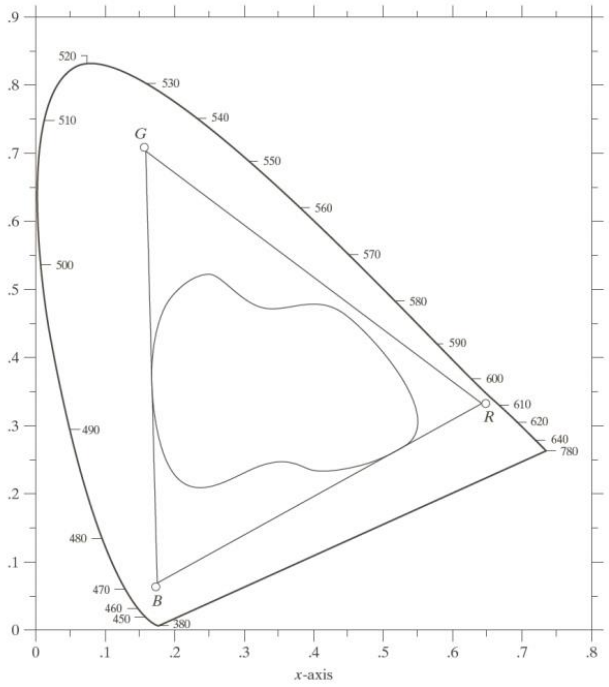
Le **quantità** di **rosso**, **verde** e **blu** necessarie per formare un colore sono dette **valori tristimolo** (X, Y, Z). Un colore viene specificato mediante i **coefficienti tricromatici**, calcolati come:

$$x = \frac{X}{X+Y+Z} \qquad y = \frac{Y}{X+Y+Z} \qquad z = \frac{Z}{X+Y+Z} \qquad \Rightarrow \qquad x+y+z=1$$

Posso normalizzare i valori tristimolo in un intervallo [0,1]. Questi 3 coefficienti tricromatici saranno quindi compresi tra 0 ed 1, e la loro somma sarà 1.

Un **altro modo** per specificare il colore è mediante il **diagramma di cromaticità**, che mostra la composizione del colore in funzione x (rosso) e y (verde).
Dalla relazione x+y+z=1, è possibile ricavare il valore z (blu).





Nel **diagramma di cromaticità** sono riportati tutti i colori generabili e questi giacciono dentro il triangolo rettangolo di coordinate (0,0), (0,1), (1,0). All'interno di questo triangolo è tracciato il **diagramma CIE*** dei colori reali, il quale è a forma di campana e racchiude tutte le tinte possibili. Al di fuori della campana (ma sempre all'interno del triangolo) ci sono tutti i colori non visibili (o non distinguibili dal nostro occhio). Il diagramma CIE gode, proprio per il modo in cui è stato generato, di alcune importanti caratteristiche:

*CIE = Commissione Internazionale per l'illuminazione.

- Il punto in cui le **tre componenti** assumono **valore uguale** prende il nome di **punto di uguale energia** (bianco).
- I punti sul bordo del diagramma sono **completamente saturi** (senza bianco).
- Più ci avviciamo al punto di uguale energia, più **diminuisce la saturazione**.
- Un **segmento lineare** che unisce due punti individua tutte le possibili variazioni di colore ottenibili mediante **combinazione lineare dei due colori** (es. il segmento che va dal blu al verde rappresenta tutti i possibili colori dal blu al verde).
- È possibile identificare il "**calore**" di un colore attraverso la **black body curve**.
- All'**interno della campana** è possibile tracciare un **ulteriore triangolo** che contiene tutti i colori che è possibile rappresentare attraverso un monitor.
- All'**interno di questo ulteriore triangolo** è possibile tracciare un'**area irregolare** che contiene tutti i colori è possibile rappresentare dalle stampanti.

Modelli colore

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

I modelli colori più utilizzati sono:

- **RGB**: per monitor/telecamere;
- **CMY** e **CMYK**: per le stampanti;
- **HSI** (HSV, HSL): particolarmente orientato alla prospettiva umana, essendo basato sulla percezione che si ha di un colore in termini di tonalità, saturazione e luminosità.

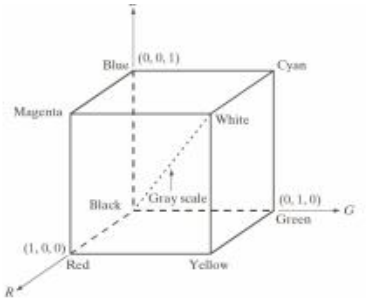
- RGB

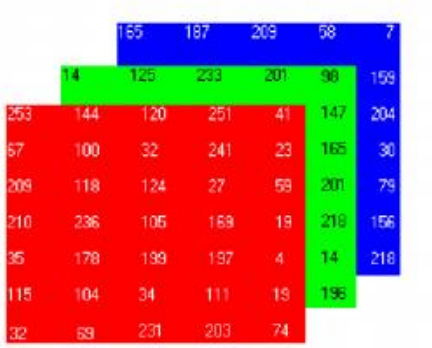
Nel modello RGB, ogni colore è rappresentato dai **primari della luce**: rosso, verde e blu.

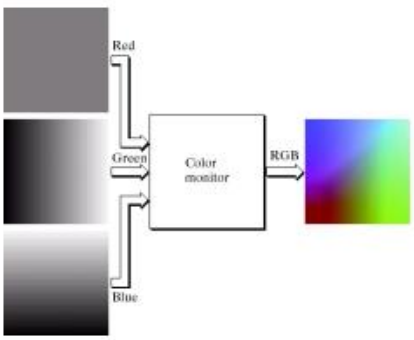
Si basa su un sistema di coordinate cartesiane, ed il **sottospazio** di interesse è un **cubo** in cui i colori primari, i colori secondari, il bianco e il nero occupano gli 8 vertici.
I colori secondari si trovano all'incrocio di due primari. Il bianco e il nero si trovano in vertici opposti (nero = (0,0); bianco = (1,1)).

Dentro questo cubo troviamo tutti i possibili colori (ovviamente, dall'immagine si possono osservare solo le facce esterne).

Le immagini rappresentate nel modello RGB sono **formate da 3 immagini**, una per ogni colore primario. In genere, ognuna delle immagini rossa, verde e blu è un'immagine ad **8 bit**, per cui la profondità di un pixel RGB è **24 bit** (3 piani x 8 bit).







- CMY, CMYK

Nel modello CMY, ogni colore è rappresentato dai **primari di pigmenti**: ciano, magenta e giallo.

Questi spazi colori sono più utilizzati nelle stampanti (e non molto nelle elim).

Per effettuare le **conversioni** da RGB a CMY (e viceversa) basta ricordare che i primari dei pigmenti sono i complementari dei primari della luce, dunque:

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

Tuttavia, se uso solo CMY non sono in grado di ottenere il **nero puro** (al massimo ottengo un marrone scuro), per questo aggiungo un quarto colore **nero K** (*key black*).

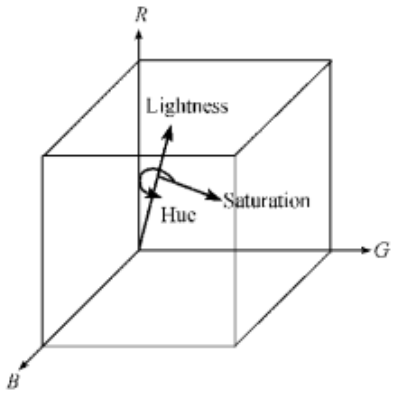
- HSI

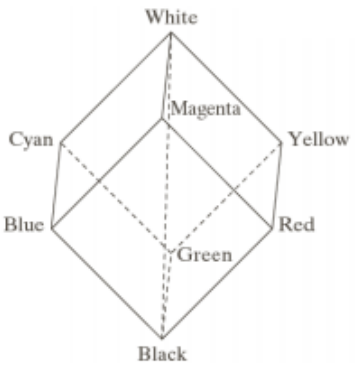
Nel modello HSI, ogni colore è rappresentato da **3 caratteristiche**: tonalità, saturazione e intensità (luminosità).

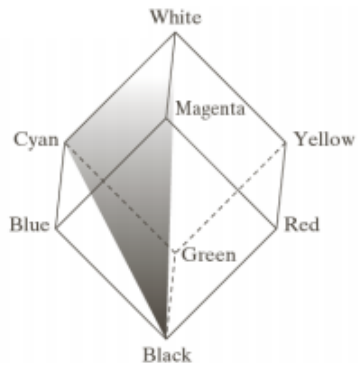
Immaginiamo che dato un punto colore RGB, vogliamo calcolare il suo HSI.

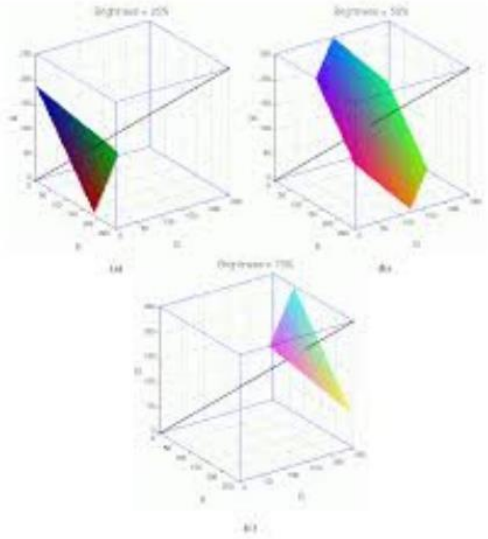
- Per l'**intensità**, bisogna far passare un piano perpendicolare all'asse di intensità che contiene il punto colore; il punto di intersezione sarà il valore di intensità.
- Per la **saturazione**, bisogna calcolare la distanza del punto dall'asse di intensità. Unendo il *bianco*, il *nero* ed un punto colore, tutti i punti del triangolo ottenuto sono caratterizzati dalla stessa tonalità;
- Per la **tonalità**, questa dipende dall'**angolo** rispetto ad un punto (solitamente rosso primario a 0°, passando per il verde primario a 120° e il blu primario a 240°, e quindi tornando al rosso a 360°). La tonalità aumenta in senso orario.

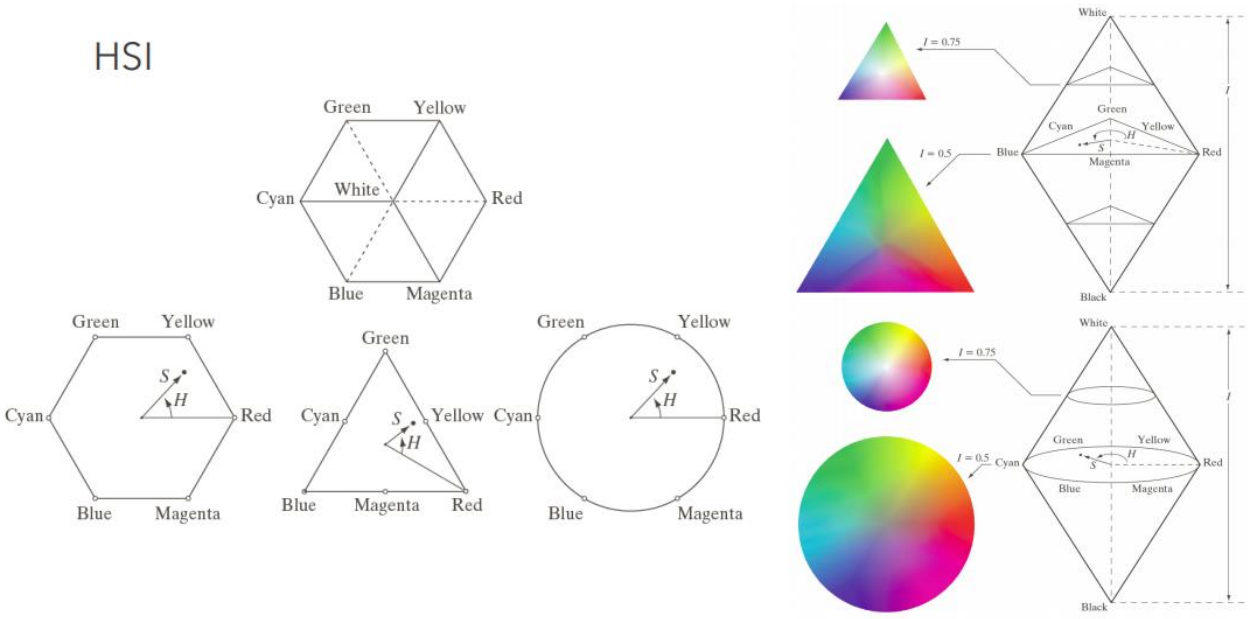
Se faccio ruotare questo triangolo, ottengo un cono. Tuttavia, la forma ottenuta potrebbe fuoriuscire dal cubo (per questo potrebbe risultare "tagliato").
Posso spostarmi a livelli superiori/inferiori di questo cono. Più salgo, più i valori saranno luminosi (fino ad arrivare al bianco); più scendo, meno i valori saranno luminosi (fino ad arrivare al nero).



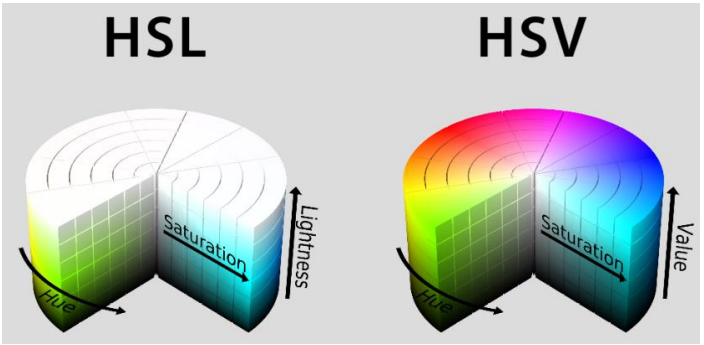








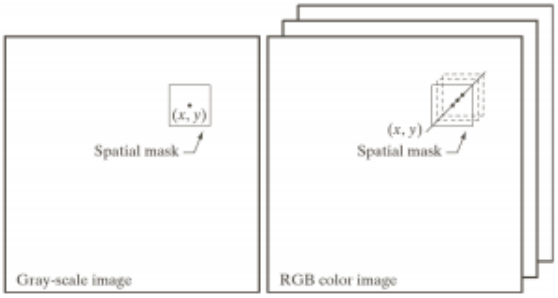
nb: l'HSI (intensity) e l'HSV (value) sono la stessa cosa. Tuttavia, l'HSI (HSV) è una variante del modello HSL (-lightness).
La differenza è che in *HSL* un colore con massima luminosità è bianco puro, mentre in *HSI* è sempre quel colore ma più intenso.



Elaborazioni full-color

Possiamo distinguere **due metodi** di elaborazioni delle immagini **full color**:

1. ogni componente viene elaborata separatamente e combinata con le altre;
2. vengono elaborati i pixel colore (tutte le componenti sono elaborate insieme).



- Smoothing

Lo **smoothing RGB** di un'immagine si può realizzare con un'operazione di filtraggio spaziale ed una maschera opportuna.
Il valore di ogni pixel viene sostituito con la **media** dei valori dei pixel nell'intorno definito nella maschera.

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

Lo **smoothing HSI** ci permette di eseguire lo smoothing solo sulla componente di intensità, lasciando inalterate la tonalità e la saturazione.
Infatti, così facendo preservo la componnete cromatica ma smusso i picchi di intensità.

Sharpening

Lo **sharpening RGB** 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.

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

Lo **sharpening HSI** ci permette di eseguire lo sharpening solo sulla componente di intensità, lasciando inalterate la tonalità e la saturazione (anche in questo caso l'edge riguarda l'intensità, e preservo la componente cromatica).

nb: se facciamo il laplaciano, troveremo dei bordi di vari colori che corrispondono all'intensità dell'edge all'interno di quella componente colore.

In OpenCV, possiamo aprire un'immagine a colori usando la funzione `imread()` usando la macro `IMREAD_COLOR`.

Le immagini sono caricate in **matrici a 3 canali** con una profondità di 8 bit a canale.

- se l'immagine è a colori, viene allocata una matrice con tre canali in formato BGR;
- se l'immagine è a scala di grigi, viene comunque allocata una matrice con tre canali dove viene replicato il livello di grigio per ogni canale.

È anche possibile usare `IMREAD_ANY_COLOR`, così che le immagini vengano caricati in matrici con il numero di canali corretto.

Possiamo fare l'**accesso ai canali** come:

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

Possiamo modificare lo **spazio colore** con cui è rappresentata un'immagine attraverso la funzione `cvtColor()`, che ha 4 parametri:

- 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')
);
```

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

nb (?): il prof dice che in OpenCV non esiste HSI, ma solo HSV e HSL. Tuttavia, HSI e HSV dovrebbero essere la stessa cosa.

LEZ 6 – Introduzione alla segmentazione

Denotiamo con **R** la regione occupata dall'immagine. La **segmentazione** dell'immagine consiste nel partizionare R in *n* sottoregioni R_1, R_2, …, R_n tali che:

1. la loro unione mi dia la regione completa;
2. R_i è un insieme connesso;
3. le regioni sono disgiunte. Di conseguenza, ogni pixel appartiene ad una sola regione;
4. tutti i pixel che appartengono ad una regione devono rispettare il predicato logico Q;
5. applicato il predicato Q a due regioni, il risultato deve essere falso.

In breve, la **segmentazione** è il processo attraverso cui un'immagine viene suddivisa in più regioni disgiunte. La qualità della segmentazione è fondamentale in quanto può determinare l'esito delle elaborazioni successive.

Il problema della segmentazione è detto "mal posto", in quanto non c'è una soluzione univoca, cioè in una stessa immagine potremmo identificare più o meno regioni in base alla situazione.

La maggior parte degli algoritmi sfruttano le *discontinuità* o le *similirità*:

- se si sfruttano le **discontinuità**, la segmentazione sarà guidata da bruschi cambiamenti di intensità. Dall'unione di tutti i punti dove è avvenuta la discontinuità ottengo un bordo (*edge*).

  Possiamo distinguere diversi algoritmi:

  - Canny: edge detector;
  - Harris: corner detector;
  - Hough: permette di cercare forme specifiche (es. rette, circonferenze e in generale qualsiasi forma parametrica).

- se si sfruttano le **similarità**, si raggruppano pixel simili in base a dei criteri di similarità.

  Possiamo distinguere diversi algoritmi:

  - Sogliatura: tutti i pixel che hanno valori di intensità compresi in un certo *range* vengono assegnati ad un certo gruppo (es. tutti i pixel compresi fra 0 ed x appartengono alla regione A, tutti i pixel compresi fra x e y alla regione B, ecc…);
  - Region growing: partendo da un pixel, gliene aggiungo altri in base alla regola di similarità adottata;
  - Split and merge: parto dall'immagine originale e la divido in regioni (regolari ad esempio). Se la regola di similarità è rispettata abbiamo finito, altrimenti suddivido ulteriormente la regione;
  - Clustering: si suddivide ulteriormente in k-means e mean-shift.

Consideriamo i 3 tipi di **adiacenza** rispetto ai **valori** in un insieme V:

- **4-adiacenza**: due pixel *p* e *q* con valori in V seguono la 4-adiacenza se *q* appartiene al 4-intorno di *p* (ovvero se appartiene ad una di quelle 4 posizioni).
- **8-adiacenza**: due pixel *p* e *q* con valori in V seguono la 4-adiacenza se *q* appartiene al 8-intorno di *p*.
- **m-adiacenza**: distinguiamo due casi:
    - q appartiene al 4-intorno di p, oppure
    - q appartiene all'intorno diagonale di *p* e nessun pixel nell'intersezione fra il 4-intorno di *p* e il 4-intorno di *q* ha valori in V.

Ricordiamo che l'm-adiacenza è usata per evitare l'ambiguità che deriva dall'uso della 8-adiacenza. Ad esempio, quando un pixel è legato a più pixel.

Un **path** da *p* a *q* è una sequenza di pixel adiacenti dal pixel *p* al pixel *q*.

Due pixel *p* e *q* sono detti *connessi in S* se esiste un percorso tra di essi formato interamente di pixel appartenenti ad S.

Per ciascun pixel *p* in *S*, l'insieme dei pixel che sono connessi ad esso è detto **componente connessa** di *S*.
Se si ha un'unica componente connessa, allora l'insieme *S* è detto **insieme connesso**.

Una **regione dell'immagine** *R* è un qualsiasi sottoinsieme di pixel connesso.

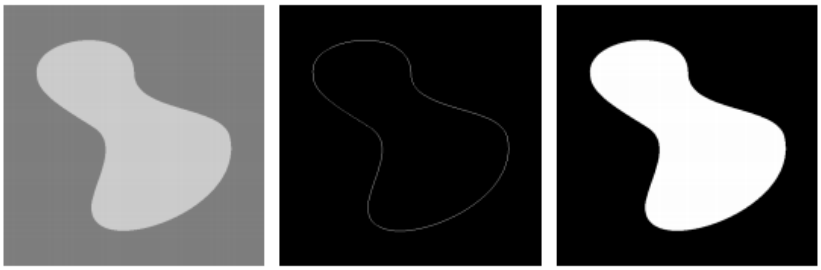
Segmentazione edge based

Se si sfruttano le **discontinuità**, si assume che i bordi siano sufficientemente diversi tra le regioni e dallo sfondo in modo da poter sfruttare le intensità locali.

In un'immagine sfrutto la discontinuità attraverso il concetto di derivata, ovvero individuo quei punti dove avviene un cambio repentino di intensità.
Dove l'intensità è costante la risposta della derivata sarà 0 (es. il background e l'interno del foreground), altrimenti sarà diversa da 0 (dunque l'edge).

Posso creare una maschera ponendo ad 1 tutti i pixel all'interno di questa regione, e a 0 tutto il resto.

Tuttavia non sempre ho una situazione così conveniente dove tutti i pixel di una regione hanno intensità costante: se anche fosse presente un po' di rumore questo approccio (e dunque la derivata) non va bene, ma posso sfruttare la similarità fra pixel.

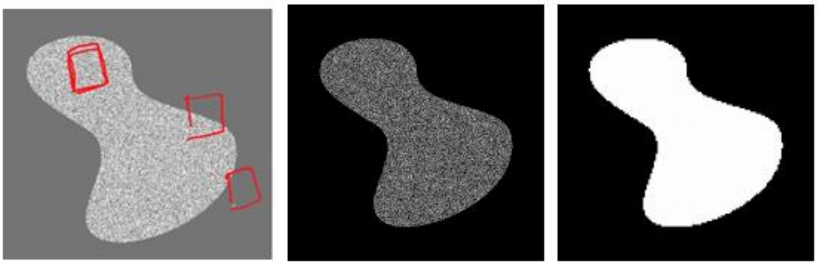


Segmentazione region based

Se si sfruttano le **similarità**, si partiziona l'immagine in regioni simili in base ad un criterio di similarità.

L'idea è che se in una certa regione i pixel non variano più di tanto, allora quei pixel appartengono a quella regione. Per fare questo, si tiene in considerazione il valore medio di un intorno.

Ovviamente le variazioni calcolate in questo modo saranno sempre più basse nelle regioni all'interno della forma rispetto a quelle che si trovano sui bordi: dunque rispetto alla derivata non ho una risposta puntuale, ma vado a "blocchi" (infatti l'edge è frastagliato).



Individuazione delle caratteristiche di un'immagine

Siamo interessati a **tre caratteristiche** di un'immagine:

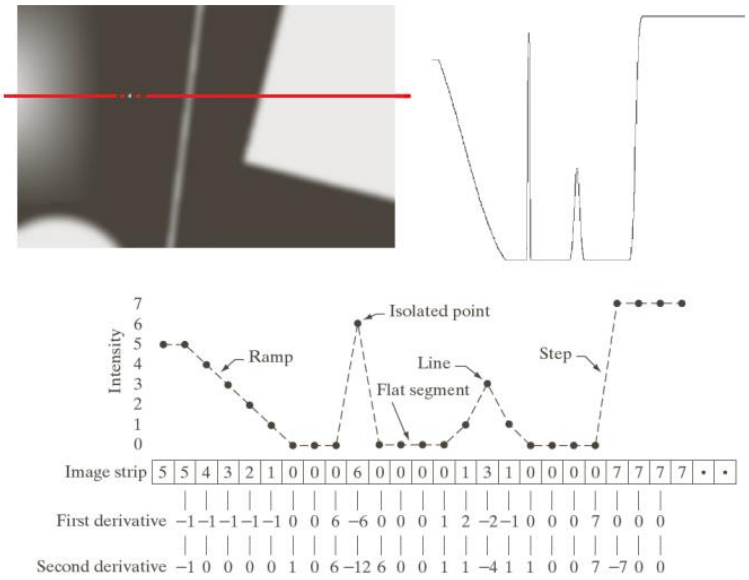
- **edge**: insieme di pixel in cui si presenta una repentina variazione di intensità;
- **linee**: segmenti di edge in cui l'intensità ai lati della linea è minore o maggiore dell'intensità dei pixel sulla linea;
- **punti**: linee di lunghezza e larghezza pari a 1 pixel.

Per **individuare le variazioni** di intensità a cui siamo interessati, utilizziamo le **approssimazioni** delle **derivate prime e seconde** definite in termini di differenze (ie differenze in avanti per le derivate prime, differenze centrali per le derivate seconde).

Ricordiamo i concetti delle derivate e del laplaciano nella LEZ 4.

Nel seguente esempio analizziamo le *caratteristiche delle derivate*:

Sulla sinistra abbiamo l'immagine originale, sulle destra abbiamo la linea di scansione del suo profilo (praticamente abbiamo una linea orizzontale che passa per l'immagine e la "scansiona"). Notiamo i picchi: il primo picco è un punto bianco (molto repentino), poi ho un altro picco (dove sta la retta quasi verticale) e infine ho una variazione che poi rimane costante.



Da qui possiamo osservare le **proprietà** delle derivate prima e seconda:

- in presenza di **edge di rampa**, la derivata seconda produce edge sottili mentre la derivata prima produce edge spessi;
- la **risposta** della derivata seconda in presenza di punti isolati è più forte rispetto a quella derivata prima;
- sia sugli **edge di rampa** che su quelli a **gradino**, la derivata seconda ha **segni opposti**;
- il **segno** della derivata seconda può essere utilizzato per determinare se un edge è una transizione chiaro/scuro o viceversa.

Per i **punti isolati**, dopo aver applicato il *filtro laplaciano*, si utilizza la **soglia** sulla risposta del filtro per determinare i punti di discontinuità:

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

dove *g* è l'immagine di output, *T* è una soglia non negativa ed *R* è la risposta del filtro (valore assoluto perché il laplaciano in ingresso e in uscita alla rampa hanno un valore positivo ed uno negativo).

L'intensità di un punto isolato sarà abbastanza diversa dalle intensità dei suoi 8 vicini.
La differenza di intensità tra i punti vicini viene controllata dal *parametro T*.

Per le **linee**, anche in questo caso si può utilizzare il Laplaciano, ma bisogna gestire la doppia risposta della derivata seconda.
A tal fine, è possibile utilizzare:

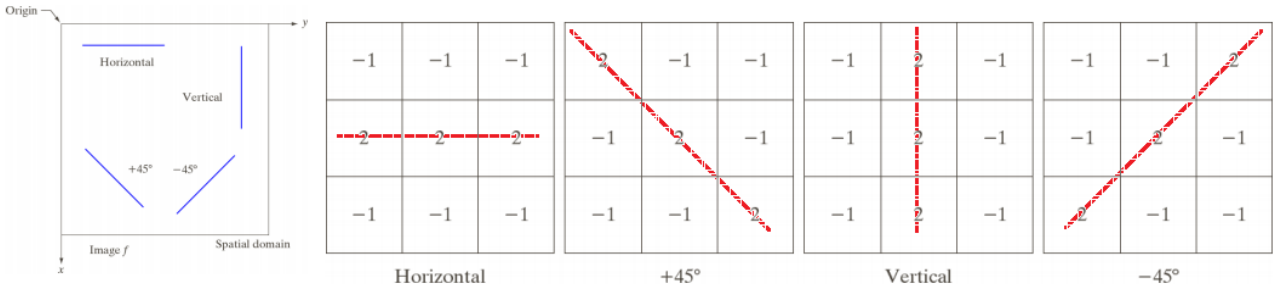
- il *valore assoluto* della risposta; oppure
- solo i *valori positivi* (eventualmente sogliati per attenuare l'effetto del rumore).

Nel *primo caso*, in uscita mi troverò una situazione del tipo "alto, basso, alto" (che in realtà sarebbe "positivo, costante, negativo" ma consideriamo i valori assoluti). Tuttavia non sappiamo dove passa l'edge, dunque otteniamo linee più spesse.
Nel *secondo caso* si ottengono linee più sottili.

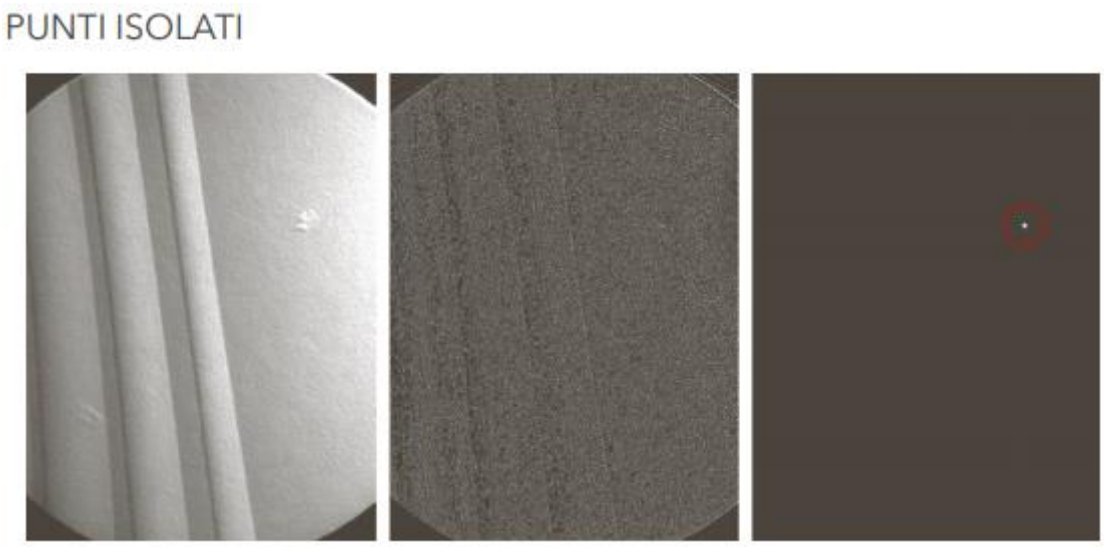
Se le linee sono più ampie rispetto alla dimensione del filtro, si otterrà l'effetto di una "valle" di valori nulli che separe le due linee.

nb: il filtro laplaciano è *isotropico*, ovvero è indipendente dalla direzione.
Per individuare **rette con direzioni specifiche** è possibile utilizzare **filtri specifici**.

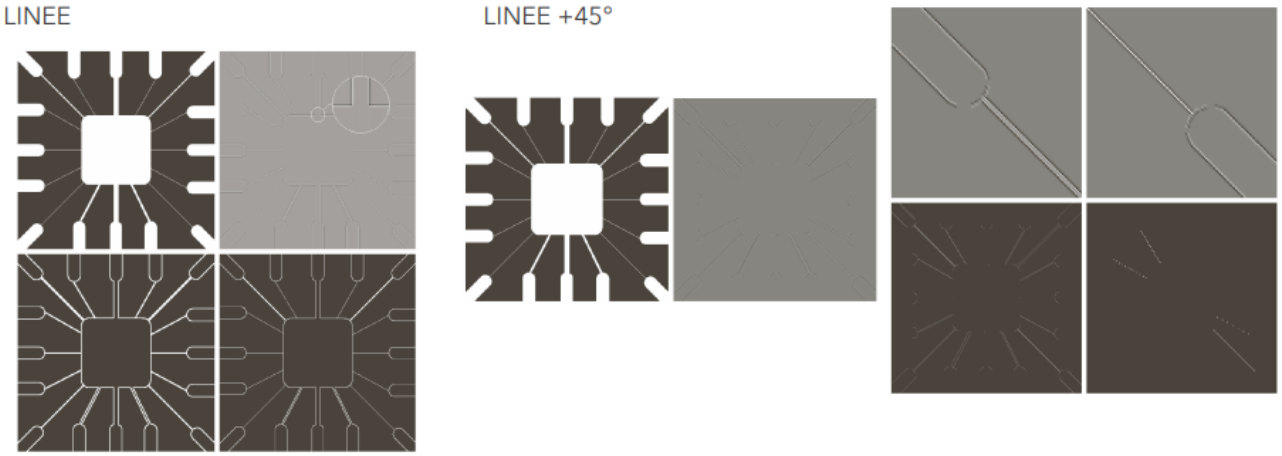
Per capire qual è la direzione individuata dal filtro, dobbiamo porre il filtro come un piano (x,y) e osservare l'angolo formato dai valori positivi.
Ad esempio, il filtro a 45 gradi presenta una diagonale "passante per i 2" che va dall'alto verso il basso, da sinistra verso destra.



esempi:



Applicare una soglia significa che tolgo da mezzo tutte le risposte più basse. Quel punto risalta perché è sopra la soglia.



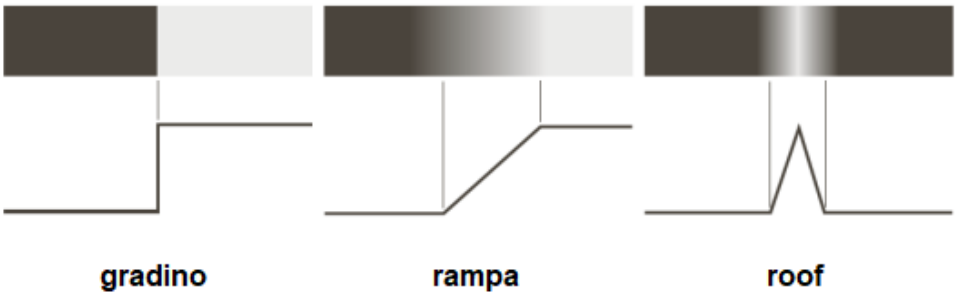
In “linee”, in basso a sx ho delle linee più spesse (considero solo i valori assoluti) mentre in basso a dx ho linee più sottili (considero anche quelli negativi).

Modelli di edge

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

- **edge a gradino**: transizione tra due 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 "a gradino". Difficilmente si tratta di linee sottili;
- **roof edge**: associato al bordo di una regione, ha una base determinata dallo spessore e dalla sfocatura della linea;

spesso nelle immagini si trovano più tipi di edge con profili diversi da quelli ideali.

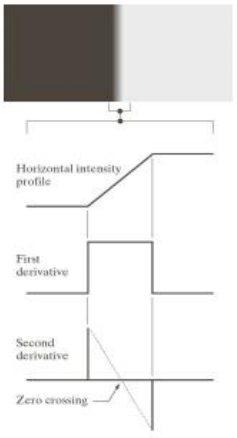


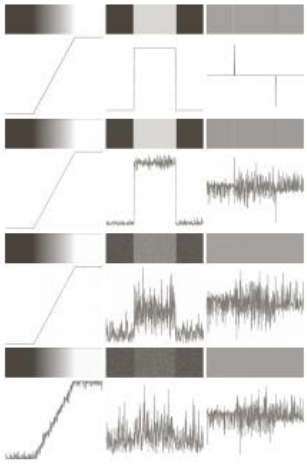
• Edge a rampa

L'edge a rampa può essere individuato utilizzando la derivata prima.
Per determinare se un pixel si trova sul lato chiaro o sul lato scuro si può utilizzare il *segno* della derivata seconda.

La derivata seconda produce due valori per ogni edge (opposti l'uno all'altro), e possiamo usare lo **zero-crossing** (attraversamento dello zero) per trovare il centro di un edge.

Notare che sebbene la derivata seconda dia una risposta più forte rispetto alla derivata prima, una risposta più forte significa anche che esalta molto di più il rumore.
Per ridurre il rumore, possiamo applicare lo smoothing.





nb: rumore gaussiano con media nulla e variazione standard 0.1, 1.0 e 10.0. Notiamo come nel 2 e 3 esempio ho dei picchi della derivata seconda che sono superiori ai bordi veri e propri, e dunque quando farò la sogliatura avrò dei bordi non veri.

Individuazione basata su gradiente

Ricordiamo la LEZ4:

[Il gradiente è un **vettore** formato dalle sue **derivate parziali** (in pratica, faccio la derivata rispetto ad *x* e ad *y*, e queste due derivate sono le componenti del vettore gradiente):

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

Essendo un vettore ha due informazioni in più: la direzione e la lunghezza (magnitudo).

La **magnitudo del gradiente**, M(x,y) è un'immagine delle stesse dimensioni di f(). Più la lunghezza è grande, maggiore è la variazione. Per calcolarla dobbiamo fare la radice quadrata della somma delle derivate parziali al quadrato, ma siccome la sqrt è un'operazione onerosa possiamo fare la più banale somma dei valori assoluti.

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

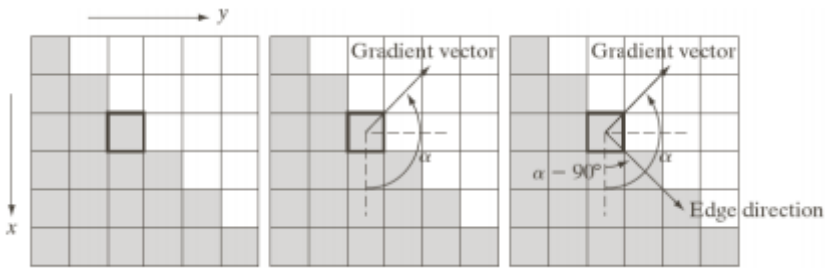
$$M(x, y) \approx |g_x| + |g_y|$$

il vettore gradiente punta sempre nella direzione di massima variazione].

La direzione del vettore gradiente è data dall'arcotangente del rapporto delle due componenti g_y e g_x.

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

Poiché il vettore gradiente punta sempre nella direzione di massima variazione, la direzione del vettore gradiente è **ortogonale** alla direzione dell'edge.



Ricordiamo inoltre gli *operatori di derivazione della LEZ4*.

[Possiamo distinguere diversi **operatori di derivazione**:

a) definizioni base;

b) operatori di **Roberts**: sono filtri 2x2, ma sono difficili da applicare in quanto si preferisce usare filtri con righe/colonne "dispari". Infatti non si usa molto;

c) opeatori di **Sobel**: sono filtri 3x3 nei quali è anche inglobata un'operazione per ridurre un po' il rumore (sarebbe il -2 e il 2);

d) opeatori di **Prewitt**: sono filtri 3x3 ma che non includono operazioni per ridurre il rumore (infatti ci sono solo 1 e -1).

a)

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

g_x:

-1	1

g_y:

-1
1

b)

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

g_x:

-1	0
0	1

g_y:

0	-1
1	0

c)

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

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

g_x:

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

g_y:

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

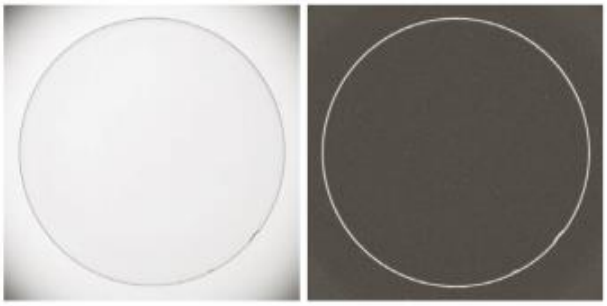
d)

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

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

Esempio Sobel

Il filtraggio di **Sobel** riduce la visibilità di quelle regioni in cui l'intensità cambia lentamente, permettendo di **evidenziare i dettagli** (rendendo l'individuazione dei difetti più semplice per un'elaborazione automatica).



*notiamo come si riescano a notare alcune "imperfezioni" nel cerchio. **]**

La risoluzione dell'immagine può determinare l'esito dell'operazione di estrazione degli edge.

I *dettagli fini* sono assimilati al rumore, che per un algoritmo sono terribili in quanto sono informazioni che non gli servono, e dunque comportano la produzione di molti *punti edge*. Prima di applicare un estrattore di edge è buona norma effettuare uno smoothing dell'immagine.

In base al tipo di edge che vogliamo enfatizzare (diagonali, verticali, orizzontali) è necessario utilizzare maschere specifiche.

Per ottenere **immagini gradiente** più "pulite" è anche possibile utilizzare la tecnica del **thresholding** (sogliatura).

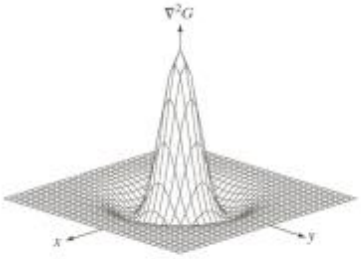
Algoritmo di Marr-Hildreth

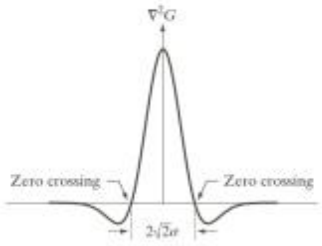
L'algoritmo di Marr-Hildreth è un algoritmo che cerca di sfruttare l'idea dello smoothing/thresholding.

In particolar modo, questi due personaggi proposero il filtro **Laplacian of Gaussian (LoG**, o $\nabla^2 G$ (nabla)).

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

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





0	0	−1	0	0
0	−1	−2	−1	0
−1	−2	16	−2	−1
0	−1	−2	−1	0
0	0	−1	0	0

In pratica: faccio il laplaciano della funzione gaussiana, e il filtro che ottengo lo applico a tutta l'immagine. L'effetto sarà di trovare gli edge.
La funzione gaussiana dipende da sigma: più è piccola, minore sarà l'effetto di smoothing.

Questa funzione ha un termine positivo e centrale circondato da una regione negativa adiacente i cui valori aumentano in funzione della distanza dall'origine e una regione esterna nulla. I coefficienti devono dare come somma zero in modo che la risposta della maschera sia zero nelle aree a intensità costante.

La dimensione del filtro **n** deve essere scelta pari al più piccolo intero dispari maggiore o uguale a **6σ**.

Es: σ = 4, n=25.

In OpenCV, è possibile ottenere un filtro Gaussiano attraverso la f. **`getGaussianKernel`**:

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

Ovviamente, possiamo poi utilizzare la funzione filter2D (o la nostra funzione di convoluzione) per poter applicare il filtro.

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

Proprietà del LoG

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

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

equivale a:

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

ovvero filtrare l'immagine con un **filtro Gaussiano** di dimensione nxn e poi applicare il **filtro Laplaciano**.
Infine, bisogna individuare gli *zero-crossing*.

Zero-crossing

Con zero crossing intendiamo il punto in cui avviene la transizione fra positivo e negativo, dunque si tratta della ricerca di g(x,y)=0.

Si consideri un intorno 3x3 centrano in un pixel *p* dell'immagine filtrata, uno zero-crossing in *p* implica che i segni di almeno due pixel vicini opposti siano diversi (sopra-sotto, destra-sinistra e le due diagonali).

Inoltre, il *valore assoluto della loro differenza* deve superare una **soglia**.

Il **LoG** può essere approssimato con **DoG (Differences of Gaussian**).

LEZ 7 – Algoritmo di Canny e di Harris

Algoritmo di Canny

L'algoritmo di Canny è uno dei migliori algoritmi per l'individuazione degli edge.
Questo si basa su tre obiettivi di base:

- **basso tasso di errore**: deve ridurre al minimo la probabilità di falsi positivi e falsi negativi;
- **punti di edge ben localizzati**: gli edge rilevati devono essere il più vicino possibile agli edge reali, cioè la distanza tra il punto rilevato e quello reale deve essere minima;
- **risposta puntuale per edge singolo**: deve restituire un solo punto per ogni punto di edge, dunque non dovrebbe individuare edge multipli dove esiste un unico punto di edge.

Questo algoritmo si contraddistingue per la sua rigida formulazione matematica.

Canny dimostrò che una buona approssimazione per l'individuazione di edge a gradino è la **derivata prima della Gaussiana** (nb: tiene conto solo della x):

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

La direzione e l'intensità dell'edge non sono conosciute a priori, ma possiamo sfruttare la definizione di gradiente per poter recuperare queste due informazioni.
Infatti il vettore gradiente punta nella *direzione* di massima variazione, dunque è ortogonale all'edge, e possiamo calcolare quanto sia forte tale variazione calcolando la *magnitudo*.

Data un'immagine di input f(x,y) e la **funzione Gaussiana** G(x,y), dove:

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

per conoscere **direzione** e **magnitudo** dobbiamo innanzitutto effettuare la **convoluzione** fra l'immagine di input f(x,y) e la funzione Gaussiana G(x,y):

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

Avendo adesso $f_s(x,y)$, possiamo calcolare le **derivate parziali g_x** e **g_y**, con le quali possiamo finalmente calcolare la **magnitudo** e la **direzione** del gradiente:

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

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

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

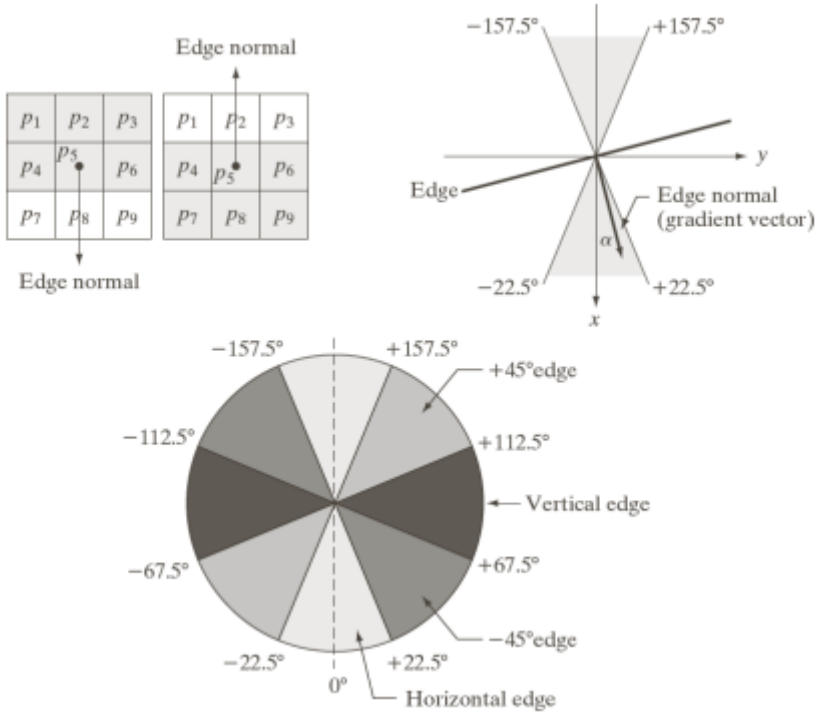
M(x,y) e α(x,y) sono matrici delle stesse dimensioni dell'immagine da cui vengono calcolati.

Dato che M(x,y) è stata generata utilizzando il gradiente, essa contiene molti *picchi* intorno ai massimi locali. Dunque dovremo **sopprimere** tali valori, ad esempio utilizzando il metodo della **Non Maxima Suppression**.

La NMS considera, in un intorno 3x3, i 3 pixel che si trovano in una certa direzione: di questi consideriamo il pixel centrale, e verifichiamo se i suoi vicini nella stessa direzione hanno una magnitudo maggiore della sua: se così fosse, sopprimo il pixel centrale perché significa che non è un edge forte (sopprimo i massimi). Altrimenti, lo conservo.

A tale scopo, è fondamentale **specificare un numero finito di orientamenti da considerare**. Ad esempio, per una regione 3x3 possiamo definire 4 orientamenti per un edge che passa attraverso il punto centrale della regione: *orizzontale*, *verticale*, +45° e -45°.

Inoltre, poiché è necessario **quantizzare** tutte le possibili direzioni in 4 settori, bisogna definire degli opportuni **intervalli**.



Si tenga a mente che ogni edge ha due possibili orientamenti. Ad esempio, un edge la cui normale è orientata a 0° e un edge la cui normale è orientata a 180° sono considerati entrambi edge orizzontali.
Siccome ogni edge ha 2 possibili orientamenti, per ogni orientamento possiamo considerare 2 aree opposte che combaciano nel centro della regione.

In tal caso, distinguiamo 4 orientamenti per un totale di 8 aree nella regione: dunque ogni area avrà un angolo di 360/8= 45°.
Ad esempio, considereremo edge orizzontale qualsiasi edge la cui normale sia orientata fra -22.5° a 22.5°, ma anche quelli la cui normale sia orientata fra -157.5° e 157.5°. (partiamo da 180 ° invece di 0°, sarebbe l'area nel lato opposto).

Individuato il numero di orientamenti, definiamo la **NMS** come segue:

Siano d_1, d_2, d_3, d_4 le quattro direzioni orizzontale, verticale, +45°, -45°.
Lo schema per un intorno 3x3 prevede i seguenti **passi**:

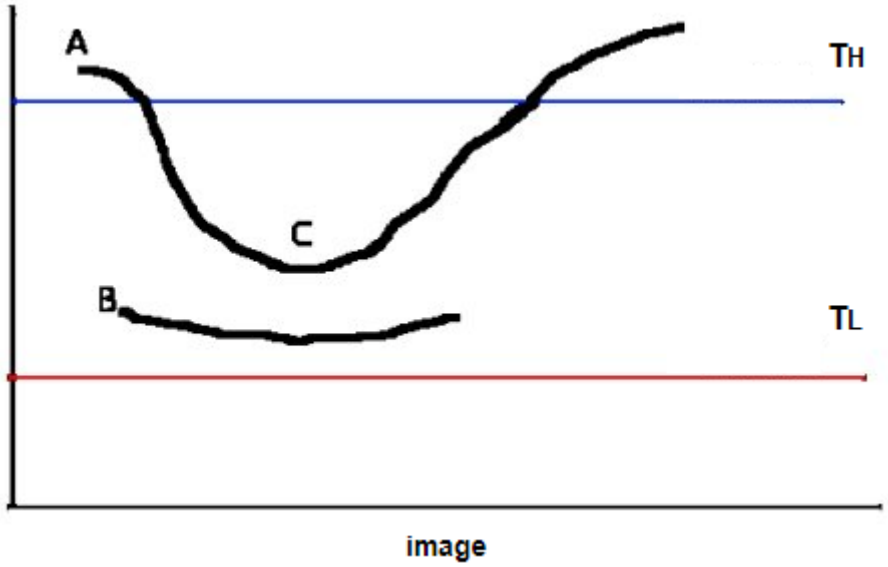
1. trovare la direzione d_k che sia più vicina a $\alpha(x,y)$;
2. se il valore di $M(x,y)$ è minore di almeno uno dei suoi due vicini lungo d_k, poniamo $g_n(x,y) = 0$ (lo sopprimiamo) altrimenti $g_n(x,y) = M(x,y)$.

L'operazione successiva consiste nel **sogliare** l'immagine $g_n(x,y)$ per *ridurre* i *falsi positivi*.

Se utilizziamo una sola soglia: se questa è troppo bassa otteremo dei falsi positivi, invece se è troppo alta otteremo dei falsi negativi. Per questo Canny propose di utilizzare **due soglie**.

Infatti Canny effettua la sogliatura utilizzando l'**isteresi**, che utilizza appunto due soglie: una soglia bassa T_L e una soglia alta T_H.

Questo passo decide quali edge sono "veri" edge e quali no. Qualsiasi edge la cui intensità è superiore a T_H sono sicuramente edge mentre tutti quelli al di sotto di T_L non lo sono (e dunque sono scartati). Fra le due soglie abbiamo una regione di "incertezza" nella quale i pixel saranno o non saranno considerati edge in base alla loro connettività: in breve, se sono connessi a degli edge "sicuri" saranno considerati anche loro edge, altrimenti saranno scartati.



es.

L'edge A si trova al di sopra di T_H, dunque è considerato un "edge sicuro". Nonostante l'edge C sia al di sotto di T_H, è connesso ad A e dunque anche lui sarà considerato un edge valido. B invece, sebbene sia al di sopra di T_L, non è connesso ad alcun edge e dunque sarà scartato.

Quindi questo passo rimuove anche il rumore provocato dai singoli pixel in quanto assume che gli edge siano delle lunghe linee. L'unica cosa ci rimane sono gli *edge forti*.

Possiamo visualizzare l'operazione di **thresholding** creando *due immagini ausiliari*:

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

$g_{NH}(x,y)$ avrà meno pixel non nulli di $g_{NL}(x,y)$, ma tutti i pixel non nulli in $g_{NH}(x,y)$ saranno contenuti in $g_{NL}(x,y)$, per cui rimuoviamo quelli già in $g_{NH}(x,y)$.

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

I pixel di edge in $\mathbf{g_{NH}(x,y)}$ sono pixel di edge "forti".
I pixel di edge in $\mathbf{g_{NL}(x,y)}$ sono pixel di edge "deboli".

Dopo le operazioni di thresholding, gli edge in $\mathbf{g_{NH}(x,y)}$ hanno tipicamente dei "vuoti".
Per riempirli, si utilizza la seguente procedura:

1. localizzare il prossimi pixel di edge *p* in $g_{NH}(x,y)$.
2. etichettare come pixel di edge "forti" tutti i pixel di edge di $g_{NL}(x,y)$ in $N_8(p)$.
3. ripetere il passo 1 finchè tutti i pixel in $g_{NH}(x,y)$ non sono stati visitati.

Dal punto di vista implementativo, non è necessario creare le due immagini ausiliare.
È possibile eseguire il thresholding con isteresi direttamente sull'immagine $g_{NH}(x,y)$.

Se paragoniamo Canny a un algoritmo simile (es. LoG), noteremo come gli edge in Canny non presenteranno vuoti. Di conseguenza, avremo una segmentazione molto pulita.

Riassumendo, l'algoritmo di Canny consiste nei seguenti **passi fondamentali**:

1. convolvere (sottoporre a smoothing) l'immagine di input con un filtro gaussiano;
2. calcolare la magnitudo e la direzione (angolo) del gradiente;
3. applicare la non maxima suppression;
4. appricare il thresholding con isteresi

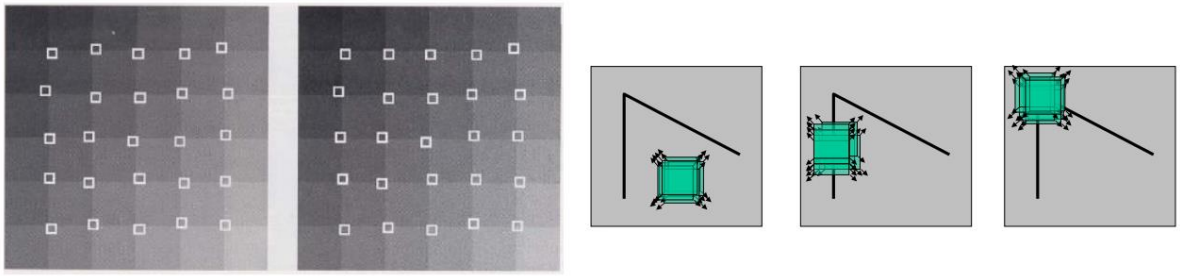
Algoritmo di Harris

L'algoritmo di Harris permette di individuare i **corner** (angoli).
I **corner** sono caratteristiche molto importanti in quanto permettono di tracciare gli oggetti, infatti questi si dimostrano stabili in sequenze di immagini (video).

Un **corner** rappresenta l'**intersezione di linee** all'interno dell'immagine, o **strutture** in pattern di intensità.

Il concetto di Corner è strettamente legato a quello di edge.

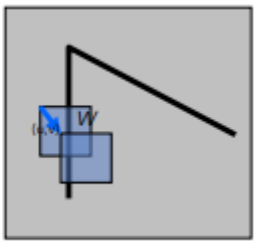
L'**idea** è che:
in una zona di intensità costante, la variazione di intensità rilevata all'interno di una finestra è nulla;
in corrispondenza di un edge, la variazione di intensità si sviluppa in un'unica direzione (quella ortogonale alla direzione del gradiente);
se invece becco un corner, la variazione di intensità si sviluppa in più direzioni.



Dunque vado ad analizzare l'immagine considerando dei blocchi (le finestre), e vado a vedere per ogni finestra che cosa succede se la sposto di una certa quantità (u,v).

Le **variazioni all'interno di una finestra** possono essere calcolate in base a uno spostamento (u,v) attraverso la funzione **E(u,v)**:

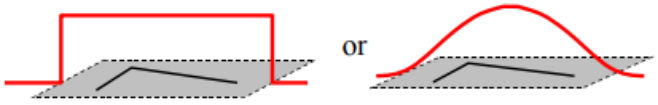
$$E(u,v) = \sum_{x,y} w(x,y)[I(x+u, y+v) - I(x,y)]^2$$



In E(u,v) distinguiamo due "contributi": la parte a sinistra rappresenta la finistra, la parte a destra di occupa di calcolare la *variazione di intensità*.

La finestra è rappresentata da una funzione **w(x,y)**, utilizzata per assegnare un peso ad ogni pixel dell'immagine. Può essere di due tipi:

- composta da 0 e 1, dove assume il valore 1 all'interno della finestra e 0 all'esterno. Ovviamente, dov'è 0 annulla il "secondo contributo" di E(u,v);
- gaussiana, dove al di fuori è sempre 0 ma all'interno i pesi assumono un valore maggiore man mano che mi avvicino ai pixel centrali.



Il secondo contributo è detto **SSD** (Sum of Squared Distance), ovvero vado a calcolare la distanza, in termini di valori di intensità, tra il pixel di posizione (x,y) e il suo valore shiftato della quantità (u,v).
Elevo questa quantità al quadrato in modo da poter esaltare ancora di più le variazioni.

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

Tutte queste variazioni vengono sommate e danno luogo ad un valore che in qualche modo rappresenta la variazione dei valori di intensità all'interno di questa finestra.

Per un piccolo spostamento, la funzione E(u,v) può essere **approssimata dal gradiente**:

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

La matrice E è *simmetrica*, ovvero $E = E^T$, per cui può essere decomposta tramite la tecnica della **SVD** (decomposizione ai valori singolari). Si tratta di una particolare fattorizzazione di una matrice basata sull'uso di autovalori e autovettori.

In questo caso, la matrice E viene decomposta in 3 sottomatrici: U, S, e U^T.
Ci interessano in quanto:

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

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

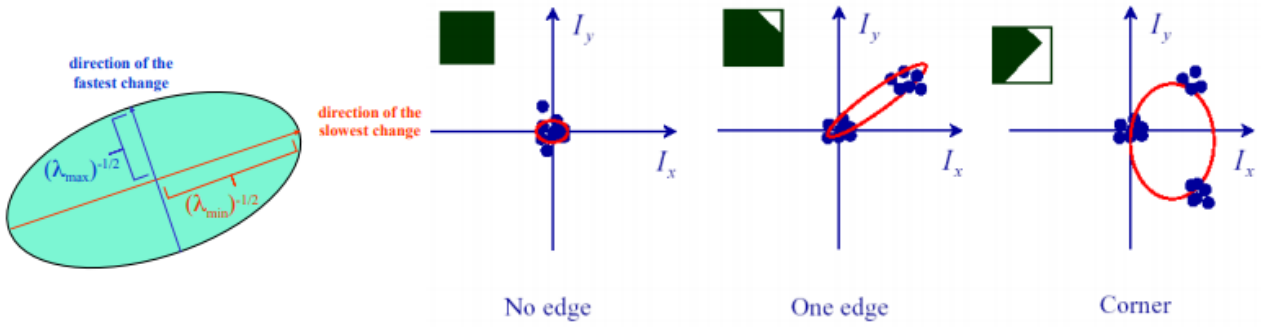
Abbiamo scomposto questa matrice in quanto gli autovalori e gli autovettori posso usarli per descrivere la variazione dei valori all'interno della matrice E(u,v).
In particolare:

- se avrò **$\lambda_1 >> 0$ e $\lambda_2 \approx 0$**, nell'intorno del punto che sto considerando ho una (e una sola) grossa variazione di intensità nella direzione dell'autovettore associato a λ_1. Dunque sono su un edge.
  (l'edge è ortogonale all'autovettore).
- se entrambi gli autovalori **λ_1, λ_2 sono >> 0**, nell'intorno del punto che sto considerando ho una grossa variazione di intensità sia nella direzione dell'autovettore associato a λ_1 che nella direzione dell'autovettore associato a λ_2. Dunque sono su corner.
- se entrambi gli autovalori **λ_1, λ_2 sono ≈ 0**, si tratta di una zona di intensità uniforme.

nb: ">>" molto più grande di… "≈" vicino a…

Dall'interpretazioni geometrica degli autovalori possiamo formare un ellisse, nella quale possiamo osservare come gli autovettori mi indicano come l'ellisse sia orientata rispetto ai dati, e la lunghezza degli assi è data dagli autovalori (ovvero da quanto è forte la variazione).

- Se gli **assi sono molto piccoli**, tutti i valori sono prossimi allo zero. Dunque mi trovo in un'area di intensità costante). Infatti se calcolo il gradiente non troverò forte variazioni ne sull'asse x che sull'asse y.
- Se **ho un'asse molto grande e l'altro molto piccolo**, solo alcuni valori sono prossimi allo zero, mentre gli altri sono caratterizzati da una forte intensità (in una sola direzione): questo perché alcuni si trovano in un'area di intensità costante mentre altri si trovano sull'edge.
- Se **entrambi gli assi sono molti grandi**, significa che mi trovo in un'area in cui avrò delle variazioni molto diverse fra di loro, e dunque alcuni valori sono prossimi allo zero, alcuni sono caratterizzati da una forte intensità in una direzione, ed altri ancora sono caratterizzata da una forte intensità ma in un'altra direzione. Dunque ho trovato un corner.



Il calcolo della SVD andrebbe ripetuto per ogni punto dell'immagine, ma si tratta di un'operazione onerosa, per questo possiamo **ridurre il costo computazionale** utilizzando degli indici più veloci da calcolare ma che sia comunque legata agli autovalori della matrice E.

Per fare ciò, dobbiamo ricordare i concetti di *traccia* e *determinante* di una matrice, i quali sono rispettivamente dati dalla somma e dal prodotto degli autovalori.

Dunque se della matrice E mi calcolo la traccia e il determinante, e combino questi due valori, posso ottenere un **indice R** che mi offre un'approssimazione della SVD in quanto dipende sempre dall'andamento degli autovalori.

Dunque utilizziamo l'indice R al posto della SVD, per ogni pixel (u,v), che è dato dal determinante della matrice calcolata nell'intorno meno il quadrato della traccia moltiplicato per un certo valore *k*.

$$trace\big(C(u,v)\big) = \lambda_1 + \lambda_2$$

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

da cui:

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

ovviamente dovrò andare a sogliare anche questo indice R.
Se R >>0 allora mi troverò in presenza di un corner.

Per calcolare l'indice R…

ricordiamo innanzitutto la matrice E(u,v), le cui componenti C chiamiamo per semplicità C_{00}, C_{11}, C_{01}, C_{10}.

1. **Calcoliamo il determinante**, che ricordiamo essere dato "dal prodotto della diagonale principale meno il prodotto della diagonale secondaria";
2. **Calcoliamo la traccia**, che ricordiamo essere dato "dalla somma delle componenti della diagonale principale";
3. infine, **calcoliamo l'indice R**.

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

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

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

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

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

L'algoritmo di Harris consiste nei seguenti **passi fondamentali**:

1. calcolare le derivate parziali rispetto ad x e y (D_x, D_y), che in realtà sarebbero I_u e I_v.
2. calcolare D_x^2, D_y^2 e $D_x * D_y$, che in realtà sarebbero I_u^2, I_v^2 e $I_u * I_v$.
3. applicare un filtro Gaussiano a D_x^2, D_y^2 e $D_x * D_y$;
   praticamente, è quello che facciamo quando moltiplichiamo quei valori per w(x,y).
   Ciò ci permette anche di avere un effetto di smoothing.
4. dal punto 3 dunque si ottengono le componenti di E(u,v), cioè: C_{00}, C_{11}, C_{01} (e quindi anche C_{10});
5. **calcolare l'indice R**, che possiamo fare in quanto date le componenti possiamo calcolare determinante e traccia;
6. siccome i valori calcolati sono molto elevati, per questioni di manegevolezza di normalizza l'indice R in [0,255];
7. infine sogliare R.

Step

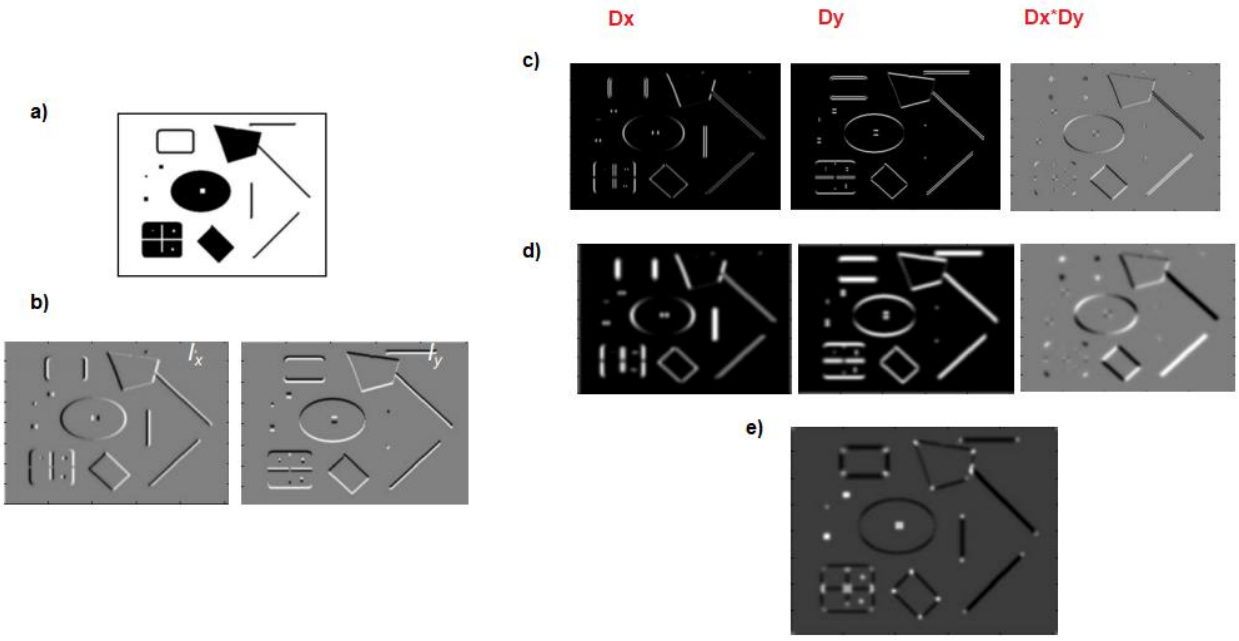
a) immagine originale

b) derivata lungo l'asse X e l'asse Y

c) le riporto nell'intervallo [0,255]. Da sinistra verso destra abbiamo D_x, D_y e D_x*D_y.

d) effettuo lo smoothing col filtro Gaussiano;

e) calcolo gli indici ed individuo i corner (i puntini più bianchi).
Notare che ho una risposta anche all'estremità dei segmenti perché di fatto è l'inizio e la fine dei due bordi, che però non è forte come negli "angoli veri e propri".



LEZ 8 – Trasformata di Hough

La trasformata di Hough è una tecnica che permette di riconoscere particolari **configurazioni di punti** presenti nell'immagine, quali: *rette*, *cerchi* e altre forme (generalizzazione che non vedremo).

In generale questa tecnica si applica se la forma cercata può essere espressa tramite una **funzione parametrica**, ovvero di una funzione che fa uso di un insieme di insieme di parametri. L'insieme dei parametri definsce una particolare **istanza della forma**.

La **caratteristica della trasformata di Hough** è che **sfrutta gli edge** trovati dagli altri algoritmi (es. Canny) per capire se i pixel di edge che sono stati rilevati si dispongono globalmente all'interno dell'immagine lungo la forma che stiamo cercando.

Viene definito un **operatore globale** perché non opera sull'intorno di ogni pixel ma analizza le sue proprietà globali (rispetto agli altri pixel dell'immagine).

Ad esempio, individuare le corsie su un'autostrada. Ovviamente nell'immagine avremo molti punti di edge, ma le corsie sono delle *rette* bianche e possiamo individuarle cercando nello specifico delle rette.
Oppure, individuare delle monete all'interno di un'immagine, in tal caso dovremo indivuduarle cercando dei cerchi (dove ognuna ha un proprio raggio).

Hough per rette

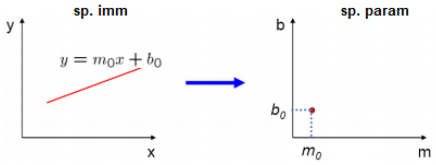
Una **rappresentazione** della retta può essere espressa sia attraverso la **forma canonica: y=mx+b**; oppure attraverso la **forma normale: ρ = x cosθ + y sinθ**.

- **forma canonica**: un'istanza della retta sarà caratterizzata dai parametri (m,b), che rappresentano rispettivamente il coefficiente angolare e l'intercetta all'origine. Fissati questi due parametri, al variare di *x* ricavo la *y* e dunque ottengo tutte le coppie (x,y) che si dispongono su quella particolare retta;
- **forma normale**: un'istanza della retta sarà caratterizzata dai parametri (ρ,θ), che rappresentano rispettivamente la distanza dall'origine e l'angolo formato dal segmento ρ con l'asse delle x.

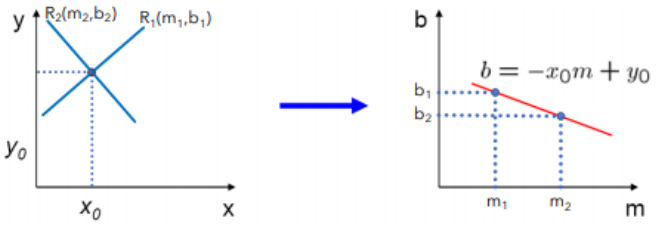
Il nostro scopo è trovare tutti i punti di edge caratterizzati da questi parametri, ovvero i parametri che descrivono la retta dove giacciono questi punti.

• Forma canonica

Fissata la forma di interesse (retta) e la sua *rappresentazione*, è possibile considerare una trasformazione dallo spazio dell'immagine (x,y) allo **spazio dei parametri (m,b)**, ovvero uno spazio dove l'asse delle ascisse è rappresentato dal coefficiente angolare *m* mentre l'asse ordinate è rappresentato dall'intercetta all'origine *b*. In tal caso, i parametri fissati di m,b (m_0, b_0) rappresenteranno un punto.
Dunque, una **retta** nello spazio immagine diventa un **punto** nello spazio dei parametri.

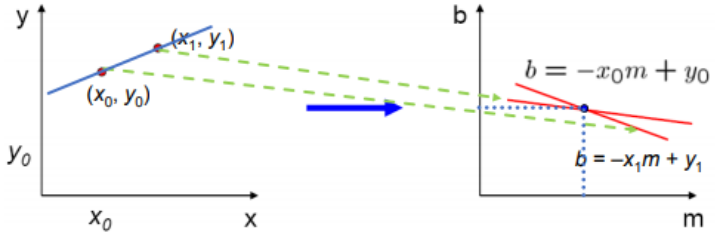


Viceversa, un **punto** nello spazio immagine rappresenta una **retta** nello spazio dei param. Infatti basta osservare che se riscriviamo la forma canonica come: **b=-mx+y**, quello che facciamo è fissare *x* e *y* e al variare di *m* ottengo *b*, quindi otterrò tutte le infinite rette che passano per quel punto. Nello spazio dei parametri troverò dunque una retta, ottenuta tenendo conto delle diverse combinazioni di *b* e di *m*.



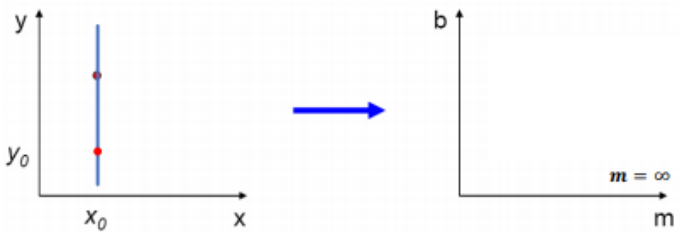
Ovviamente questo è possibile perché sappiamo che per un punto passano infinite rette, e il motivo per cui nello spazio dei parametri riusciamo a identificare un'unica retta è perché sto invertando la funzione andando a fissare x e y, dunque tutti i punti condividono lo stesso coefficiente angolare x_0 e la stessa intercetta y_0 (e di conseguenza, si tratta della stessa retta).

Se si considerano **due punti** nello spazio immagine, sappiamo che per entrambi i punti passano infinite rette ma esiste una sola retta che passa per entrambi i punti, caratterizzata dai parametri m_0 e b_0.
Nello spazio dei parametri avremo dunque due rette il quale **punto di intersezione** ci darà proprio questi due parametri m_0 e b_0.



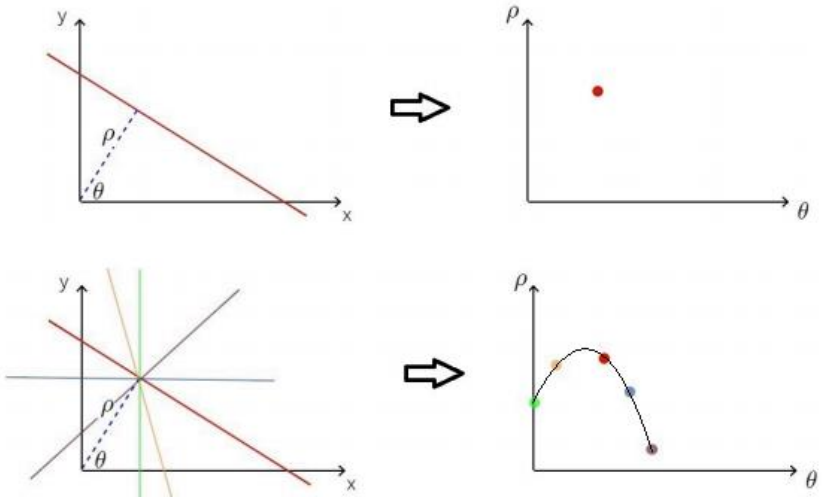
Lo **scopo di tutto ciò** è individuare nello spazio immagine la retta **$y=m_0x+b_0$** partendo da solo due punti (punti di edge), in quanto si tratta della **retta sulla quale giacciono tutti i punti di edge**.

Questa rappresentazione presenta tuttavia un **problema**: quando i punti di edge si dispongono su un'asse verticale, il coefficiente angolare è infinito (m=inf).
In questa situazione, tutti i punti che si trovano su questa retta dovrebbero considerare un *m* infinito, che non è rappresentabile nello spazio dei parametri. Per questo motivo si utilizza la **forma normale**.

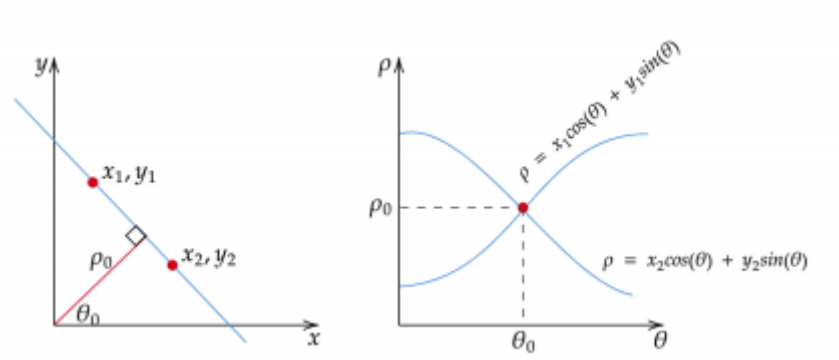


• **Forma normale**

Vale il discorso fatto prima, solo che adesso considereremo la rappresentazione polare: **ρ = x cosθ + y sinθ**. ρ e θ rappresentano i parametri della funzione.
Una **retta** nello spazio immagine viene rappresentata da un punto nello spazio dei parametri (ρ,θ)., dove rappresentano rispettivamente l'asse delle ordinate e l'asse delle ascisse.
Un **punto** nello spazio immagine viene rappresentato da una **sinusoide** (cambia solo la forma, ma l'idea rimane la stessa).



Anche in questo caso, dati due punti nello spazio immagine, possiamo individuare due sinusoidi nello spazio dei parametri. Il punto di intersezione fra questi due sinusoidi avrà coordinate (ρ_0,θ_0).



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

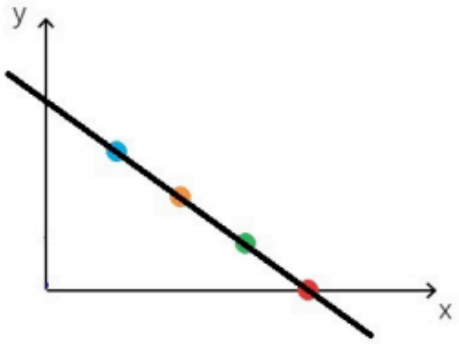
Schema di voto

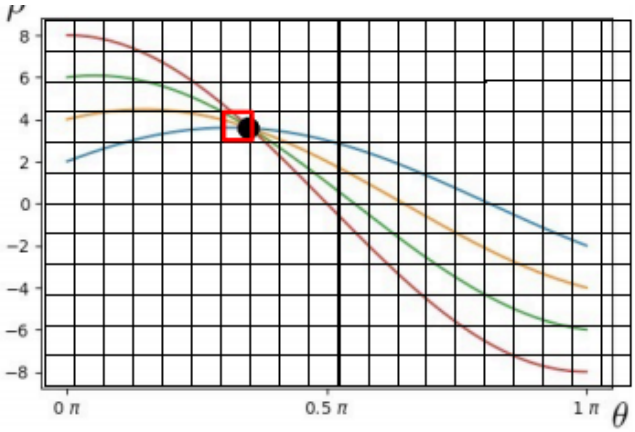
Se ci trovassimo in un mondo ideale, ed avessimo precisamente la posizione dei punti di edge, potremmo semplicemente mettere tutti i punti a sistema per trovare la retta. Tuttavia le immagini sono soggette a rumore e i calcoli potrebbero essere affetti da "errore", quindi non abbiamo precisamente la loro posizione.

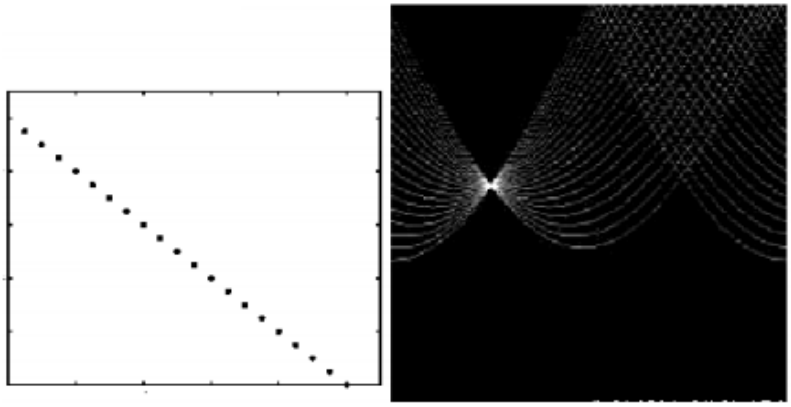
Per trovare i **punti di intersezione** delle rette dobbiamo **quantizzare** lo spazio dei parametri. Così facendo, considereremo solo degli intervalli di valori di ρ e θ.

Lo spazio quantizzato, detto **spazio dei voti**, è caratterizzato da "celle" dette **accumulatori** (si tratta di una matrice).
È detto "spazio dei voti" in quanto ogni punto di edge della sinusoide *vota* le celle che attraversa. Quello che ci aspettiamo è che nel punto di intersezione troviamo il numero maggiore di voti.
Più piccolo sarà l'intervallo [ρ,θ], più precisa sarà l'individuazione della retta.

Alla fine della votazione potremmo ritrovarci con più picchi, per questo motivo possiamo applicare una soglia e considerare solo gli intervalli che hanno ricevuto il maggior numero di voti. Più bassa sarà la **soglia**, maggiore sarà il numero di rette che andremo a considerare.







Hough per cerchi

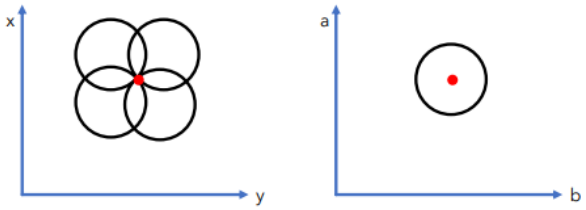
La trasformata di Hough può essere usata anche per individuare i cerchi. Utilizza la stessa idea vista per le rette, dunque anche qui sfrutteremo lo spazio dei parametri.

Una circonferenza con centro (a,b) e raggio R è rappresentato dall'**equazione parametrica**:

$$x = a + \mathrm{R}\cos\theta$$
$$y = b + \mathrm{R}\,sin\,\theta$$

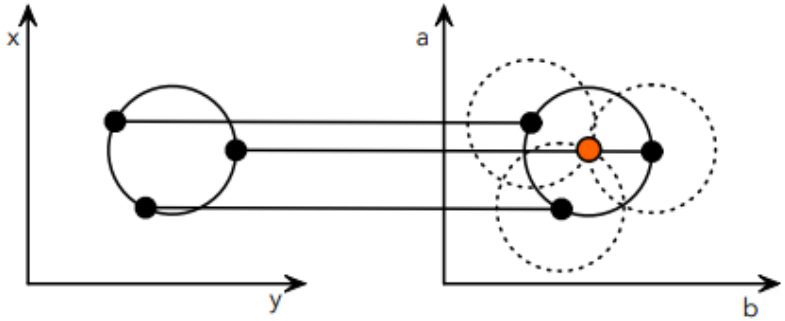
Siccome in un'immagine possiamo avere cerchi con centri e raggi differenti, dovremo quindi tenere in considerazione 3 parametri: *a*, *b* ed *R*.
In questo caso lo **spazio dei parametri** e lo **spazio dei voti** è **tridimensionale**.

Per motivi di ragionamento, supponiamo che R sia fissato. Dunque dovremo preoccuparci di soli 2 parametri: *a* e *b*.
Per un **punto** nello spazio immagine passano infinite circonferenze di raggio R. Questo punto nello spazio immagine corrisponde ad una **circonferenza** nello spazio dei parametri il cui centro sarà dato da (x_0,y_0) e i punti sulla circonferenza corrispondono ai centri delle possibili circonferenze nello spazio immagine.



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

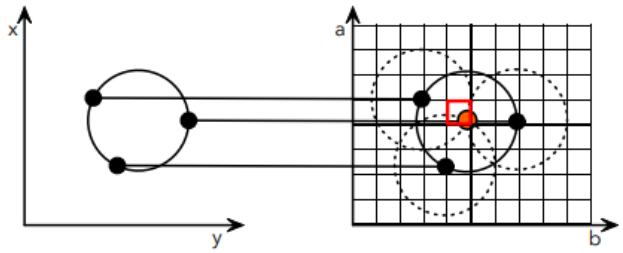
$$a = x - r * \cos\left(\theta * \frac{\pi}{180}\right)$$
$$b = y - r * \sin\left(\theta * \frac{\pi}{180}\right)$$



Una volta che ho la spazio dei parametri effettuo la **quantizzazione dello spazio**, considerando solo degli intervalli di valori *a* e *b*, così da ottenere lo **spazio dei voti**.

Ogni punto di edge della circonferenza *vota* le celle che attraversa. Quello che ci aspettiamo è che nel punto di intersezione troviamo il numero maggiore di voti.
Più piccolo sarà l'intervallo [a,b], più precisa sarà l'individuazione della circonferenza.

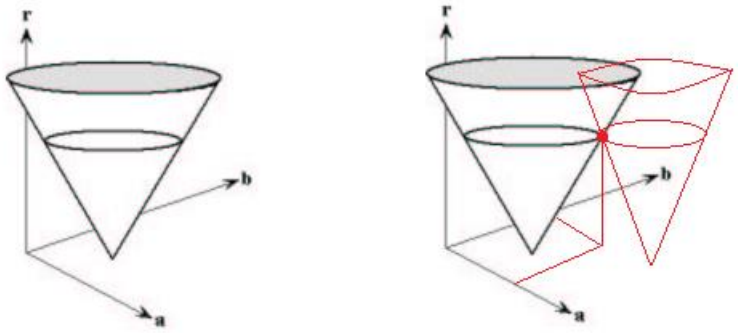
Alla fine della votazione potremmo ritrovarci con più picchi, per questo motivo possiamo applicare una soglia e considerare solo gli intervalli che hanno ricevuto il maggior numero di voti. Più bassa sarà la **soglia**, maggiore sarà il numero di circonferenze che andremo a considerare.



Nel caso il **raggio non sia noto**, tutti i parametri (a,b,R) possono variare per cui lo spazio di voto è **tridimensionale**.

In particolare, ogni **punto** nello spazio immagine corrisponde alla **superficie di cono** di nello spazio dei parametri (cioè,abbiamo una circonferenza per ogni raggio, tutte con lo stesso centro.).
Ogni punto di edge voterà per tutti i possibili raggi delle circonferenze dello stesso centro.
È come se i voti si accumulassero sulla superficie esterna di questo cono.
Un altro punto voterà ovviamente per un altro cono, con un centro diverso.

Le intersezioni delle superfici di questi coni individuano le coordinate del centro (a,b) ed il raggio del centro R.



Caratteristiche della trasformata di Hough

La trasformata di Hough:

- permette di ridurre il numero dei parametri utilizzando l'informazione relativa alla direzione del gradiente;
- è **robusta al rumore** presente nell'immagine e ad eventuali interruzioni (gap) nelle forme cercate;
- può individuare più istanze di una forma in una singola esecuzione (infatti è globale);
- è parallelizzabile.

L'algoritmo di Harris (per rette) consiste nei seguenti **passi fondamentali**:

1. creare l'accumulatore H (bidimensionale);
2. applicare l'algoritmo di Canny per individuare i punti di edge;
3. per ogni punto di edge (x,y):
    a. calcolare ρ = x cosθ + y sinθ, per ogni angolo θ = 0:180 (consideriamo una sinusoide);
    b. incrementare H(ρ,θ). Dunque H(ρ,θ) = H(ρ,θ) +1; (votiamo quella specifica cella).
4. una volta che tutti i punti di edge dell'immagine hanno inserito il loro voto nello spazio di voto, si analizza questo spazio e tutte le celle H(ρ,θ) che hanno un valore superiore ad una certa soglia th corrispondono alle rette nell'immagine.

L'algoritmo di Harris (per cerchi) consiste nei seguenti **passi fondamentali**:

1. creare l'accumulatore H (tridimensionale);
2. applicare l'algoritmo di Canny per individuare i punti di edge;
3. per ogni punto di edge (x,y):
    a. calcolare $a = x - r * \cos\left(\theta * \frac{\pi}{180}\right)$ **e** $b = y - r * \sin\left(\theta * \frac{\pi}{180}\right)$, per ogni angolo θ = 0:360 e per ogni raggio r= R_{min}:R_{max} (considero solo un intervallo di raggi per motivi di efficienza);
    b. incrementare H(a,b,r). Dunque H(a,b,r) = H(a,b,r)+1.
4. una volta che tutti i punti di edge dell'immagine hanno inserito il loro voto nello spazio di voto, si analizza questo spazio e tutte le celle H(a,b,r) che hanno un valore superiore ad una certa soglia th corrispondono alle circonferenze nell'immagine.

nb: seno e coseno vogliono l'angolo in radianti.

In OpenCV, la funzione che implementa Hough (per rette) è:

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

HoughLines vuole in input direttamente gli edge dell'immagine, che deve essere a 8bit e che verrà trattata come un'immagine binaria.
`lines` è un vettore di linee. Ogni linea è rappresentando da un vettore di 2 o 3 elementi (ρ,θ) o (ρ,θ,*voti*). ρ è la distanza dalle coordinate di origine (0,0) (angolo in alto a sx). θ è l'angolo della linea in radianti (0 è circa una retta vert., π/2 è circa una retta orizzont.).
voti è il valore dell'accumulatore.
0 è un punto di edge, diverso da 0 non è un punto di edge.
I parametri ρ,θ identificano la risoluzione dello spazio di voto, definiti in pixel e radianti.
La threshold rappresenta il numero di voti che deve ricevere ogni cella per essere restituita.

In OpenCV, la funzione che implementa Hough (per cerchi) è:

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

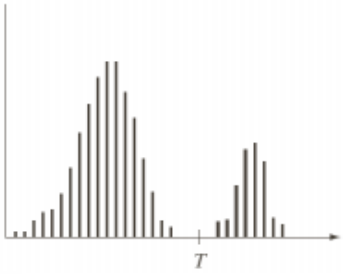
HoughCircles richiama al suo interno l'algoritmo di Canny, quindi basterà solo passargli l'immagine originale.
In output avremo un'immagine di tre canali: rispettivamente a,b ed R.; per un totale di *n* triple a,b,R.
dp sarebbe la risoluzione dell'accumulatore, che può essere più grande o più piccolo in modo da ridurre le dimensioni dello spazio di voto.
minDist è la distanza minima che deve esserci fra due centri per considerarli due cerchi diversi.
param1 è la soglia alta di Canny, mentre param2 è la soglia sull'accumulatore.

LEZ 9 – Sogliatura

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

Le tecniche di sogliatura si basano sull'**analisi** dei valori di intensità e su alcune proprietà per la partizione delle immagini in regioni.
Tipicamente, l'analisi viene effettuata sull'**istogramma dei valori di intensità**.

Si consideri l'istogramma di un'immagine f(x,y) composta di **oggetti chiari su sfondo scuro** (o viceversa), in modo tale da avere **due mode** principali.
Per segmentare l'immagine è possibile scegliere una **soglia T** che separa le due mode:



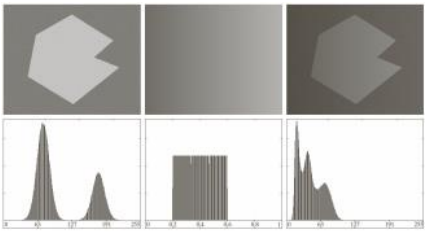
$$g(x,y) = \begin{cases} 1, & \text{if } f(x,y) > T \\ 0, & \text{if } f(x,y) \leq T \end{cases}$$

Possiamo distinguere diversi **tipi di sogliatura**:

- **globale**: la soglia è applicata a tutta l'immagine;
- **variabile**: la soglia viene modificata durante il processo;
- **locale**: la soglia dipende dall'intorno del pixel;
- **dinamica**: la soglia dipende dalla posizione del pixel;
- **multipla**: si utilizzano più soglie.

I **fattori** che influenzano la sogliatura sono:

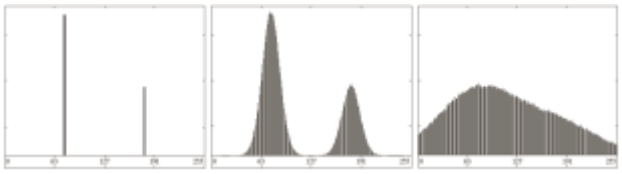
- *distanza tra i picchi*: più questi sono distanti fra di loro, più è facile trovare la soglia ottimale;
- *rumore*: modifica l'istogramma;*
- *dimensione dell'oggetto rispetto allo sfondo*: se ho un oggetto piccolo, il numero di pixel che contribuiscono alla formazione dell'istogramma dell'oggetto sono pochi;
- *uniformità* della fonte di *illuminazione* e delle proprietà di *riflettanza* nell'immagine: se illuminazione e riflettanza non sono uniformi, possono comportare la scomparsa della "valle" che separa i due picchi, non rendendo più possibile segmentare l'immagine.



*Il **rumore** influenza la sogliatura in quanto modifica l'istogramma.
Se ci trovassimo di fronte ad un'immagine perfetta senza rumore in cui abbiamo solo 2 intensità ben distinte, ci troveremmo di fronte ad un istogramma con due mode "a chiodo". In tal caso, qualunque valore di soglia tra le due mode consente di ottenere la segmentazione.

Supponiamo di aggiungere un po' di rumore Gaussiano (10 livelli di intensità). In tal caso le due mode risultano più ampie ma dovrebbero essere ancora facilmente separabili.

Se aggiungiamo molto rumore (50 livelli di intensità) l'istogramma diventerebbe così corrotto da non rendere più distinguibili le due mode, e dunque non sarebbe possibile trovare una soglia che consenta di segmentare l'immagine.

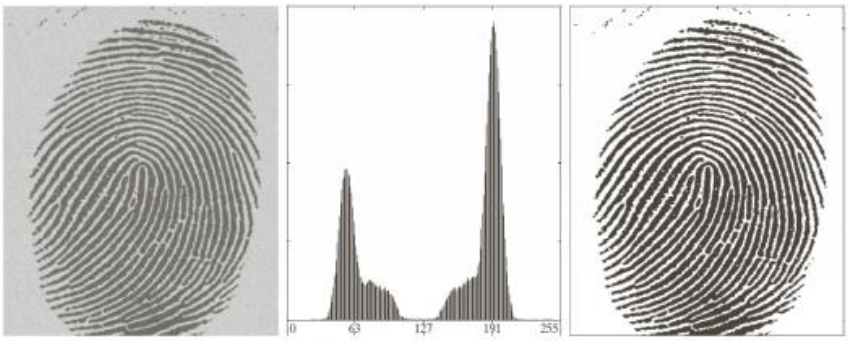


• **Sogliatura globale**

La **sogliatura globale** è applicata a tutta l'immagine, e può essere utilizzata quando le distribuzioni di intensità dei pixel in background e in foreground sono sufficientemente distinte. Infatti tale sogliatura funzione bene solo quando si hanno 2 mode ben separate.

Per trovare il valore della soglia possiamo utilizzare un semplice algoritmo iterativo diviso in 5 passi:

1. stimare un **valore iniziale T**;
2. **segmentare** l'immagine utilizzando T in modo da ottenere due gruppi di pixel: G_1 e G_2 (dove $G_1 > T$, $G_2 \leq T$);
3. **calcolare** l'**intensità media** m_1 e m_2 per rispettivamente G_1 e G_2;
4. **ricalcolare** la soglia $\mathbf{T=(m_1 + m_2)/2}$.
   Di fatto faccio una "media delle medie", andando a riposizionare T in una nuova posizione (più "equilibrata" rispetto a quella di prima).
5. **Ripetere** i passi **2-4** finché le soglie in due iterazioni successive non sono minori di un valore: $\mathbf{|T_i - T_{i+1}| < \Delta T}$ (ovvero la variazione di T è più piccola di un valore da noi stabito).



Metodo di Otsu

Il **metodo di Otsu** è un metodo di sogliatura automatica che opera esclusivamente sull'istogramma. Presuppone che nell'immagine da sogliare siano presenti due sole classi, e quindi calcola la soglia ottima per separarle. Il metodo è detto *ottimale* in quanto massimizza la **varianza interclasse** (separazione tra valori di intensità delle due classi).

In effetti, la sogliatura può essere vista come un problema di statistica dove si vuole ridurre l'*errore medio* dividendo i pixel in due classi. La soluzione a questo problema è conosciuta come *regola di decisione di Bayer*, la quale implementazione tuttavia è complessa e non adatta per situazioni pratiche. Per questo usiamo Otsu.

Supponiamo di avere un'immagine MxN con livelli di intensità compresi in [0, L-1].
Con $\mathbf{n_i}$ denotiamo il numero di pixel dell'immagine con intensità **i**.

L'**istogramma normalizzato** si ottiene dividendo n_i per il numero di pixel MxN. I valori così calcolati possono essere visti come una probabilità (qual è la probabilità che possa comparire l'i-simo valore, ovvero chi è p_i?), infatti dalla somma di tutte le probabilità otteniamo 1.

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

Selezionando una **soglia k** (0 < k < L-1) è possibile segmentare l'immagine ottenendo **due classi C_1** e **C_2** contenenti, rispettivamente, i pixel con intensità minore e maggiore della soglia.

- la **probabilità** che un pixel appartenga a **C_1** è: $\mathbf{P_1(k) = \sum_{i=0}^{k} p_i}$
- la **probabilità** che un pixel appartenga a **C_2** è: $\mathbf{1 - P_1(k)}$

Tuttavia non mi basta calcolare solo la probabilità, in quanto non conosco ancora la **forma della distribuzione** dell'istrogramma. Per introdurre questa informazione, calcolo il **valore di intensità media** nelle due classi **C_1** e **C_2** :

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

nb: pesare la sommatoria con "i" significa usare $P_1(k)$/ $P_2(k)$ come un fattore di normalizzazione.

La **media cumulativa** (utilizzata per visualizzare la somma totale dei dati man mano che crescono nel tempo) fino al livello al **livello k** è:

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

L'**intensità media** dell'intera immagine (es. media gloable) è:

$$m_G = \sum_{i=0}^{L-1} i p_i$$

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

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

dove σ^2_B è la varianza interclasse e σ^2_G è la *varianza globale*, a loro volta definite come:

$$\sigma_B^2 = \mathrm{P_1 P_2 (m_1 - m_2)^2} = \frac{(\mathrm{m_G P_1 - m})^2}{P_1(1-P_1)} \qquad \sigma_G^2 = \sum_{i=0}^{L-1} (i - m_G)^2 p_i$$

Notiamo dalla formula di σ^2_B che al **crescere** della distanza delle **due medie**, la varianza interclasse aumenta, e per il metodo di Otsu è un'informazione importante in quante la varianza interclasse è una misura della separabilità tra le classi.

Dal momento che σ^2_G è una **costante** (fattore di normalizzazione), trovandosi al denominatore significa che anche **η** è una misura della separabilità tra le classi, e massimizzare questa quantità significa massimizzare anche σ^2_B.

Dunque il mio **obiettivo** sarà trovare un **k** che **massimizzi** σ^2_B, ovvero trovare un *k* che mi distanzi quanto più possibile le due mode dalla media globale.
Tale soglia prende il nome di **soglia ottimale k***.

esplicitiamo k nelle formule di prima:

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

k è il k che massimizza* σ^2_B, ovvero è il massimo di tutti i $\sigma^2_B(k)$ in [0,L-1].

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

Ottenuto *k**, è possibile **segmentare** l'immagine.

nb: se il massimo non è unico, si può ottenere k* facendo la media dei valori *k* corrispondenti ai differenti massimi calcolati.

L'algoritmo di Otsu consiste nei seguenti **passi fondamentali**:

1. calcolare l'istogramma normalizzato dell'immagine;
2. calcolare le somme cumulative $P_1(k)$ per *k* in [0,L-1];
3. calcolare le medie cumulative m(k) per *k* in [0,L-1];
4. calcolare l'intensità globale media m_G;
5. calcolare la varianza interclasse $\sigma^2_B(k)$ per *k* in [0,L-1];
6. calcolare la soglia k*, ovvero il massimo di tutti i $\sigma^2_B(k)$ in [0,L-1];
   se il massimo non è unico, si può ottenere k* facendo la media dei valori *k* corrispondenti ai differenti massimi calcolati.
7. (opzionale) calcolare il valore di separabilità η(k*).

nb: il punto 7 fornisce qualche informazione sulla qualità della massimizzazione.

nb formule:

1) $p_i = \frac{n_i}{MN}$,

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

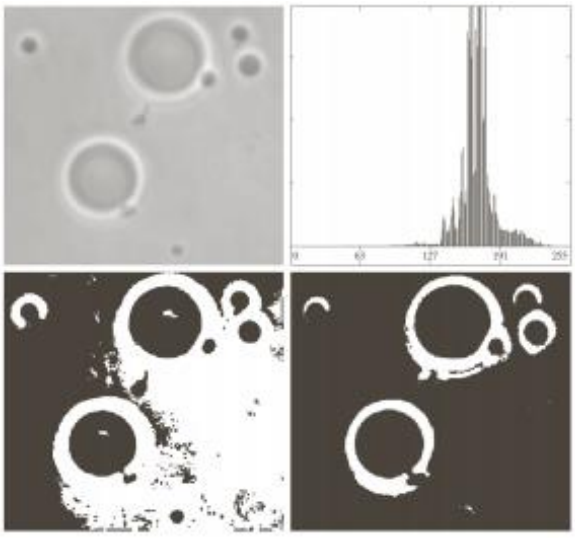
3) $m(k) = \sum_{i=0}^{k} i p_i$

4) $m_G = \sum_{i=0}^{L-1} i p_i$

5) $\sigma_B^2(k) = \frac{[m_G P_1(k) - m(k)]^2}{P_1(k)[1 - P_1(k)]}$

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

7) $\eta(k) = \frac{\sigma_B^2(k)}{\sigma_G^2}$



Esempio

In basso a sx ho dei pixel che vanno in foreground nonostante siano palesemente di background (ciò è dovuto non solo al rumore, ma anche perché l'istogramma è molto stretto). Otsu ha comunque un errore, ma lo MINIMIZZA in quanto massimizza la variazione (l'errore può essere anche pixel di back che vengono valutati come fore).

Smoothing per migliorare la sogliatura (globale)

Il rumore complica il problema della sogliatura, per questo si può applicare dello smoothing così da migliorare le prestazioni dell'algoritmo.

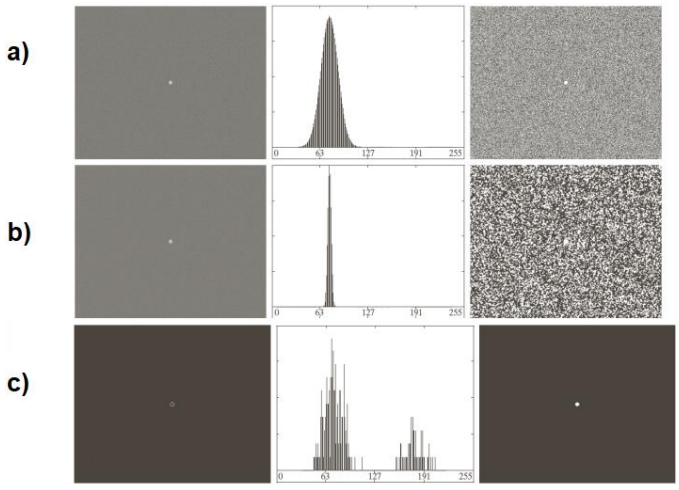
Tuttavia, nel caso in cui la regione da segmentare è così piccola da rendere il suo contributo all'istogramma trascurabile (rispetto a quello del rumore), lo smoothing non risolve il problema.
Possiamo però migliorare l'istogramma considerando solo i pixel di edge tra gli oggetti e il background. In questo modo la probabilità che un pixel appartenga al foreground è pressoché uguale alla probabilità che appartenga al background, per cui le mode saranno simili.
Per l'estrazione dei punti di edge si può utilizzare il Laplaciano o la magnitudo del gradiente.

Un **algoritmo** che sfrutta questa idea è costituito dai seguenti passi:

1. calcolare un'immagine di edge;
2. individuare un valore di soglia T;
3. applicare la soglia all'immagine di edge, ottenendo un'immagine binaria g_r ;
4. calcolare l'istogramma utilizzando solo i pixel dell'immagine di input che corrispondo alle posizioni dei pixel con valore 1 in g_r ;
5. utilizzare il metodo di Otsu con l'istogramma ottenuto.

Per la scelta di T, di solito di sceglie un percentile alto (90esimo percentile), ovvero corrisponde al più piccolo valore di quelli maggiori al 90% dei valori dell'insieme, che in altre parole significa che prendo la maggior parte dei pixel di edge trovati.

nb: il nostro obiettivo non è trovare gli edge, ma individuare le classi (infatti gli algoritmi che individuano gli edge non individuano anche le classi).



a) immagine originale (oggetto piccolo) a cui abbiamo applicato Otsu;

b) immagine con smoothing a cui abbiamo applicato Otsu;

c) immagine a cui abbiamo applicato l'algoritmo di cui sopra.

• **Soglie multiple**

Il metodo di Otsu può anche essere generalizzato al caso di K soglie, tuttavia se si utilizzano troppe soglie il metodo perde di significato.
In linea generale, per Otsu non sia usa mai un k>2, ovvero si usa Otsu per separare al più 3 classi.

Se consideriamo **k=2**, ovvero 3 classi, per l'algoritmo di Otsu:

- la **varianza interclasse** è data da:

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

- le **soglie ottimali** sono:

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

- il **grado di separabilità** è dato da:

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

dove le *somme cumulative* e le *medie cumulative* per ogni classe sono date da:

$$P_1 = \sum_{i=0}^{k_1} p_i \qquad m_1 = \frac{1}{P_1} \sum_{i=0}^{k_1} i p_i$$

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

$$P_3 = \sum_{i=k_2+1}^{L-1} p_i \qquad m_3 = \frac{1}{P_3} \sum_{i=k_2+1}^{L-1} i p_i$$

e ovviamente valgono le stesse relazioni viste prime:

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

$$P_1 + P_2 + P_3 = 1$$

Facciamo delle **osservazioni**:

La procedura inizia selezionando la **prima soglia k_1 = 1** (si sceglie 1 perché non ha senso cercare una soglia con intensità pari a 0), e poi si incrementa la soglia k_2 a partire dai valori maggiori di k_1 e minori di L-1 (es. k_2 va da 2 a L-2).
Successivamente si incrementa k_1 e si incrementa la soglia k_2 a partire dai valori maggiori di k_1.

Al termine della procedura si ottiene una **matrice $\sigma^2_B(k_1,k_2)$** all'interno della quale si trova il valore **massimo** in corrispondenza di **k^*_1 e k^*_2**.

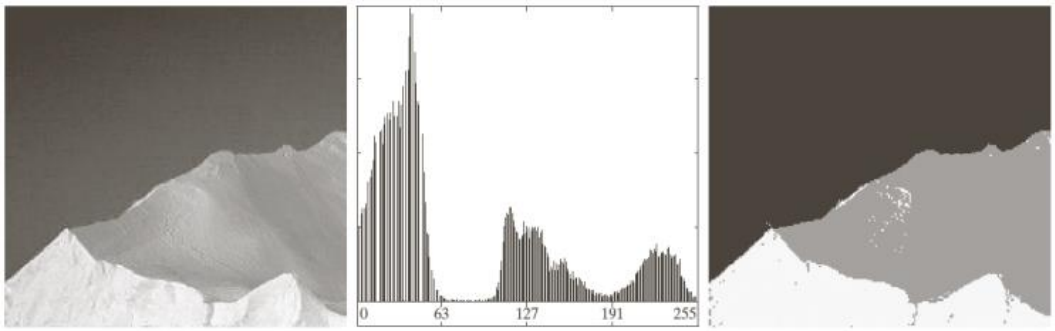
Anche in questo caso, se sono presenti più massimi si fa una media.

L'immagine **segmentata** si ottiene da:

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

dove *a*, *b* e *c* sono classi.

Esempio:

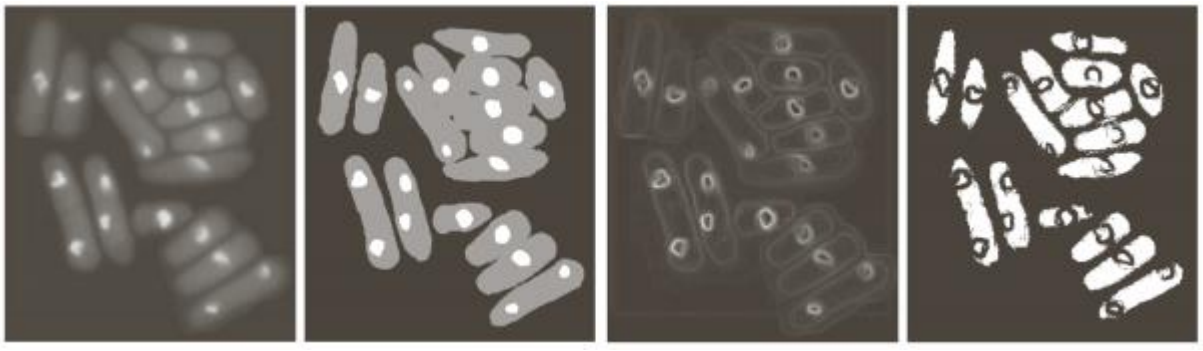


• **Soglie variabili**

Nella sogliatura variabile la soglia viene modificata durante il processo, e ciò risulta particolarmente utile in situazione in cui si presenta un'illuminazione non uniforme.

Uno dei metodi più semplici di sogliatura variabile consiste nel **suddividere** l'immagine in **rettangoli non sovrapposti**, ma si può ottenere una soluzione più efficace utilizzando degli approcci più generali.

Ad esempio, un altro approccio consiste nel calcolare una soglia considerando una o più **proprietà** dell'**intorno di ogni pixel**, ad esempio considerando la **media** e la **deviazione standard** dei valori di **intensità** nell'intorno di ogni pixel.
In alternativa, è possibile usare un **predicato** sui parametri locali.



nb: se esce Otsu all'esame verrà chiesto sempre per k=1 o k=2.

RICORDIAMO DA LEZ3 ->

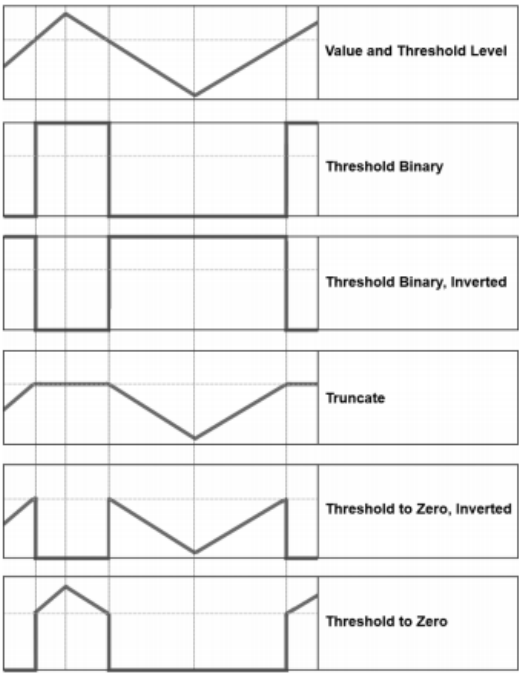
In OpenCV, esistono diverse **funzioni per effettuare la sogliatura** (thresholding):

```
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
);
```

Possiamo distinguere diverse **tipologie** di thresholding. La più importante è quella binaria.

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	$DST_i = (SRC_i > thresh)\ ?\ 0 : SRC_i$



Il thresholding adattivo considera anche l'intorno. L'adaptiveMethod può essere:

- cv::ADAPTIVE_THRESH_MEAN_C
- cv::ADAPTIVE_THRESH_GAUSSIAN_C

blocksize è la dimensione dell'intorno in cui viene calcolata la media pesata.
C è una costante che viene sottratta alla media calcolata.

nb: inoltre consideriamo un altro Threshold type: cv::THRESH_OTSU (metodo di Otsu).

LEZ 10 – Region Growing & Split and Merge

Ricordiamo dalla LEZ6:

[Denotiamo con **R** la regione occupata dall'immagine. La **segmentazione** dell'immagine consiste nel partizionare R in *n* sottoregioni R_1, R_2, …, R_n tali che:

1. la loro unione mi dia la regione completa;
2. R_i è un insieme connesso;
3. le regioni sono disgiunte. Di conseguenza, ogni pixel appartiene ad una sola regione;
4. tutti i pixel che appartengono ad una regione devono rispettare il predicato logico Q;
5. applicato il predicato Q a due regioni, il risultato deve essere falso.

In breve, la **segmentazione** è il processo attraverso cui un'immagine viene suddivisa in più regioni disgiunte. La qualità della segmentazione è fondamentale in quanto può determinare l'esito delle elaborazioni successive. **]**

Region Growing

Il **Region Growing** è una tecnica che raggruppa i pixel o le sottoregioni in regioni via via più grandi in base a dei criteri predefiniti.

Il procedimento inizia dai dei punti iniziali, detti **seed**, e si *propaga* ai pixel adiacenti che rispettano delle proprietà predefinite, i quali vengono aggiunti alla regione.
Quando la regione non può più essere espansa, si ricontrollano i pixel partendo dal seed e si controlla che non facciano già parte di una regione: in tal caso, quel pixel viene considerato un seed a sua volta. Ciò ci assicura la **segmentazione completa.**
Potenzialmente tutti i pixel possono essere seed.

L'**intensità** può fornire da *predicato*, tuttavia bisogna prestare attenzione anche alla **connettività**: infatti raggruppare pixel "simili" ma non adiacenti può generare segmentazioni inconsistenti.

Il **predicato** può essere **calcolato in due modi**:

- può essere calcolato rispetto al seed, cioè aggiungo alla regione tutti i pixel che hanno un livello di intensità simile al seed;
- può essere calcolato rispetto al fronte, cioè aggiungo alla regione tutti pixel che hanno un livello di intensità simile al pixel della regione da cui siamo arrivati.

Il secondo punto può risultare in alcuni casi più precisa, tuttavia se ho lievi variazioni di intensità nell'immagine rischio di creare poche regioni se non addirittura un'unica regione

Tuttavia considerare solo i descrittori locali può non essere sufficiente per definire una *regola di arresto*, per questo si potrebbe tener conto della "**storia**" del processo di accrescimento (es. calcolare la media dei pixel).

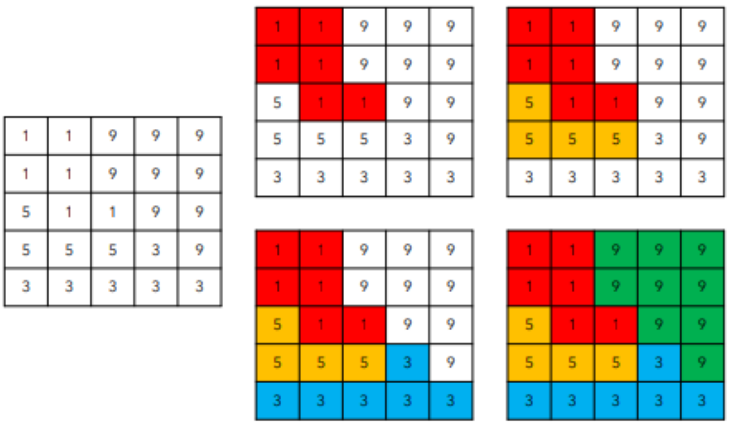
Dunque:

Sia **f(x,y)** l'immagine di input,
Sia **S(x,y)** la matrice dei seed: assegna 1 alle posizioni dei seed, 0 per altro;
Sia **Q** un predicato da applicare ad ogni pixel.

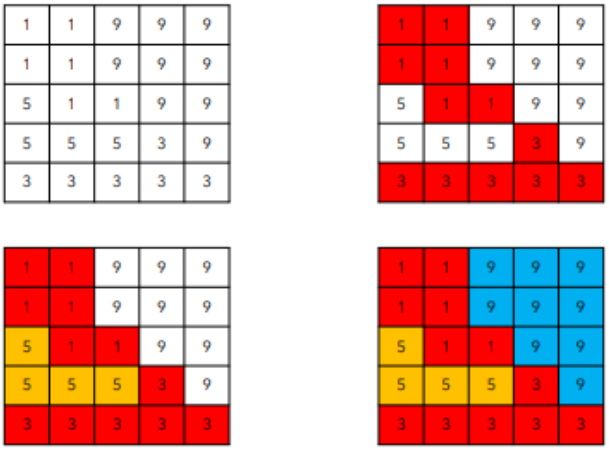
L'algoritmo di Region Growing consiste nei seguenti **passi** fondamentali:

1. formare l'immagine f_Q che nel punto (x,y) contiene il valore 1 se Q(f(x,y)) è vero, altrimento 0;
2. aggiungere ad ogni seed i pixel impostati ad 1 in f_Q che risultano [4 o 8]-connessi al seed stesso;
3. marcare ogni componente connessa con un'etichetta diversa (es. 1,2,3…).

Esempio
Il livello di intensità del pixel deve essere uguale a quello del seed. Consideriamo la 8-connettività dunque ci spostiamo anche in diagonale.



Esempio:
Il livello di intensità del pixel deve essere al massimo più grande di due unità rispetto a quello del seed (Q<=2). Infatti 3-1 = 2.



Split and Merge

Lo **Split and Merge** è una tecnica che consiste nel **dividere** l'immagine in regioni disgiunte di forma e dimensioni arbitrare e successivamente **fonderle** in base a dei criteri di similarità.

Sia **R** la regione corrispondente ad un'intera immagine e **Q** un predicato, è possibile **dividere** R in regione sempre più piccole finché il predicato non risulta **vero** (o comunque se la regione è considerata troppo piccola per essere divisa ulteriormente).
Una strategia molto utilizzata consiste nel dividere le regioni in **quadranti** (mediante quadtrees, ovvero alberi di quadranti).
Questa fase prende il nome di **fase di splitting**.

Al termine della fase di splitting, la partizione finale potrebbe contenere regioni adiacenti con caratteristichie **simili**, per questo motivo le regioni possono essere **fuse**.
Questa fase prende il nome di **fase di merging**.

Notare che la **fase di merging** può essere di **due tipi**:

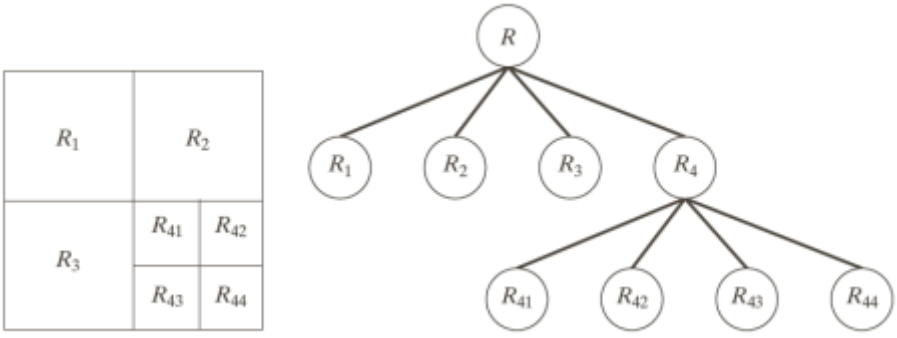
- bottom-up: parto dalle regioni più piccole e formo man mano le regioni più grandi. È più costoso, ma mi permette di avere una segmentazione più fine;
- top-down: parto dalle regioni più grandi.

L'algoritmo di Split and Merge consiste nei seguenti **passi** fondamentali:

1. **dividere** in quattro quadranti tutte le regioni per cui il predicato Q risulta falso;
2. dopo la fase di split, si applica il processo di merging a tutte le regioni adiacenti R_i e R_j, per cui $Q(R_i \cup R_j)$ = Vero;
3. il processo termina quando non è più possibile effettuare unioni.

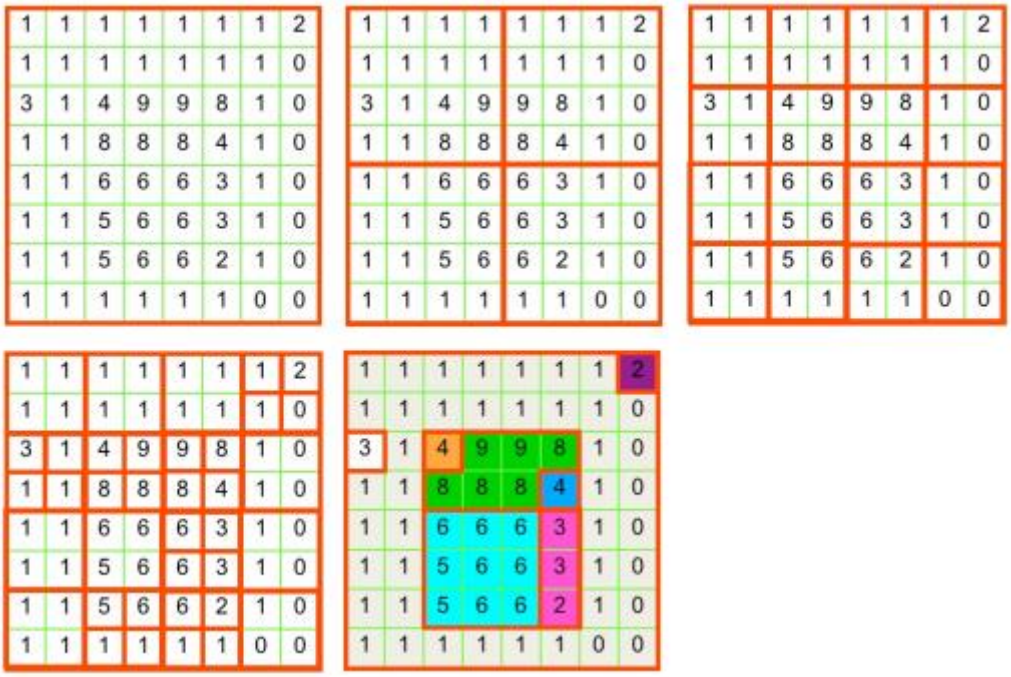
Solitamente si definisce una **dimensione minima** della regione oltre la quale non si effettua lo split

Per ragioni di efficienza, la fase di merge si può eseguire se il predicato è vero per le singole regioni adiacenti (non si effettua l'unione).



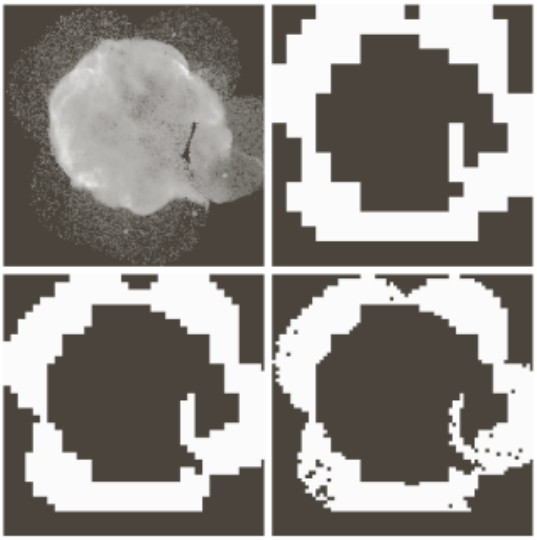
Esempio

Arrivati al singolo pixel (penultima immagine) comincia la fase di merge (ultima immagine).



Esempio

Vogliamo estrarre la corona dall'immagine.
Notiamo come con blocchi più grandi ho un'approssimazione più grossolana, tuttavia con blocchi più piccoli otteniamo sì un'approssimazione più fine ma anche degli errori (cioè i pixel neri dentro la corona).



a	b
c	d

$Q := \sigma > \alpha$ AND $0 < m < \beta$

- (b) 32×32
- (c) 16×16
- (d) 8×8

LEZ 11 – Clustering

Il **clustering** è una tecnica che consiste nel **raggruppare i pixel** in base alle caratteristiche di **intensità**, **colore** e **posizione**.

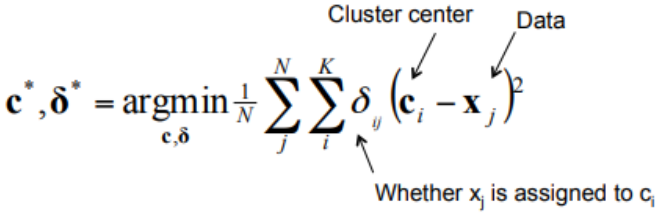
K-means

Nell'algoritmo di **k-means** si creano **k gruppi** (cluster) **disgiunti**, ognuno dei quali è rappresentato da un **centro** (o **media** di valori di intensità dei pixel).

L'**idea** è semplice: all'inizio scelgo come rappresentati *k* punti a caso ed assocerò ad ogni singolo punto il gruppo più vicino. Al passo successivo calcolo il centroide di questo cluster che fungerà da nuovo rappresentate.
Questa procedura iterativa si ferma quando il nuovo rappresentate "è molto vicino al vecchio rappresentante", ovvero la variazione è minima.

La scelta migliore per i centri è quella che minimizza la **SSD** (sum of squared distances) tra tutti i punti ed il centro più vicino. Dunque il nostro **obiettivo** è minimizzare la SSD, ovvero minimizzare la varianza in ogni cluster.

Possiamo descrivere il funzionamento del k-means attraverso una **funzione**:



cluster

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

Voglio trovare dei parametri *c* e δ che minimizzino la funzione, dove *c* rappresenta il centro del cluster mentre δ rappresenta una matrice dove sulle righe ho i pixel *x* e sulle colonne i cluster. x_j rappresenta un generico pixel.

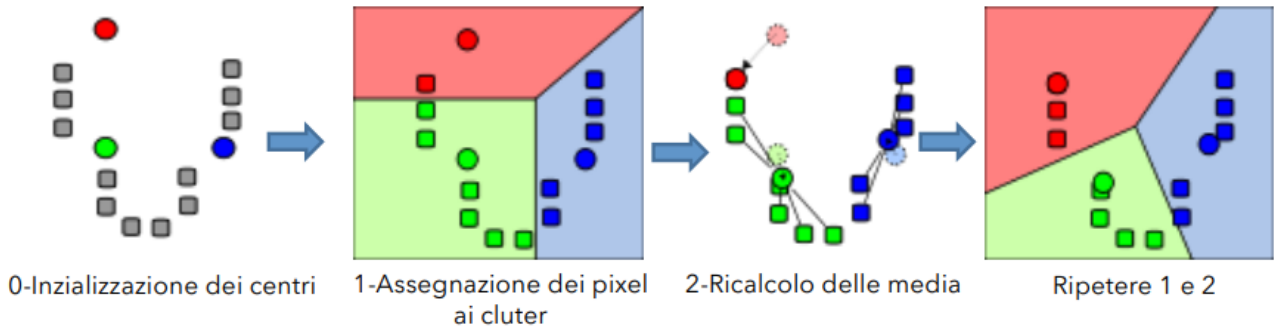
Il valore di δ nella matrice può essere 1 o 0. Nello specifico, per ogni riga potrò avere solo un 1 in quanto un pixel può appartenere ad un solo cluster.
Se il valore è 0, siccome tutto è moltiplicato per δ, quel "contributo" non contribuisce alla sommatoria.

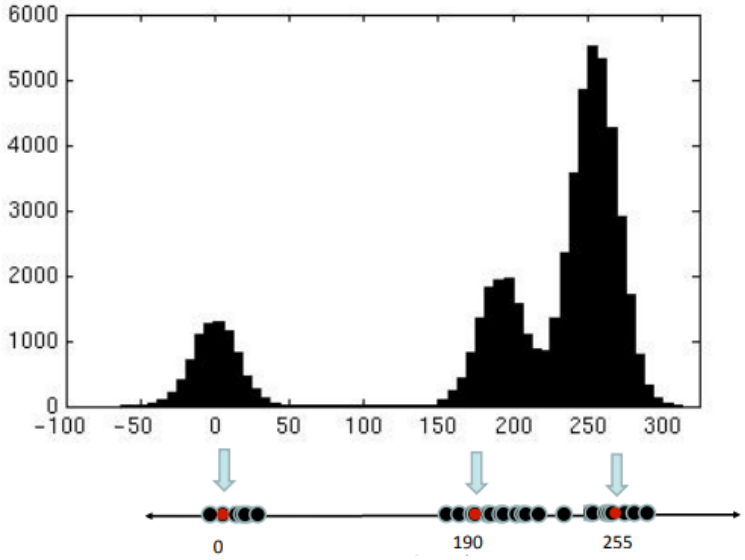
Il motivo per cui all'inizio i *k* centri sono scelti a caso, è perché per calcolare i centrodi ho bisogno dei gruppi, ma all'inizio non avrò nessun gruppo.
Individuati i rappresentati (dunque ho *c*) posso calcolare la matrice δ, e da questi posso calcolare i nuovi gruppi.
Quando le medie non si spostano più, avrò trovato UN minimo della funzione: infatti, il k-means non è in grado di assicurare il minimo globale, ma anzi molto spesso cade all'interno di un minimo locale.

Per questo motivo, si può rilanciare il k-means più volte (in quanto ad ogni nuova esecuzione avrò dei centroidi diversi). Si possono anche tentare più *k*, in quanto non so quanti gruppi potrei avere (almeno se non vedo l'immagine).
Il nostro **obiettivo** è trovare dei gruppi quanto più **compatti** possibili.

L'**algoritmo k-means** consiste nei seguenti **passi** fondamentali:

1. **inizializzare** i centri di ogni cluster;
2. **assegnare** ogni pixel al cluster con il centro più vicino.
   Per ogni pixel $\mathbf{p_j}$ calcolare la distanza (euclidea) dai *k* centri $\mathbf{c_i}$, ed assegnare p_j al cluster con il centro c_i più vicino;
3. **aggiornare** i centri. Dunque calcolare la media dei pixel in ogni cluster.
4. **Ripetere i punti 2 e 3** finché il centro (media) di ogni cluster non viene più modificata (o comunque è minima).





nb: i cluster non devono avere (per forza) componenti connesse, infatti si possono avere "macchie" sparse per l'immagine della stessa intensità.

Il k-means ha un **problema**: se i *k* centri iniziali sono **troppo vicini** il risultato finale potrebbe non essere ottimale. Per questo si usa un'alternativa del k-means, il **k-means++**.

Nel **k-means++** i centri iniziali sono scelti in modo che siano **distanti** tra loro.
Nello specifico:

- Si sceglie il primo centro $\mathbf{c_1}$ in maniera casuale;
- Si calcola la distanza di ogni punto da c_1;
- Si sceglie il prossimo centro c_2 in modo che sia il più lontano possibile da c_1.

Se dovessi scegliere un ulteriore centro c_3, questo dovrebbe essere il più lontano possibile sia da c_1 che da c_2. E così via per tutti gli altri centri.
Il tutto ha un costo k*N.

Lo stesso procedimento si può considerare anche se si considerano immagini a colori. In questo caso ogni pixel e i centri di ogni cluster saranno un vettore di 3 componenti.
Se si usa il colore si ottiene una clusterizzazione migliore, in quanto il colore è un descrittore forte.

Un **ulteriore miglioramento** consiste nel considerare anche la **componente spaziale**.
In questo caso sto chiedendo di trovarei dei gruppi di pixel con valori di intensità simili e che siano spazialmente vicini fra di loro.
Attraverso questa tecnica riusciamo ad ottenere delle regioni un po' più compatte.

La componente spaziale è rappresentata ovviamente da due componenti: *x* e *y*.
Dunque per la scala di grigi avrò un vettore di 3 componenti [x,y,i], mentre per le immagini a colori avrà un vettore di 5 componenti [x,y,B,G,R].

Alcune considerazioni

- **k** deve essere **scelto dall'utente** prima dell'esecuzione dell'algoritmo. In linea generale, se voglio una segmentazione più fine aumento *k*, però dipende molto da quanti oggetti ho nell'immagine e dalla loro intensità/colore.
- in generale, per **valutare la qualità** dei cluster vogliamo valutarne la compatezza. Più questi sono compatti, ovvero meno i valori si disperdono vicino alla media, più il clustering può essere considerato buono.
- L'algoritmo ha complessità **O(kNd)**, dove *k* è il numero di cluster, N il numero di pixel, e *d* il numero di feature (1 in scala di grigi, 3 a colori, oppure 3 o 5 se consideriamo anche le componenti spaziali).
  Dunque fissati *k* e *d*, la complessità è grossomodo lineare.

Mean-shift

Il **mean-shift** è un algoritmo di segmentazione più avanzato, ma con un'idea simile al k-means. Però, in questo caso il parametro *k* non deve essere deciso a priori: il numero ottimale di cluster è individuato a partire dai valori di intensità presenti nell'immagine.

Il mean-shift consiste nel cercare, per ogni valore, la moda più vicina nello spazio delle feature. Per ogni valore si considera una finestra di ampiezza W, si calcola la media di questa finestra, e la si sposta sulla media calcolata. Si ripete il procedimento finchè la variazione della media è nulla (o comunque minima).

L'ampiezza della finestra è fondamentale: più è piccola, più considero valori fra di loro simili.

L'algoritmo ha complessità **$O(kN^2d)$**, dunque è più pesante rispetto al k-means ma offre anche risultati migliori.

In **OpenCV**, il k-means e il mean-shift si implementano come segue:

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

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

nb:

- K-means non vuole l'immagine in input, ma solo i "dati" (un vettore "stacked"). Possiamo convertire l'immagine in un vettore attraverso l'istruzione:

  ```
  data = src.reshape(1,src.total())
  ```

  dove data deve essere di tipo CV_32F.

- Pyr = piramidale. È una variante del mean-shift.