ELIM – LAB

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

INDICE

FERONE LEZ 1-2

OpenCV_1_exercise (OpenCV_1_LoadViewImg.cpp)

```
#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).

OpenCV_2_exercise (OpenCV_Padding.c++)

Realizzare una funzione che effettui il padding di un'immagine.
Sostituire al valore di ogni pixel il valore medio dei livelli di grigio in un intorno 3x3.

A COLORI:

Distinguiamo due funzioni:

- `Mat mean3x3(Mat)`: restituisce l'immagine sulla quale abbiamo effettuato il calcolo del valore medio di TUTTI i pixel in un intorno 3x3;
- `Vec3b submean3x3(Mat,int,int)`: restituisce il pixel sul quale abbiamo calcolato il valore medio in un intorno 3x3. È usata in mean3x3(), e infatti dovremo passare l'indice della riga e della colonna di partenza.

In `mean3x3` distinguiamo 3 sezioni:

1. creazione dell'immagine con padding.
   Aggiungeremo 1 vector di pixel neri ad ogni lato;
2. definiamo la matrice `result`. Ovviamente avrà lo stesso numero di righe e colonne, e lo stesso tipo dell'immagine che abbiamo passato a funzione;
3. Effettuiamo il calcolo. Usiamo un doppio ciclo for perché vogliamo iterare su ogni elemento della matrice.
   Essendo che vogliamo fare questa operazione su un'immagine a colori, ogni pixel sarà composto da 3 valori (BGR), per questo usiamo Vec3b (Vector 3 byte).
   Il valore del pixel nell'intorno 3x3 sarà calcolato attraverso la funzione submean3x3.

In `submean3x3`, inizializziamo a 0 (nero) la somma `sum`, di tipo Vec3i. Il motivo per cui usiamo un vector di interi (e non byte) e perché i byte hanno valore massimo 255, dunque andremmo subito in overflow.
Usiamo un doppio ciclo for dove sommiamo tutti gli elementi di una sottomatrice 3x3 che parte da (istart,jstart).
Infine ritorniamo `sum`, effettuando un cast a Vec3b (perché è il tipo che vogliamo ritornare).

nb: il padding ci serve perché senza di esso faremmo un accesso illegare alla memoria quando andiamo a calcolare degli intorni: si pensi al primo pixel (0,0), se vogliamo calcolare un intorno 3x3 ci accorgeremo che sopra e a sinistra non abbiamo pixel, e ciò comporterà un errore; tuttavia se aggiungiamo un padding (pixel fantoccio) a quei lati, potremo calcolare l'intorno.
Se l'intorno è 3x3 ne basta 1, se 5x5 allora 2, se 7x7 allora 3, ecc.

Il motivo per cui istart e jstart partono da (-1) è perché vogliamo considerare il "centro del quadrato 3x3", e dunque se (i,j) rappresentano il centro, allora (i-1, j-1) rappresentano l'angolo in alto a sx.

NB: Si tratta di una versione vecchia e più caotica, valida solo per mask 3x3.
Vedere la correlazione e la convoluzione in LEZ3 (Correlation e Convolution).

```
Mat mean3x3(Mat img){

    //1.
    Mat imgPadded;
    Scalar value(0,0,0);    //black padding
    copyMakeBorder(img,imgPadded,1,1,1,1,BORDER_CONSTANT,value);

    //2.
    Mat result(img.rows,img.cols,img.type()); //CV_8UC3

    //3.
    int i,j;
    for(i=1; i<imgPadded.rows-1; i++)   //2 -> num pixel added left/right
        for(j=1; j<imgPadded.cols-1; j++)   //2 -> num pixel added top/bottom
            result.at<Vec3b>(i-1,j-1) = submean3x3(imgPadded,i,j);

    return result;
}

Vec3b submean3x3(Mat img, int istart, int jstart){
    Vec3i sum(0,0,0);
    for(int i=istart-1; i<istart+2; i++)
        for(int j=jstart-1; j<jstart+2; j++)
            sum += img.at<Vec3b>(i,j);

    return Vec3b(sum/9);
}
```

IN SCALA DI GRIGI:

Ci sono alcune differenze: innanzitutto, non abbiamo più 3 canali per pixel, ma uno solo. Dunque i pixel non li rappresenteremo più come Vec3b, ma come `unsigned char`.

`sum` può essere più banalmente rappresentato come un `int`.

```
Mat mean3x3_gs(Mat img){

    //1.
    Mat imgPadded;
    Scalar value(0);    //black padding
    copyMakeBorder(img,imgPadded,1,1,1,1,BORDER_CONSTANT,value);

    //2.
    Mat result(img.rows,img.cols,img.type()); //CV_8UC3

    //3.
    int i,j;
    for(i=1; i<imgPadded.rows-1; i++)   //2 -> num pixel added left/right
        for(j=1; j<imgPadded.cols-1; j++)   //2 -> num pixel added top/bottom
            result.at<unsigned char>(i-1,j-1) = submean3x3_gs(imgPadded,i,j);

    return result;
}

unsigned char submean3x3_gs(Mat img, int istart, int jstart){
    int sum = 0;
    for(int i=istart-1; i<istart+2; i++)
        for(int j=jstart-1; j<jstart+2; j++)
            sum += img.at<unsigned char>(i,j);

    return (unsigned char)(sum/9);
}
```

FERONE LEZ 3

Filtraggio1_esercizio (Filtering1_CorrConv_Filter2D.cpp)

Implementare l'algoritmo di correlazione/convoluzione e confrontare l'output con quello prodotto dalla funzione filter2D() usando un filtro media.

Un modo semplice per creare una maschera con pesi tutti uguali è creare una matrice di 1 (attraverso Mat::ones(dim, dim, ddepth)), e poi dividerla per la dimensione della matrice. In questo caso usiamo un dim=3, dunque abbiamo una mask 3x3 con pesi w=1/9.

Se vogliamo creare una maschera ma con pesi diversi (media ponderata), dobbiamo prima creare un array inizializzando manualmente ogni valore, e poi creeremo un oggetto Mat passando questo array come struttura dati iniziale.

NB: Le maschere sono di tipo float!

Distinguiamo 3 funzioni: `correlation()`, `convolution()` e `CalcFilteredValue()`.

Correlazione

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

In `correlation(Mat src, Mat mask)`:

- passiamo come argomenti la matrice src e la maschera (di qls dimensione dispari) che vogliamo applicare.
- Essendo che dobbiamo restituire un oggetto Mat, creiamo un oggetto Mat `dst` della stessa dimensione di `src`, che andremo poi a restituire;
- Per poter appliccare il filtro, dobbiamo aggiungere il padding(in tal caso ho scelto il Reflect). l'ampiezza del bordo è calcolata come il numero di colonne (o righe) diviso 2, che sarà approssimato per difetto. Infatti se ho una mask 3x3 dovrò aggiungere 1 riga per ogni lato (3/2=1) dell'immagine, per una mask 5x5 dovrò aggiungere 2 righe per ogni lato (5/2=2), ecc.
- Questo doppio ciclo for si occuperà solo di scorrere sugli elementi della matrice dst. Ad ogni posizione (i,j) richiamerà la funzione `calcFilteredValue()` il quale restituirà il valore calcolato, e a cui passeremo la matrice col padding, la maschera, e i due indici (che ci serviranno come offset.
- ritorniamo la matrice.

In `convolution(Mat src, Mat mask)`:

- creiamo una Mat temporanea la quale conterrà la matrice `src` ruotata di 180 gradi.
- banalmente, richiamiamo `correlation` ma passando temp invece di `src`, e ritornandone il risultato.

In `calcFilteredValue(Mat neighb, Mat mask, int offx, int offy)`:

- `value=0` conterrà il valore finale che verrà restituito. Sebbene dovremo restituire un valore uchar, le operazioni sono di tipo `float` (abbiamo numeri con la virgola) e dunque lo dichiareremo come tale.
- il doppio ciclo for si occuperà di effetturare la somma dei prodotti (elemento per elemento) della maschera con un intorno (neighb) dim*dim dell'immagine con il padding. Notiamo 2 cose:
    - gli elementi di mask sono di tipo float;
    - gli elementi di neighb hanno un OFFSET, con il quale stabiliamo l'intorno. Ad es, se stessimo calcolato il pixel(0,2), allora: offx = 0, offy = 2. Ovviamente l'intorno di cui teniamo conto è quello del padding.
- infine ritorniamo il valore (castato, da float a uchar).

NB: rispetto alla versione vecchia, sono state effettuate alcune modifiche:

- ora è possibile effettuare l'applicazione del filtro per maschere di QUALSIASI dimensione dispari;
- il modo con cui itero è stato reso più chiaro ed elegante, ma funziona allo stesso modo. Basta ricordare che in correlation il doppio ciclo for lo usiamo per spostarci sulla matrice dst, mentre in calcFilteredValue ci spostiamo sugli elementi della maschera e dell'intorno (e dunque è dim*dim).

main

```
int main(int argc, char** argv){

    Mat img = imread(argv[1],IMREAD_GRAYSCALE);
    if(img.empty()) return -1;

    int dim=3;
    Mat mask = Mat::ones(dim,dim,CV_32F)/(float)(dim*dim);
    printMask(mask);

    /* weighted average mask
    float data[]={
        1.0,2.0,1.0,
        2.0,4.0,2.0,
        1.0,2.0,1.0
    };
    for(int i=0; i<9; i++) data[i] /= 16.0;
    Mat wmask = Mat(3,3,CV_32F,data);
    */

    Mat filter2Dimg;
    filter2D(img,filter2Dimg,img.type(),mask);

    Mat corrImg = correlation(img, mask);

    Mat convImg = convolution(img, mask);

    imshow("original",img);
    imshow("filter2D",filter2Dimg);
    imshow("correlation",corrImg);
    imshow("convolution",convImg);

    waitKey(0);
    return 0;
}
```

```
Mat correlation(Mat src, Mat mask){
    Mat dst(src.rows,src.cols,src.type());

    Mat pad;
    int bw = mask.rows/2; //border width, 3/2 =1, 5/2 =2, etc.
    copyMakeBorder(src,pad,bw,bw,bw,bw,BORDER_REFLECT);

    for(int i=0; i<dst.rows; i++)
        for(int j=0; j<dst.cols; j++)
            dst.at<uchar>(i,j) = calcFilteredValue(pad,mask,i,j);

    return dst;
}
```

```
Mat convolution(Mat src, Mat mask){
    Mat temp;
    rotate(src,temp,ROTATE_180);
    return correlation(temp, mask);
}
```

```
uchar calcFilteredValue(Mat neighb, Mat mask, int offx, int offy){
    float value=0;
    for(int i=0; i<mask.rows; i++)
        for(int j=0; j<mask.cols; j++)
            value += mask.at<float>(i,j)*neighb.at<uchar>(i+offx,j+offy);
    return uchar(value);
}
```

FERONE LEZ 4

Filtraggio2_esercizi (Filtering2_LapSobel.cpp)

Implementare il Laplaciano con kernel isotropico a 45°e 90°utilizzando la funzione di correlazione/convoluzione (o filter2D()).
Per normalizzare i livelli di griggio è possibile usare la funzione `normalize()`.

```
normalize(src, dst, 0, 255, NORM_MINMAX, CV_8U);
```

Utilizzo la funzione `myLaplacian()`, alla quale passo l'immagine di partenza, il tipo, e il ksize (per stabilire quale filtro usare).

Infatti la prima cosa che facciamo è *l'assegnazione del Kernel*, attraverso uno switch. Ovviamente abbiamo due tipi di filtro: quello isotropico a 45° e quello a 90°.

Poi applichiamo la maschera: usiamo filter2D per applicare la maschera.

Infine ritorniamo l'immagine ottenuta.

```
Mat myLaplacian(Mat src, int ddepth, int ksize){

    //Kernel assignment
    float data[9];
    switch(ksize){
    case 1:
        {
        float data90[]={
            0.0,1.0,0.0,
            1.0,-4.0,1.0,
            0.0,1.0,0.0
        };
        copy(data90, data90+sizeof(data90)/sizeof(data90[0]), data);
        break;
        }

    case 3:
        {
        float data45[]={
            1.0,1.0,1.0,
            1.0,-8.0,1.0,
            1.0,1.0,1.0
        };
        copy(data45, data45+sizeof(data45)/sizeof(data45[0]), data);
        break;
        }
    }

    //Mask application
    Mat mask = Mat(3,3,CV_32F,data);
    Mat dst, imgF2D;
    filter2D(src,dst,ddepth,mask);

    return dst;
}
```

Implementare il filtro di Sobel (g_x e g_y) utilizzando la funzione di correlazione/convoluzione (o filter2D()).

- Calcolare la risposta di entrambi i filtri;
- Calcolare la magnitudo del gradiente (entrambe le formulazioni)

Presumo che calcolare il filtro di Sobel sia più o meno la stessa cosa: dunque applichiamo la maschera con filter2D() e poi normalizziamo.

Per calcolare la magnitudo del gradiente:

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

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

Praticamente creiamo le maschere tenendo conto dei filtri di Sobel per la derivata rispetto ad x e quella rispetto ad y (SobelX sarebbe Dx, SobelY sarebbe Dy).
Imitando la funzione di Sobel originale, (1,0) ci sarà Dx mentre (0,1) ci darà Dy, infine (1,1) ci darà la combinazione fra le due (che alla fine sarebbe la magnitudo).

Come nel caso del laplaciano, creata la maschera basterà fara la convoluzione.

Il caso di Sobel(1,1) è più particolare: non mi trovo esattamente con la funzione di Sobel originale (l'immagine dovrebbe essere molto più scura, invece i pixel hanno la stessa intensità di Dx e Dy).

Utilizzare la risposta ottenuta per effettuare lo sharpening di un'immagine.

Basta semplicemente applicare questa formula:

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

cioè, sommo l'immagine all'immagine su cui ho applicato il filtro moltiplicata per una certa costante c. Più *c* si avvicina a -1, più esalto i bordi. Se c=0, non ottengo alcun effetto.

```
Mat test;

float c=-1; // -1 <= c <= 0

test = (img+c*lapImg45);
```

```
Mat mySobel(Mat src, int ddepth, int xorder, int yorder){

    float sobelX[]={
        -1.0,-2.0,-1.0,
        0.0,0.0,0.0,
        1.0,2.0,1.0
    };

    float sobelY[]={
        -1.0,0.0,1.0,
        -2.0,0.0,2.0,
        -1.0,0.0,1.0
    };

    //Get Filters
    Mat maskSX = Mat(3,3,CV_32F,sobelX);
    Mat maskSY = Mat(3,3,CV_32F,sobelY);

    //Mask application
    Mat dst;
    if(xorder == 1 && yorder == 0){ //SobelX
        filter2D(src,dst,ddepth,maskSX);
    }
    else if(xorder == 0 && yorder == 1){ //SobelY
        filter2D(src,dst,ddepth,maskSY);
    }
    else if(xorder == 1 && yorder == 1){

        Mat imgSobX = mySobel(src,ddepth,1,0);
        Mat imgSobY = mySobel(src,ddepth,0,1);

        dst = abs(ImgSobX)+abs(ImgSobY);
    }

    return dst;
}
```

FERONE LEZ 6

Segmentazione1_esercizi (Seg1_LoGwithOpenCVfuncs.cpp)

Utilizzando le funzioni di OpenCV, implementare il LoG.

Per fare ciò:

1. ottenere il filtro gaussiano attraverso la funzione `getGaussianKernel()` e applicare il filtro all'immagine (oppure usare direttamente la funzione `GaussianBlur()` con cui otteniamo lo stesso effetto);
2. applicare il laplaciano all'immagine filtrata. Il risultato sarà il LoG;
3. effettuare lo sharpening.

1/2 – Otteniamo LoG (nb: 1 sarebbe ksize, messo per sbaglio)

```
Mat myLoG(Mat src, int ddepth, int ksize){
    Mat GBImg, LoG;
    GaussianBlur(src, GBImg, Size(3,3),0,0);
    Laplacian(GBImg,LoG,ddepth,1);
    return LoG;
}
```

```
Mat Gkernel = getGaussianKernel(4,25);
filter2D(src, GBImg,ddepth,Gkernel);
```

3 – Effettuiamo lo sharpening

```
float c=-1;
Mat LoG = myLoG(src,CV_8U,1);
Mat output = (src+c*LoG);
```

DB dove trovare immagini per fare test:
nb: https://www2.eecs.berkeley.edu/Research/Projects/CS/vision/bsds/

Implementare una funzione che trovi i punti di 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**.

FERONE LEZ 7

Segmentazione2_esercizi (canny.cpp / harris.cpp)

Algoritmo di Canny ed Harris con funzioni di OpenCV

• Canny

In OpenCV, la funzione che implementa l'algoritmo di Canny è:

```
void cv::Canny  (
      InputArray      image,
      OutputArray     edges,
      double          TH,
      double          TL,
      int             apertureSize = 3,
      bool            L2gradient = false
)
```

T_H e T_L sono rispettivamente la soglia alta e la soglia bassa.
`apertureSize` è la dimensione del kernel di Sobel utilizzato per trovare i gradienti dell'immagine (di default è 3).
`L2gradient` specifica quale equazione deve essere usata per calcolare la magnitudo del gradiente: se è True, usa la più accurata $\sqrt{G_x^2 + G_y^2}$, se è False usa l'approssimazione $|G_x| + |G_y|$. Di default, è False.

Di solito la soglia T_H è calcolata come "percentuale" rispetto a T_L (ad es. T_H è 3 volte T_L. TH = TL*ratio, dove ratio = 3).

È bene effettuare un po' di blurring prima di applicare l'algoritmo.

• Harris

In OpenCV, la funzione che implementa l'algoritmo di Harris è:

```
void cv::cornerHarris (
      InputArray      src,
      OutputArray     dst,
      int             blockSize,
      int             ksize,
      double          k,
      int             borderType = BORDER_DEFAULT
)
```

`blockSize` è la dimensione dei vicini (es. 2), `ksize` è la dimensione del kernel di Sobel utilizzato per trovare i gradienti dell'immagine (es. 3), `k` è un parametro arbitraio (es. 0.04).

nb: applicata la funzione, dovremo poi normalizzare il risultato con "normalize()" e poi applicare a questo "convertScaleAbs()" per scalare, calcolare i valori assoluti, e convertire il risultato a 8bit.

Possiamo anche evidenziare i corner attraverso la funzione `circle()`.

```
void cv::circle (

InputOutputArray      img,
Point                 center,
int                   radius,
const Scalar &        color,
int                   thickness = 1,
int                   lineType = LINE_8,
int                   shift = 0
)
```

Questa funzione disegnerà dei cerchi con un certo centro e raggio.
lineType sarebbe tipo 4-intorno, 8-intorno, ecc…

Qui ho chiamato la funzione `circleCorners()`

Implementazioni con OpenCV

Canny

```
1       Mat blurSrc, CEdges;
2       blur(src,blurSrc,Size(3,3));
3       Canny(blurSrc,CEdges,35,85);
```

Harris

```
1       Mat CHarris, CH_norm, CH_norm_scaled;
2       int blockSize=2, hksize=3;
3       float k=0.06;
4       cornerHarris(src,CHarris, blockSize, hksize,k);
5       normalize(CHarris,CH_norm,0,255,NORM_MINMAX, CV_32FC1,Mat());
6       convertScaleAbs(CH_norm, CH_norm_scaled);
7
8       circleCorners(CH_norm,CH_norm_scaled,200);
```

- circleCorners

```
1  void circleCorners(Mat &src, Mat &dst, int thresh){
2      for(int i=0; i<src.rows; i++)
3          for(int j=0; j<src.cols; j++)
4              if( (int)src.at<float>(i,j) > thresh)
5                  circle(dst,Point(j,i),5,Scalar(0),2,8,0);
6  }
```

Implementa l'algoritmo di Canny

Ricordiamo dagli "Appunti ELIM – Ferone":

[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]

Dunque:

1.

Facciamo convolvere all'immagine originale `src` un filtro gaussiano attraverso la funzione `GaussianBlur`.

2.

Per calcolare la magnitudo e l'angolo dobbiamo innanzitutto calcolare le derivate parziali D_x e D_y, derivate che possiamo ottenere attraverso il filtro di Sobel a cui passiamo l'immagine filtrata con Gauss (`Gauss`).

Calcoliamo la **magnitudo**: applichiamo la formula $\sqrt{G_x^2 + G_y^2}$
Dx2 e Dy2 sono le der. parziali al quadrato.
Poi normalizzo fra 0 e 255 (NORM_MINMAX è una macro, praticamente pone `alpha` come il valore minimo dell'immagine, e `beta` come il valore massimo).

Calcoliamo la **direzione/angolo** (orientations): utilizziamo la funzione di openCV `phase`(Dx, Dy, orientations, AngleInDegrees = false), nella quale passiamo le derivate parziali e ci restituisce in `orientations` le orientazioni.
Quando AngleInDegrees = true, la funzioni calcola gli angoli in gradi, altrimenti li calcola in radianti.

3.

Avendo la Magnitudo e la direzione, posso ora calcolare la Non Maxima Suppression attraverso l'analoga funzione.

4.

Applico il thresholding con istèresi attraverso l'analoga funzione.

```
void canny(Mat& src, Mat& dst, int lth, int hth, int ksize){

    //1
    Mat gauss;
    GaussianBlur(src,gauss, Size(3,3),0,0);

    //2
    Mat Dx, Dy, Dx2, Dy2, mag, orientations;
    Sobel(gauss, Dx, CV_32FC1, 1, 0, ksize);
    Sobel(gauss, Dy, CV_32FC1, 0, 1, ksize);
    pow(Dx,2,Dx2);
    pow(Dy,2,Dy2);
    sqrt(Dx2+Dy2,mag);
    normalize(mag,mag,0,255,NORM_MINMAX,CV_8UC1);
    phase(Dx,Dy,orientations,true);

    //3
    Mat nms = Mat::zeros(src.rows,src.cols,CV_8UC1);
    nonMaximaSuppression(mag,orientations,nms);

    //4
    dst = Mat::zeros(src.rows,src.cols,CV_8UC1);
    hysteresisThreshold(nms, dst, lth, hth);
}
```

• **Implementazione della nonMaximaSuppression**

Passiamo alla funzione la Magnitudo `mag`, la direzione `orientations` (calcolate prima), e una matrice `nms` della stessa dimensione di `mag` nella quale verrà conservato l'output.

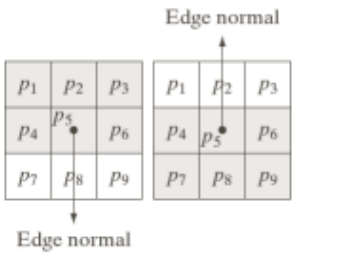
La matrice `nms` è inizializzata a 0, in quanto dovrà contenere solo gli edge forti: infatti se un pixel (i,j) è considerato edge forte assumerà lo stesso valore del pixel (i,j) di `mag`. Altrimenti, lo rimango a 0 (in un certo senso, è come se lo "avessi rimasto soppresso").

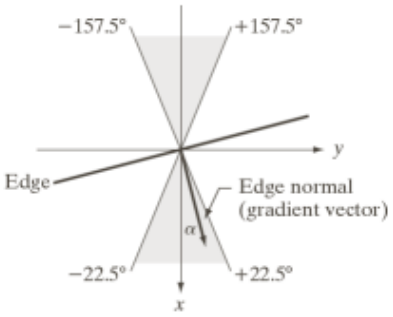
Scorriamo su tutta la Magnitudo (ignorando la prima e l'ultima riga/colonna).
`angle` sarà una variabile temporanea che conterrà il valore di `orientations` nella posizione (i,j). Innanzitutto dobbiamo però portare l'angolo dall'intervallo 0° a 360° all'intervallo -180° a 180° (cosa che possiamo fare facilmente attraverso un controllo: se `angle` > 180 allora gli sottraiamo 360. ES, 195 diventa -15).

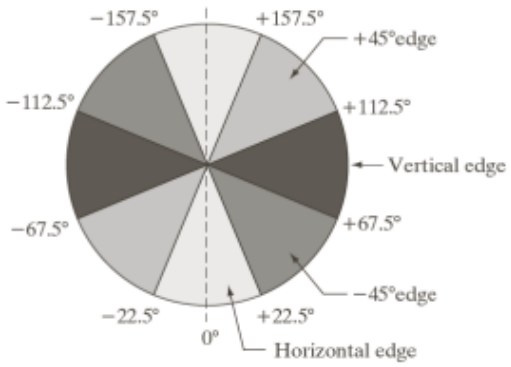
Utilizzeremo 4 blocchi if per identificare i 4 possibili orientamenti: verticale, orizzontale, +45° e -45°. Dove per ogni orientamento distinguiamo due intervalli opposti fra di loro. Per ogni pixel (i,j), che è il pixel centrale che sto considerando nell'intorno, verifico se è maggiore dei suoi vicini in quella direzione (in tal caso, è un edge forte).

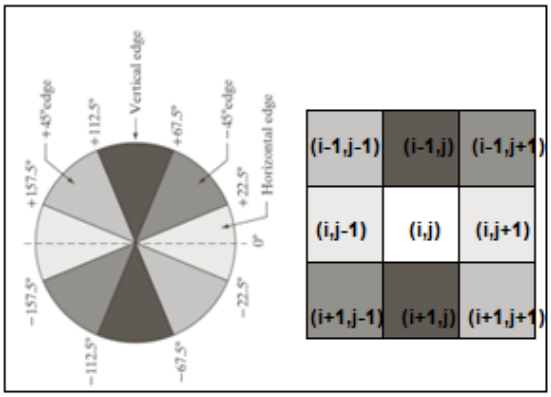
consideriamo in ordine: orizzontale, +45°, verticale, -45° (andiamo in senso orario partendo da -22.5°).

- *orizzontale*: i vicini sono quelli a sinistra e a destra del pixel centrale, dunque risp. `(i,j-1)` e `(i, j+1)`;
- *+45°*: i vicini solo quelli sulla diagonale principale, dunque rispettivamente `(i-1,j-1)` e `(i+1,j+1)`;
- *verticale*: i vicini sono quelli sopra e sotto il pixel centrale, dunque rispettivamente `(i-1, j)` e `(i+1, j)`;
- *-45°*: i vicini sono quelli sulla diagonale secondaria, dunque rispettivamente `(i-1, j+1)` e `(i+1, j-1)`;









nb: possiamo anche operare al contrario, ovvero partiamo dall'immagine `mag` (o copiamo `mag` in `nms`) e poniamo a 0 i pixel che non rispettano le condizioni.

• **Implementazione della hysteresisThresholding**

Scorro su tutti i pixel dell'immagine (con un doppio ciclo for), e per ogni pixel verifico se è maggiore della soglia alta hth: in tal caso verifico se nel suo intorno sono presenti pixel compresi fra la soglia alta e la soglia bassa (dunque dovremo scorrere in un intorno 3x3) così da promuoverli a pixel forti (intensità a 255).
Se il pixel NON è maggiore della soglia alta hth, lo lascio a 0.

CODICE

```
void nonMaximaSuppression(Mat& mag, Mat& orientations, Mat& nms){
    float angle;
    for(int i=1; i<mag.rows-1; i++){
        for(int j=1; j<mag.cols-1; j++){
            angle = orientations.at<float>(i,j);
            angle = angle > 180 ? angle - 360 : angle;
            if((-22.5 < angle && angle <= 22.5) || (157.5 < angle && angle <= -157.5)){ //horizontal
                if(mag.at<uchar>(i,j) > mag.at<uchar>(i,j-1) && mag.at<uchar>(i,j) > mag.at<uchar>(i,j+1)){
                    nms.at<uchar>(i,j) = mag.at<uchar>(i,j);
                }
            }
            else if((-67.5 < angle && angle <= -22.5) || (112.5 < angle && angle <= 157.5)){ //+45
                if(mag.at<uchar>(i,j) > mag.at<uchar>(i-1,j+1) && mag.at<uchar>(i,j) > mag.at<uchar>(i+1,j-1)){
                    nms.at<uchar>(i,j) = mag.at<uchar>(i,j);
                }
            }
            else if((-112.5 < angle && angle <= -67.5) || (67.5 < angle && angle <= 112.5)){ //vertical
                if(mag.at<uchar>(i,j) > mag.at<uchar>(i-1,j) && mag.at<uchar>(i,j) > mag.at<uchar>(i+1,j)){
                    nms.at<uchar>(i,j) = mag.at<uchar>(i,j);
                }
            }
            else if((-157.5 < angle && angle <= -112.5) || (67.5 < angle && angle <= 22.5)){ //-45
                if(mag.at<uchar>(i,j) > mag.at<uchar>(i-1,j-1) && mag.at<uchar>(i,j) > mag.at<uchar>(i+1,j+1)){
                    nms.at<uchar>(i,j) = mag.at<uchar>(i,j);
                }
            }
        }
    }
}
```

Nb: Fare estrema attenzione. È molto facile confondersi con gli angoli.

```
void hysteresisThresholding(Mat &img, Mat &out, int lth, int hth){

    for(int i=1; i<img.rows-1; i++){
        for(int j=1; j<img.cols-1; j++){
            if(img.at<uchar>(i,j) > hth){
                out.at<uchar>(i,j) = 255;
                for(int k=-1; k<=1; k++){
                    for(int l=-1; l<=1; l++){
                        if(img.at<uchar>(i+k,j+l) > lth && img.at<uchar>(i+k,j+l) < hth)
                            out.at<uchar>(i+k,j+l) = 255;
                    }
                }
            }
            else if(img.at<uchar>(i,j < lth))
                out.at<uchar>(i,j) = 0;
        }
    }

}
```

Implementa l'algoritmo di Harris

Ricordiamo dagli "Appunti ELIM – Ferone":

[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]; *…e applicare convertScaleAbs();
7. infine sogliare R. **]**

Dunque:

1.

Calcoliamo le derivate parziale Dx e Dy con Sobel.

2.

Calcolare Dx2, Dy2, Dxy attraverso le funzioni `pow()` e `multiply()`.

3/4.

Usiamo `GaussianBlur()` per applicare un filtro gaussiano a Dx2, Dy2 e Dxy.
Il risultato lo conserviamo nelle componenti C00, C01, C10. C11 (che sono appunto Dx2, Dxy, Dxy e Dy2 dopo aver applicato il filtro).

5.

Per calcolare l'indice R, ricordiamo:

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

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

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

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

Calcoliamo il **determinante**: è dato dal prodotto della diagonale principale (PPD) meno il prodotto della diagonale secondario (PSD).

Calcoliamo la **traccia**: è data dalla somma degli elementi sulla diagonale principale, cioè C00 e C11. Calcoliamoci per dopo anche `trace`^{2}.

Possiamo finalmente calcolare **R** applicando la formula.

6.

Normalizziamo R con `normalize()` ed applichiamo convertScaleAbs()* (quest'ultimo non è presente nell'algoritmo originale ma la funzione originale cornerHarris ne ha bisogno, e in effetti se vogliamo un risultato visualizzabile dobbiamo applicarlo).

Dunque `R` sarà Harris normalizzato e `dst` sarà Harris normalizzato scalato.

dst contiene l'immagine finale.

7.

Sogliamo R, nel senso che indichiamo con dei cerchietti i punti dove sono presenti i corner. Lo facciamo attraverso la nostra personale funzione `circleCorners()`.

(questo passo non è fondamentale ai fini dell'algoritmo, è solo per rappresentazione).

- ricordiamo circleCorners

```
1 ▾ void circleCorners(Mat &src, Mat &dst, int thresh){
2       for(int i=0; i<src.rows; i++)
3           for(int j=0; j<src.cols; j++)
4               if( (int)src.at<float>(i,j) > thresh)
5                   circle(dst,Point(j,i),5,Scalar(0),2,8,0);
6   }
```

CODICE - Harris

```
void myHarris(Mat &src, Mat &dst, int ksize, float k, int thresh){

    //1.
    Mat Dx,Dy;
    Sobel(src,Dx,CV_32FC1,1,0,ksize);
    Sobel(src,Dy,CV_32FC1,0,1,ksize);

    //2.
    Mat Dx2,Dy2,Dxy;
    pow(Dx,2,Dx2);
    pow(Dy,2,Dy2);
    multiply(Dx,Dy,Dxy);

    //3-4.
    Mat C00,C01,C10,C11;
    GaussianBlur(Dx2,C00,Size(7,7),2,0);
    GaussianBlur(Dy2,C11,Size(7,7),0,2);
    GaussianBlur(Dxy,C01,Size(7,7),2,2);
    C10=C01;

    //5.
    Mat det, PPD, PSD, trace, trace2, R;

    multiply(C00,C11,PPD);  //Product of the Principal Diagonal
    multiply(C01,C10,PSD);  //Product of the Secondary Diagonal
    det = PPD-PSD;

    trace = C00+C11;
    pow(trace,2,trace2);

    R = det-k*trace2;

    //6.
    normalize(R,R,0,255,NORM_MINMAX,CV_32FC1);
    convertScaleAbs(R, dst);

    //7.
    circleCorners(R,dst,thresh);
}
```

FERONE LEZ 8

Segmentazione3_esercizi (houghLines.cpp / houghCircles.cpp)

Implementazioni con OpenCV

Hough per rette

Ricordandoci la funzione di OpenCV: HoughLines, dobbiamo passare a questa direttamente gli edge dell'immagine (es. Canny) e ci restituirà in output un vettore di rette (2 canali, ro e theta).
Per poter visualizzare tale vettore disegneremo ogni singola retta attraverso un ciclo for.

- **drawHoughLines()**

le rette di Hough sono espresse in coordinate polari, quindi dovremo prima convertirle nella forma canonica.

Per fare ciò, dovremo scorrere sull'intero vettore di rette attraverso un ciclo for, disegnando una retta sull'immagine ad ogni iterazione.

Ricordiamo che ogni retta è espressa dalla formula: **ρ = x cosθ + y sinθ**
dunque ogni retta è identificata dai parametri ρ e θ. Essendo questo un vettore di rette, `lines[i][0]` rappresenterà il rho dell'i-sima retta, `lines[i][1]` rappresenterà il theta dell'i-sima retta.

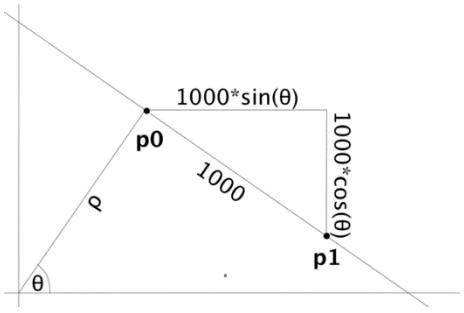
Per semplificare i prossimi calcoli, poniamo `cos_t = cos(theta)` e `sin_t = sin(theta)`.

Individuiamo x_0 e y_0 che ci aiuteranno ad individuare i due punti sulla retta.
dove:

- `x0 = rho*cos(theta)`, dunque `rho*cos_t`;
- `y0 = rho*sin(theta)`, dunque `rhp*sin_t`;

I due punti `pt1` e `pt2` dovranno essere spostati lungo la retta (per questo moltiplichiamo per alpha) ma in direzioni opposte dalla posizione iniziale (per questo in `pt1` sommiamo mentre in `pt2` sottraiamo).

Utilizziamo cvround per arrotondare il double al suo valore intero più vicino.



Per disegnare la retta utilizziamo la funzione di OpenCV: `line()`.
Passiamo l'immagine di destinazione (nel nostro caso la copia di src, ovvero dst), i due punti, definiamo il colore delle retta, lo spessore, e il tipo.

Hough per cerchi

Ricordandoci la funzione di OpenCV: HoughLines, dobbiamo passare a questa solo l'immagine originale, e ci restituirà in output un vettore di circonferenze (3 canali: a, b e R). Per poter visualizzare tale vettore disegneremo ogni singola cerchio attraverso un ciclo for.

- **drawHoughCircles()**

le circonferenze sono espresse dai parametri (a,b) ed R, ovvero il centro e il raggio. Passiamo il vettore di cerchi calcolato con l'Hough per cerchi, dove: `circles[i][0]` rappresenta l'ascissa del centro, `circles[i][1]` rappresenta l'ordinata del centro, e `circles[i][2]` rappresenta il raggio.

Per disegnare la circonferenza utilizziamo la funzione di OpenCV: `circle()`.
Passiamo l'immagine di destinazione (nel nostro caso la copia di src, ovvero dst), il centro, il raggio, definiamo il colore delle retta, lo spessore, e il tipo.

In questo caso facciamo anche una distinzione fra "centro" e "bordo".

nb: sia in `drawHoughlines()` che in `drawHoughCircles()` viene innanzitutto clonata src in dst MA convertendo il tipo in BGR, questo perché disegniamo delle rette(circonferenze a colori).

CODICE PER DISEGNARE le rette e i cerchi

```
void drawHoughLines(Mat& src, Mat& dst, vector<Vec2f>& lines){

    cvtColor(src,dst,COLOR_GRAY2BGR);
    float rho, theta;
    double cos_t, sin_t, x0, y0;
    int alpha = 1000;
    Point pt1, pt2;

    for(size_t i=0; i<lines.size(); i++){
        rho = lines[i][0]; theta = lines[i][1];
        cos_t = cos(theta); sin_t = sin(theta);
        x0 = rho*cos_t; y0 = rho*sin_t;
        pt1.x = cvRound(x0 + alpha*(-sin_t));
        pt1.y = cvRound(y0 + alpha*(cos_t));
        pt2.x = cvRound(x0 - alpha*(-sin_t));
        pt2.y = cvRound(y0 - alpha*(cos_t));
        line(dst, pt1, pt2, Scalar(0,0,255),3,16); //CV_AA = 16
    }

}
```

```
void drawHoughCircles(Mat &src, Mat& dst, vector<Vec3f>& circles){

    cvtColor(src,dst,COLOR_GRAY2BGR);
    Point center;
    int radius;

    for(size_t i=0; i<circles.size(); i++){
        center.x = cvRound(circles[i][0]);
        center.y = cvRound(circles[i][1]);
        radius = cvRound(circles[i][2]);

        // circle center and outline
        circle(dst, center, 3, Scalar(0,255,0), -1, 8, 0);
        circle(dst, center, radius, Scalar(0,0,255), 3, 8, 0);
    }

}
```

Esempi di implementazioni con OpenCV

Hough per rette

```
1  //src = ...
2  Mat edgeCanny;
3  Canny(src, edgeCanny,80,160,3);
4
5  vector<Vec2f> HLines;
6  int rho = 1, th = 150;
7  double theta = CV_PI/180;
8  HoughLines(edgeCanny,HLines,rho,theta,th,0,0);
9
10 Mat dst;
11 drawHoughLines(src,dst,HLines);
```

Hough per cerchi

```
1  //src = ...
2  vector<Vec3f> HCircles;
3  int dimH = 10, minDist = src.rows/8;
4  int CannyTh = 100, HTh = 100;
5  HoughCircles(src,HCircles,HOUGH_GRADIENT,dimH, minDist, CannyTh, HTh, 0, 20);
6
7  Mat cdst;
8  drawHoughCircles(src,cdst,HCircles);
```

Implementa l'algoritmo di Hough per rette

Ricordiamo dagli "Appunti ELIM – Ferone":

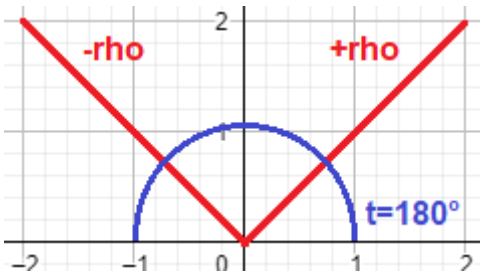
[L'algoritmo di Harris (per rette) consiste nei seguenti **passi fondamentali**:

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. **]**

Dunque:

1.

Creiamo lo **spazio dei voti**, che sarà inizialmente una matrice di tutti 0, che dovrà avere una dimensione tale da poter considerare tutte le possibili rette che passano per il piano immagine. Questo corrisponde a considerare una possibile distanza ρ (che andrà da ρ a – ρ) e dei possibili angoli θ, quindi la nostra quantizzazione avrà ρ unità e considererà θ angoli.



La distanza massima di ρ `maxDist` è di fatto la diagonale, che può essere calcolata come l'ipotenusa di un triangolo i quali cateti sono rispettivamente l'altezza e la larghezza dell'immagine.
Possiamo usare `hypot()` per effettuare facilmente questo calcolo.

Scrivere: `vector<vector<int>> votes(maxDist*2, vector<int>(180, 0));`
significa che ogni cella avrà valore 1 e coprirà 1 grado.

nb: anche se consideriamo θ=180°, in realtà consideremo gli angoli che vanno da -90° a 90°.

2.

Applichiamo l'algoritmo di Canny per individuare i punti di edge nell'immagine.
Effettuiamo anche del blurring prima così da rimuovere un po' di rumore.

3.

Effettuiamo la scansione dell'immagine di Canny. Ci interessano solo i punti di edge, ovvero tutti quei punti il quale valore è 255. Questi punti voteranno nello spazio di voti.

Per votare i pixel di edge dovranno tener conto di tutte le possibili direzioni, che in questo caso sarebbero tutte quelle comprese in un angolo da -90° a 90°.

Nell'istruzione seguente calcoliamo il rho tenendo conto della forma normale:

```
rho = round(y*cos(theta-90) + x*sin(theta-90)) + maxDist;
```

nelle funzioni `sin` e `cos` sottraggo 90 a theta, così da ritrovarmi nell'intervallo [-90,90].

Nb: il motivo per cui x e y sono invertite rispetto alla formula originale è perché in realtà non siamo nel 1° quadrante ma nel 4° (infatti l'immagine è disegnata dall'angolo in alto a sx, le quali coordinate sono (0,0). Questo vale sempre).

Nb: al valore calcolato dalla formula, dobbiamo aggiungere maxDist.

Dunque, per ogni singola iterazione fisso un angolo theta, calcolo il rho, e per questa coppia (ro,theta) inserisco il voto nello spazio dei voti.
L'intero ciclo for corrisponde ad una sinusoide. Un altro punto di edge corrisponderà ad un'altra sinusoide.

4.

Terminato il voto, dovrò andare a considerare lo spazio di voti.
Dovrò estrarre dallo spazio di voto le celle che hanno un numero di voti maggiore di una certa soglia (`Hth`).
Il risultato verrà inserito in un'altra matrice (clone di src) che infine andremo a restituire.

Innanzitutto, nel ciclo for:

- dovrò portare **rho** al suo valore originale. Ricordiamo che il valore di rho originale sarebbe quello dato dalla formula della retta normale. Nel nostro caso:
  `rho = round(y*cos(theta-90) + x*sin(theta-90))`
  questo significa che per ottenere quel valore dobbiamo sottrarre `maxDist` a `i`.
- Dovrò portare **theta** in un'intervallo compreso fra [-90,90], dunque dobbiamo sottrarre `90` a `j`.

Siamo nelle coordinate polari, dunque per disegnare una retta sull'immagine dovremo prima individuare le coordinate cartesiane. Faremo ciò attraverso la funzione `polarToCartesian()`.

- `polarToCartesian()`

Questa funzione si occupa di convertire le coordinate polari in coordinate cartesiane. In particolar modo, passeremo `rho` e `theta` e verranno restituiti due punti `p1` e `p2` (in realtà li passiamo per riferimento).

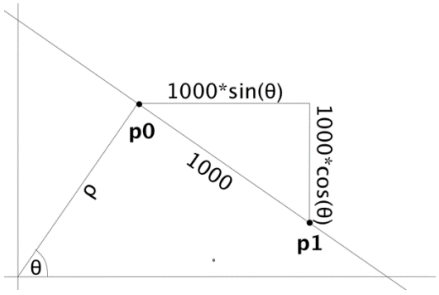
`alpha` rappresenta la lunghezza della retta da disegnare.

Individuiamo x_0 e y_0 che ci aiuteranno ad individuare i due punti sulla retta.
dove:

- `x0 = rho*cos(theta)`
- `y0 = rho*sin(theta)`

I due punti `pt1` e `pt2` dovranno essere spostati lungo la retta (per questo moltiplichiamo per alpha) ma in direzioni opposte dalla posizione iniziale (per questo in `pt1` sommiamo mentre in `pt2` sottraiamo).

Utilizziamo cvround per arrotondare il double al suo valore intero più vicino.




```
1  void polarToCartesian(double rho, int theta, Point& p1, Point& p2){
2      int alpha = 1000;
3
4      int x0 = cvRound(rho*cos(theta));
5      int y0 = cvRound(rho*sin(theta));
6
7      p1.x = cvRound(x0 + alpha*(-sin(theta)));
8      p1.y = cvRound(y0 + alpha*(cos(theta)));
9
10     p2.x = cvRound(x0 - alpha*(-sin(theta)));
11     p2.y = cvRound(y0 - alpha*(cos(theta)));
12 }
```

…infine,
per disegnare la retta utilizziamo la funzione di OpenCV: `line()`.
passiamo l'immagine di destinazione (nel nostro caso la copia di src, ovvero dst), i due punti, definiamo il colore delle retta, lo spessore, e il tipo.

Codice:

```
void houghLines(Mat& src, Mat& dst, int cannyLTH, int cannyHTH, int HoughTH){

    //1
    int maxDist = hypot(src.rows, src.cols);
    vector<vector<int>> votes(maxDist*2, vector<int>(180, 0));

    //2
    Mat gsrc, edges;
    GaussianBlur(src,gsrc,Size(3,3),0,0);
    Canny(gsrc,edges,cannyLTH,cannyHTH);

    //3
    double rho;
    int theta;
    for(int x=0; x<edges.rows; x++)
        for(int y=0; y<edges.cols; y++)
            if(edges.at<uchar>(x,y) == 255)
                for(theta = 0; theta <= 180; theta++){
                    rho = round(y*cos(theta-90) + x*sin(theta-90)) + maxDist;
                    votes[rho][theta]++;
                }

    //4 - drawing
    dst=src.clone();
    Point p1, p2;
    for(size_t i=0; i<votes.size(); i++)
        for(size_t j=0; j<votes[i].size(); j++)
            if(votes[i][j] >= HoughTH){
                rho = i-maxDist;
                theta = j-90;
                polarToCartesian(rho,theta,p1,p2);
                line(dst,p1,p2,Scalar(0,0,255),2,LINE_AA);
            }
}
```

Esempio nel **main():**

```
Mat dst;
int cannyLTH = 50;
int cannyHTH = 150;
int HoughTH = 100;
houghLines(src,dst,cannyLTH,cannyHTH,HoughTH);
```

Implementa l'algoritmo di Hough per cerchi

Ricordiamo dagli "Appunti ELIM – Ferone":

[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. **]**

Dunque:

prima di cominciare, applichiamo un blurring Gaussiano, così da ripulire un po' l'immagine.

1.

Creiamo lo **spazio dei voti**, che sarà tridimensionale. In questo caso i 3 parametri (a,b,R) saranno rappresentati rispettivamente da `edgeCanny.rows`, `edgeCanny.cols` e un *intervallo di valori* per R (consideriamo un min e un max).
Alla creazione la matrice sarà tutti 0.

2.

Applichiamo l'algoritmo di Canny per individuare i punti di edge nell'immagine.
Effettuiamo anche del blurring prima così da rimuovere un po' di rumore.

3.

Effettuiamo la scansione dell'immagine di Canny. Ci interessano solo i punti di edge, ovvero tutti quei punti il quale valore è 255. Questi punti voteranno nello spazio di voti.

In questo caso abbiamo anche un terzo ciclo for che scorre anche per tutti i raggi compresi fra Rmin e Rmax.

Per votare i pixel di edge dovranno tener conto di tutti i possibili centri (di ogni raggio), che in questo caso sarebbero tutti quelle compresi in un angolo da 0° a 180°.
Poi applichiamo la formula per calcolare *a* e *b* (il centro del cerchio). Anche in questo caso invertiamo x e y nella formula (stesso motivo di Hough per rette).

Calcolate le coordinate, controllo che queste siano interne all'immagine (non consideriamo quelle esterne).

4.

Terminato il voto, dovrò andare a considerare lo spazio di voti.
Dovrò estrarre dallo spazio di voto le celle che hanno un numero di voti maggiore di una certa soglia (`Hth`).

In questo caso li disegniamo anche. Nello specifico, disegniamo per ogni cerchio un cerchietto piccolo (il centro) e la sua circonferenza.

Codice

```
void houghCircles(Mat& src, Mat& dst, int cannyLTH, int cannyHTH, int HoughTH,
                  int Rmin, int Rmax){

    //1
    vector<vector<vector<int>>>
        votes(src.rows, vector<vector<int>>(src.cols, vector<int>(Rmax-Rmin+1,0)));

    //2
    Mat gsrc, edges;
    GaussianBlur(src,gsrc,Size(7,7),0,0);
    Canny(gsrc,edges,cannyLTH,cannyHTH);

    //3
    for(int x=0; x<edges.rows; x++)
        for(int y=0; y<edges.cols; y++)
            if(edges.at<uchar>(x,y) == 255)
                for(int r=Rmin; r<=Rmax; r++)
                    for(int theta=0; theta<360; theta++){
                        int a = y - r*cos(theta*CV_PI/180);
                        int b = x - r*sin(theta*CV_PI/180);
                        if(a>=0 && a<edges.cols && b>=0 && b<edges.rows)
                            votes[b][a][r-Rmin]++;
                    }

    //4
    dst=src.clone();
    for(int r=Rmin; r<Rmax; r++)
        for(int b=0; b<edges.rows; b++)
            for(int a=0; a<edges.cols; a++)
                if(votes[b][a][r-Rmin] > HoughTH){
                    circle(dst, Point(a,b), 3, Scalar(0), 2, 8, 0);
                    circle(dst, Point(a,b), r, Scalar(0), 2, 8, 0);
                }
}
```

Esempio nel **main():**

```
Mat dst;
int cannyLTH = 150;
int cannyHTH = 230;
int HoughTH = 130;
int Rmin = 40, Rmax = 130;
houghCircles(src,dst,cannyLTH,cannyHTH,HoughTH,Rmin,Rmax);
```

FERONE LEZ 9

Segmentazione4_esercizi (otsu.cpp / otsu2k.cpp)

Analizziamo Otsu a singola soglia (una verbosa e una più compatta/efficiente) e Otsu a 2 soglie.

Implementazione dell'algoritmo di Otsu

Applichiamo innanzitutto un poò di blurring Gaussiano così da diminuire il rumore.

Calcoliamo l'istogramma normalizzato attraverso la funzione da noi definita `normalizedHisogram().`

Fatto questo possiamo applicare Otsu. Ricordiamo che lo scopo di Otsu è fondamentalmente di **trovare una soglia**.

- **Istogramma normalizzato**

Un istogramma normalizzato consiste nel calcolare l'istogramma (ovvero il numero dei pixel che assumono un particolare livello di intensità) e dividerlo per il numero di pixel (numRighe*numColonne).

Il nostro istogramma è rappresentato dal vector `his`, composto da 256 elementi float inizializzati a 0.

Scorriamo sull'intera matrice con un doppio ciclo for: consideriamo ad ogni iterazione un pixel diverso dell'immagine il quale avrà un valore di intensità compreso fra 0 e 255. Useremo questi valori di intensità come *indici* di `his`, così quando compaiono incrementeremo di 1 la posizione di `his` corrispondente (es. se nell'immagine ho 200 pixel la cui intensità è 50, in `his[50]` avrò valore 200).

Poi dividamo `his[i]` per il numero di pixel, per ogni i.

```
vector<double> normalizedHistogram(Mat img){
    vector<double> his(256, 0.0f);
    for(int y=0; y<img.rows; y++)
        for(int x=0; x<img.cols; x++)
            his[img.at<uchar>(y,x)]++;
    for(int i=0; i<256; i++)
        his[i]/=img.rows*img.cols;
    return his;
}
```

Otsu con singola soglia

Ricordiamo dagli “Appunti ELIM – Ferone”:

[L’algoritmo di Otsu consiste nei seguenti **passi fondamentali**:

1. calcolare l’istogramma normalizzato dell’immagine;
2. calcolare le somme cumultive $P_1(k)$ 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) $\boldsymbol{p_i = \frac{n_i}{MN}}$,

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

]

1.

Possiamo farlo sfruttando la funzione `normalizedHistogram` calcolata prima.
nb: `his[i]` sarebbe p_i (utile anche per evitare confusione fra p e P (probabilità)).

2.

Calcoliamo le somme cumulative, che ricordiamo essere "la probabilità che un pixel appartenga ad una certa classe".
In questo caso vogliamo conoscere qual è la probabilità che un pixel appartenga alla prima classe, variabile `prob1`.
`prob2` (la prob. che un pixel appartenga alla 2 classe) si ottiene di conseguenza, in quanto sono "tutti i pixel che non appartengono a `prob1`", ovvero `prob2 = 1 - prob1`. (dunque è inutile calcolarlo).

Dunque: al variare della soglia calcoliamo la probabilità che un pixel appartenga alla prima classe.
Essendo somme cumulative, la somma al passo i-simo sarà data considerando la somma al passo precedente.

3.

Calcoliamo la media cumulativa. Se osserviamo la formula, per ogni soglia `i`, non è altro che `his[i]` moltiplicata per `i` (ovvero la soglia corrente).
Essendo anche in questo caso cumulative, la somma al passo i-simo sarà data considerando la somma al passo precedente.
(nb: obv se i=0, i*his[0] = 0, dunque non abbiamo bisogno di calcolare cumMean[0]).

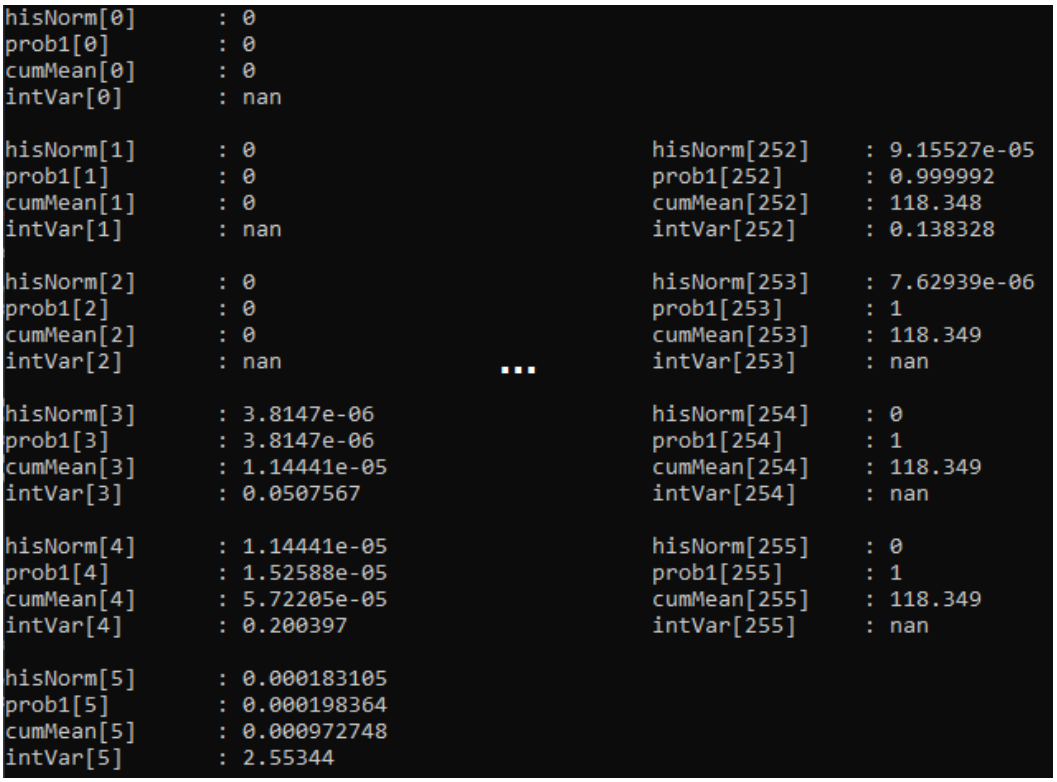
4.

Calcoliamo la media globale. In questo caso si tratta ovviamente di un unico valore.
Applichiamo la formula corrispondente.

5.

Calcoliamo la varianza interclasse (`intVariance`) $\sigma^2_B(k)$. Consideriamo un vettore nel quale calcoliamo la varianza interclasse per ogni soglia i-sima.

ATTENZIONE: in questo caso possono verificarsi delle divisioni per 0! Questo succede se `prob1 = 0` oppure `prob1 = 1`.
Il primo caso può presentarsi perché i primi livelli di intensità potrebbero non comparire mai nell'immagine, e dunque i primi elementi di his sono 0, e dunque anche i primi elementi di prob1 saranno 0.
Il secondo caso può presentarsi perché verso la fine prob1 avrà un valore così vicino ad 1 che potrebbe essere direttamente arrotondato ad 1, e dunque quando facciamo (prob1*(1-prob1)) sarebbe (1*(1-1)) = 0. Dunque divideremo per 0.



Per risolvere basta fare dei controlli preliminari:

- quando `prob1[i] = 0`, poniamo `intVariance[i]=0`.
- se `prob1[i]` non è 0, allora verifichiamo se `prob1[i]=1`. In tal caso, gli assegniamo il “valore massimo più piccolo di 1” (rappresentabile dalla macchina). Dovrebbe essere: `0x3f7fffff`, cioè 0.99999940395355224609.

adesso non compaiono più i `nan`.

6.

Otteniamo **k*** calcolando la **posizione del massimo** del vettore `intVariance`.
Possiamo usare la funzione `max_element(v.begin(), v.end())` per trovare l’iteratore dell’elemento massimo (nb: se vogliamo il valore dobbiamo anteporre l’asterisco a max_element()) insieme alla funzione `distance(`*`inizio, max_element(…)`*`)` per trovare l’indice (**nb**: per questione di template, *inizio* sarà `intVariance.begin()`).
k* sarebbe la threshold che stiamo cercando, dunque la chiamo `thresh`.

Il punto falcolativo non ci interessa.

nb:

Qui in generale usiamo `i` per indicare le soglie, ma in generale si dovrebbe parlare di “profondità *k*”.

Il punto 5 e il punto 6 possono essere accorpati in un unico punto (verifichiamo il massimo ad ogni nuova varianza interclasse calcolata), come vedremo dopo.

CODICE

```
int Otsu(Mat src){

    //1.
    vector<double> his = normalizedHistogram(src);

    //2.
    vector<double> prob1(256,0.0f);
    prob1[0]=his[0];
    for(int i=1; i<256; i++)
        prob1[i] = prob1[i-1]+his[i];

    //3.
    vector<double> cumMean(256,0.0f);
    for(int i=1; i<256; i++)
        cumMean[i] = cumMean[i-1]+i*his[i];

    //4.
    double gCumMean = 0.0f;
    for(int i=0; i<256; i++)
        gCumMean+=i*his[i];

    //5. nb: if 1 is 0x3f800000, then mST1 is 0x3f7fffff, so 0.99999940395355224609
    vector<double> intVariance(256,0.0f);
    for(int i=0; i<256; i++){
        if(prob1[i] == 0.0f){
            intVariance[0]=0.0f;
        } else {
            if(prob1[i] == 1.0f)
                prob1[i] = maxSmallerThan1;
            intVariance[i]=pow(gCumMean*prob1[i]-cumMean[i],2)/(prob1[i]*(1-prob1[i]));
        }
    }

    //6.
    auto maxVariance = max_element(intVariance.begin(), intVariance.end());
    int thresh = distance(intVariance.begin(), maxVariance); //k*


    return thresh;
}
```

Otsu con singola soglia – ottimizzato

In pratica si tratta di applicare gli stessi punti di prima, ma utilizziamo un unico ciclo for (questo perché comunque calcoliamo sempre tutti i valori in un intervallo [0,256[ ). Questo non solo ci risparmia di utilizzare più cicli for, ma ci permette di evitare di mantenere in memoria diversi vector: prob1, cumMean, intVariance.

Dobbiamo ricordare che il nostro scopo è trovare quel valore *k* (k*) che massima intVariance, dunque ci basta calcolare ad ogni passo i correnti prob1 e cumMean, da cui otteniamo il corrente intVariance, e verifichiamo se l'intVariance corrente è MAGGIORE della varianza massima (maxVariance). Se così fosse, aggiorniamo il massimo e l'**indice del massimo** (che corrisponde a k*, ovvero alla threshold che stiamo cercando),

Siccome per tutti i calcoli ad ogni passo abbiamo bisogno della media cumulativa globale, la calcoleremo all'inizio.

```
int OtsuImp(Mat src){

    vector<double> his = normalizedHistogram(src);

    double gCumMean=0.0f;
    for(int i=0; i<256; i++)
        gCumMean+=i*his[i];

    double currProb1=0.0f;
    double currCumMean=0.0f;
    double currIntVariance=0.0f;
    double maxVariance=0.0f;
    int thresh=0;
    for(int i=0; i<256; i++){
        currProb1+=his[i];
        currCumMean+=i*his[i];
        currIntVariance=pow(gCumMean*currProb1-currCumMean,2)/(currProb1*(1-currProb1));
        if(currIntVariance > maxVariance){
            maxVariance = currIntVariance;
            thresh = i;
        }
    }

    return thresh;
```

Otsu con soglie multiple (k=2)

Questa volta vogliamo far variare due soglie (dunque *k = 2*) che individueranno 3 classi, cioè 3 intervalli di intensità.

La media cumulativa è calcolata sempre allo stesso modo.

A differenza di prima, non possiamo più tenere in considerazione una sola probabilità `prob1` (da cui poi derivavamo `prob2`), ma dobbiamo **calcolare la probabilità** ad ogni passo **di ogni classe**.
Discorso simile anche per quanto riguarda le **medie cumulative**.
Dunque avremo:

- un vettore `currProb` di 3 elementi: prob[0], prob[1] e prob[2]:
- un vettore `currCumMean` di 3 elementi: …

nb: anche per semplicità facciamo direttamente riferimento alla versione improved (e non all'originale), altrimenti non avremmo dei vettori ma delle matrici.

Per calcolare `currProb` e `currCumMean` si utilizza un triplo ciclo for innestato, dove consideriamo prima il livello `i`, poi il livello fra `i` e `j`, e infine il livello fra `j` e `k`.

Avendo a disposizione le probabilità e le medie cumulative di ogni intervallo per la *soglia corrente*, possiamo calcolare la **varianza interclasse**: si tratta banalmente della somma delle probabilità, moltiplicate per la corrispondente media cumulativa meno la media globale al quadrato.

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

(nel codice usiamo un ciclo for di 3 iterazioni che fa una somma per volta, questo per non far uscire un'istruzione enorme).

Calcolato la *varianza interclasse corrente* `currIntVariance`, verifichiamo se è quella massima: in tal caso aggiorniamo il massimo e l'indice del massimo (il nostro threshold), che in questo caso sarà `i` per la prima soglia e `j` per la seconda.

Notare che alla fine del ciclo for che calcola `currProb[2]` e `currCumMean[2]`, ovvero quella che considera la soglia fra il livello j e k, li resettiamo per prepararli alle prossime iterazioni.

Vale la stessa cosa anche per `currProb[1]` e `currMean[1]`.

Alla fine, ritorneremo un vettore di due interi che contiene le due soglie (k^*_1 e k^*_2).

CODICE

```
vector<int> Otsu2Imp(Mat& src){

    vector<double> his = normalizedHistogram(src);

    double gMean = 0.0f;
    for(int i=0; i<256; i++)
        gMean += i*his[i];

    vector<double> currProb(3,0.0f);
    vector<double> currCumMean(3,0.0f);
    double currIntVar = 0.0f;
    double maxVar = 0.0f;
    vector<int> kstar(2,0);
    for(int i=0; i<256-2; i++){
        currProb[0] += his[i];
        currCumMean[0] += i*his[i];
        for(int j=i+1; j<256-1; j++){
            currProb[1] += his[j];
            currCumMean[1] += j*his[j];
            for(int k=j+1; k<256; k++){
                currProb[2] += his[k];
                currCumMean[2] += k*his[k];
                currIntVar = 0.0f;
                for(int w=0; w<3; w++)
                    currIntVar += currProb[w]*pow(currCumMean[w]/currProb[w]-gMean,2);
                if(currIntVar > maxVar){
                    maxVar = currIntVar;
                    kstar[0] = i;
                    kstar[1] = j;
                }
            }
            currProb[2] = currCumMean[2] = 0.0f;
        }
        currProb[1] = currCumMean[1] = 0.0f;
    }
    return kstar;
}
```

nb: in realà la formula dovrebbe essere:

```
for(int w=0; w<3; w++)
      currIntVar += currProb[w]*pow(currCumMean[w] - gMean,2);
```

ma il prof utilizza quella implementazione, ed in effetti sembra dare risultati migliori.

Per visualizzare il risultato quando usiamo Otsu con una sola soglia possiamo semplicemente far affidamento sulla funzione di OpenCV `threshold`.

Tuttavia se usiamo Otsu per soglie multiple, per visualizzare il risultato non possiamo usare Threshold ma “dobbiamo fare le cose a mano”. Banalmente, se abbiamo 3 classi (dunque due soglie), andiamo a posizionare uno di 3 valori di intensità (es. 0, 127, 255) al pixel di output in base a quale classe appartenga il pixel considerato.

Questa funzione prende il nome di `multipleThreshold()`.
Si crea una matrice di tutti zero delle stesse dimensioni e tipo della matrice. Se il valore di intensità del pixel è maggiore della seconda soglia allora il pixel di output è posto a 255 (classe 3), se invece è maggiore della prima soglia allora il pixel di output è posto a 127 (classe 2). Altrimenti, il valore del pixel rimane a 0.

- **multipleThreshold**

```
void multipleThreshold(Mat src, Mat& dst, vector<int> thresh){
    dst = Mat::zeros(src.size(), src.type());
    for(int y=0; y<src.rows; y++)
        for(int x=0; x<src.cols; x++)
            if(src.at<uchar>(y,x) >= thresh[1])
                dst.at<uchar>(y,x) = 255;
            else if(src.at<uchar>(y,x) >= thresh[0])
                dst.at<uchar>(y,x) = 127;
}
```

Nel main:

```
Mat src = imread(argv[1],IMREAD_GRAYSCALE);
if(src.empty()) return -1;

GaussianBlur(src, src, Size(3,3), 0);

Mat otsuImg, otsuImpImg, otsu2ImpImg;
threshold(src,otsuImg,Otsu(src),255,THRESH_BINARY);
threshold(src,otsuImpImg,OtsuImp(src),255,THRESH_BINARY);
multipleThreshold(src,otsu2ImpImg,Otsu2Imp(src));
```

FERONE LEZ 10

Segmentazione5_esercizi (regionGrowing.cpp / splitAndMerge.cpp)

Implementazione algoritmo Region Growing

L'algoritmo di Region Growing può essere nei seguenti **passi**:

1. formare l'immagine f_Q (`mask`) 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…).

I punti **1**, **2** e **3** sono relativamente semplici, e si svolgono tutti all'interno dello stesso doppio ciclo for.
Solo il punto 2 dovrebbe corrispondere circa all'esecuzione della funzione *grow()*.

Azioni preliminari…

Dobbiamo definire l'area più piccola che consideriamo come regione. Utilizziamo un "fattore" che dipende dalle dimensioni generali dell'immagine (es. 0.01*src.rows*src.cols, significa che le regioni più piccole dell'1% delle dim dell'immagine vengono scartate).
Le regioni più piccoli verranno inserite in un'unica regione detta "don't care".

`Mask` è un'immagine di appoggio che usiamo per segnarci tutti i pixel che appartengono ad una data regione. Man mano che l'algoritmo procede, si parte da un seed (che nella maschera verrà messo ad 1) e man mano che i pixel vengono aggiunti a questa regione verranno impostate le posizioni corrispondenti ad 1 all'interno della maschera.
In questo modo, alla fine di un'iterazione di accrescimento all'interno della maschera avrò tutti i pixel di quella regione ad 1 (e dunque potrò farò diverse operazioni, come assegnargli un'etichetta o estrarre quella porzione dall'immagine originale).

`Masks` è un vettore di Mat, il quale conterrà tutte le maschere. Moltiplicheremo per 255 per motivi di visualizzazione.

All'interno del doppio ciclo for…

In questo caso non abbiamo informazioni sui semi, dunque analizzeremo tutta l'immagine, e considero come seed il primo pixel in `dst` a cui non è stato ancora assegnato un valore (è ancora 0). A partire dal pixel (x,y) effettuo la fase di accrescimento attraverso la funzione `grow()`.

Al termine della funzione, `mask` avrà tutti i pixel che appartengono alla regione.
Poi calcolo l'area occupata da questa regione facendo una somma di tutti i pixel ad 1 dell'immagine. Se questa regione è abbastanza grande rispetto all'area minima, allora vado a sommare in `dst` la mask più il padding. Altrimenti imposto in `dst` l'etichetta a 255 (che sarebbe la regione don't care).

Il `padding` è la label che assegnamo alla regione. Questo parte da 1 e lo incrementiamo per ogni nuova regione (dunque la prima regione sarà 1, la seconda sarà 2, ecc.).

Posso anche definire un numero massimo di regioni, per cui se supero tale limite significa che sto over-segmentando.

Poi riporto la maschera a 0 così da prepararla per il prossimo seed.

Infine…
ritorneremo il vettore di Mat `masks`.

CODICE

```
vector<Mat> regionGrowing(Mat& src, float minRegionAreaFactor,
                          int maxRegionNum, int th){

    int minRegionArea = int(minRegionAreaFactor*src.rows*src.cols);
    uchar label = 1;
    Mat dst = Mat::zeros(src.rows, src.cols, CV_8UC1);
    Mat mask = Mat::zeros(src.rows, src.cols, CV_8UC1);
    int maskArea;
    vector<Mat> masks;

    for(int x=0; x<src.cols; x++){
        for(int y=0; y<src.rows; y++){
            if(dst.at<uchar>(Point(x,y)) == 0){
                grow(src,dst,mask,Point(x,y),th);
                maskArea = (int)sum(mask).val[0];
                if(maskArea > minRegionArea){
                    dst += mask*label;
                    label++;
                    masks.push_back(mask.clone()*255);
                    if(label > maxRegionNum) exit(-2); //oversegmentation
                } else {
                    dst += mask*255; //don't care area
                }
                mask -= mask;
            }
        }
    }
    return masks;
}
```

- **Grow()**

PointShift2D definisce l'8-intorno. Serviverà per analizzare i pixel adiacenti.

La fase di accrescimento utilizza una struttura dati di appoggio, lo stack: conterrà tutti i pixel che verranno man mano scoperti. All'inizio avrò solo il seed, dunque lo inserisco nello stack.

Finché lo stack non è vuoto, prelevo il pixel in cima allo stack e lo chiamo `center`. Imposto questo pixel ad 1 all'interno della maschera. Poi rimuovo il pixel dallo stack.

Poi analizzo l'8-intorno di `center`: si tratta di un ciclo for di 8 iterazioni (una per ogni scostamento), dove aggiungo al pixel `center` un certo scostamento in una delle 8 direzioni (dunque controllo i vicini).
Poi controllo innanzitutto che il pixel stimato sia interno all'immagine, altrimenti non ha senso effettuare questa verifica e passo direttamente alla successiva iterazione.

Se il pixel stimato è all'intero dell'immagine, mi calcolo la distanza euclidea (differenza delle componenti al quadrato) fra il seed e il pixel che sto stimando.

Poi verifico alcune condizioni:

- se dest(punto_stimato) è 0, perché significa che non è ancora stato etichettato;
- se mask è 0, perché significa che non è stato ancora inserito all'interno della regione;
- se la distanza è inferiore alla soglia prefissata (`delta<threshold`).

Se queste 3 condizioni sono rispettati allora inserisco il pixel all'interno della regione (dunque pongo il pixel all'interno della regione) e poi faccio il push sullo stack in modo che in una delle prossime iterazioni considereremo anche l'8 intorno di questo pixel per accrescere la regione.

Nb: in un'immagine RGB le componenti sono ovviamenti 3, dunque dovremo effettuare la somma della "differenza delle 3 componenti al quadrato".

Quando lo stack è vuoto significa che non ho più pixel da esplorare, e dunque la regione non può essere ulteriormente accresciuta (termino la fase di accrescimento).
All'interno di `mask` avrò tutti i pixel che appartengono alla regione.

CODICE

```
void grow(Mat& src, Mat &dst, Mat& mask, Point seed, int th){

    const Point pointShift2D[8] = //8intorno
    {
        Point(-1,-1), Point(-1,0), Point(-1,1),
        Point(0,-1), Point(0,1),
        Point(1,-1), Point(1,0), Point(1,1)
    };

    stack<Point> point_stack;
    point_stack.push(seed);

    Point ePoint; //estimated Point
    Point center;
    while(!point_stack.empty()){
        center = point_stack.top();
        mask.at<uchar>(center) = 1;
        point_stack.pop();

        for(int i=0; i<8; i++){
            ePoint = center+pointShift2D[i];
            if( ePoint.x < 0 || ePoint.x > src.cols-1 ||
                ePoint.y < 0 || ePoint.y > src.rows-1){
                    continue;
            } else {
                int delta = int(pow(src.at<cv::Vec3b>(center)[0] - src.at<cv::Vec3b>(ePoint)[0], 2)
                + pow(src.at<cv::Vec3b>(center)[1] - src.at<cv::Vec3b>(ePoint)[1], 2)
                + pow(src.at<cv::Vec3b>(center)[2] - src.at<cv::Vec3b>(ePoint)[2], 2));
                if(dst.at<uchar>(ePoint) == 0
                   && mask.at<uchar>(ePoint) == 0
                   && delta < th){
                    mask.at<uchar>(ePoint) = 1;
                    point_stack.push(ePoint);
                }
            }
        }
    }

}
```

nb:

Attenzione alla soglia. Per questo programma una soglia ideale è di circa 40 per le immagini a colori e 15 per le immagini in scala di grigi.

Implementazione algoritmo Split and Merge

L'algoritmo di Split and Merge può essere nei seguenti **passi**:

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).

Split and Merge

`th` è la threshold sulla deviazione standard.
`tsize` è la dimensione minima della regione.
Entrambe sono **variabili globali**.

L'immagine deve essere QUADRATA (questo perché creeremo delle sottoregioni quadrate), dunque dovremo cropparla (es. nel main()).
Effettuo del blurring sull'immagine così da rimuovere un po' di rumore.

Effettuo lo split, e poi il merge.

Infine effettuo la segmentazione su un *clone* sell'immagine originale, `segmented`.

*56

```
1  void splitAndMerge(Mat src, Mat& segmented, float t_size, int threshold){
2
3      tsize = t_size;
4      th = threshold;
5
6      int exponent = log(min(src.rows,src.cols))/log(2);
7      int s = pow(2.0, double(exponent));
8      Rect square = Rect(0,0,s,s);
9      src = src(square).clone();
10     GaussianBlur(src,src,Size(3,3),0,0);
11
12     TNode* root = split(src,Rect(0,0,src.rows,src.cols));
13     merge(root);
14
15     segmented = src.clone();
16     segment(root,segmented);
17 }
```

Classe TNode

La **classe TNode** rappresenta un nodo del Quadtree.
L'oggetto Rect region rappresenta la regione dell'immagine coperta da questa regione. In OpenCV un oggetti Rect è definito da un punto (x,y), che rappresenta il pixel in alto a sx, un'ampiezza e un'altezza.
I 4 figli del nodo sono UL, UR, LL, LR (Upper Left, Upper Right, Lower Left e Lower Right).

Ho una vettore di puntatori a Tnode `merged` che rappresenta le regioni che devono essere unite (queste regioni saranno i figli).

MergedB è un vector di bool che mi dice se due regioni figlie sono state unite.

Poi ho il costruttore (imposto la regione e metto a null i puntatori a nodi).

Per determinare il predicato dovrò calcolare la *deviazione standard* stddev e la *media* della regione. In particolare lavoro sulla deviazione standard mentre userò la media per assegnare un'etichetta alla regione finale.

Per aggiungere una regione userò addRegion.

Il resto sono semplicemente getter e setter.

Nb: userò una versione meno sofisticata, che non fa uso di setter e getter così da avere una classe TNode molto più compatta.

Creiamo la matrice attraverso *split*. Effettuo il merge delle regioni adiacenti attraverso il *merge*.

DrawMerged serve solo per visualizzare il quadTree con le regioni unite.

Segment fa vedere la segmentazione, dove in ogni regione individuata viene inserito il valore medio dei pixel contenuti nella regione.

```
1  class TNode{
2  public:
3      Rect region;
4      TNode *UL, *UR, *LL, *LR;
5      vector<TNode*> merged;
6      vector<bool> mergedB = vector<bool>(4,false);
7      double stddev, mean;
8
9      TNode(Rect R){ region=R; UL=UR=LL=LR=nullptr; }
10     void addRegion(TNode *R){ merged.push_back(R); }
11     void setMergedB(int i){ mergedB[i] = true; }
12 };
```

Split

Split si tratta di una funzione ricorsiva: se entro nella funzione split significa che ho una nuova regione e quindi alloco un nuovo Tnode. R è la regione dell'immagine rappresentata dal nodo. Ovviamente, la **prima volta** (dunque nel main) R parte da (0,0) e l'altezza e l'ampiezza dell'immagine sono il numero di righe e di colonne dell'immagine, dunque vado a prendere tutta l'immagine.

Poi mi calcolo media e deviazione standard all'interno della regione attraverso la funzione di OpenCV `meanStdDev`. Passiamo src(R), dove R indica una "porzione" dell'immagine, e due variabili dove sarà contenuta la media e la dev. standard. Ovviamente la prima volta passeremo tutta la regione.
Una volta calcolati, li vado ad impostare negli attributi corrispondenti di root.

Poi prima di iniziare la fase di splitting ricorsiva, dobbiamo controllare se la regione > tsize, dunque se quella regione può essere ulteriormente splittata, e se la deviazione standard è maggiore di un determinato threshold.
In tal caso, effettuo lo splitting. Avrò 4 regioni tutte che avranno la metà dell'ampiezza e dell'altezza dell'immagine originale, dobbiamo stare solo attenti all'offset.

Per ogni regione calcolata, per il nodo corrispondete dovrò richiamare ricorsivamente la funzione split a cui passerò l'immagine originale e la regione calcolata. Ricordiamo che split ritornerà la radice dell'albero quadtree, che noi assegneremo al nodo.

Ovviamente la fase di split terminerà quando una delle due condizioni diventa falsa, dunque se la regione considerata è troppo piccola oppure la deviazione standard è sotto la soglia (dunque ho una regione piuttosto uniforme come livelli di grigio).

Inoltre, visualizzo il contorno del rettangolo sull'immagine (rectangle(img,R,Scalar(0)), dunque in output avremo il Quadtree.

Infine, ritorno la radice.

Finita la fase di split, passo a quella di merge.

```
TNode* split(Mat& src, Rect R){

    TNode* root = new TNode(R);

    Scalar mean, stddev;
    meanStdDev(src(R),mean,stddev);
    root->mean = mean[0];
    root->stddev = stddev[0];

    if(R.width > tsize && root->stddev > th){

        Rect ul(R.x,R.y,R.height/2,R.width/2);
        root->UL=split(src,ul);

        Rect ur(R.x,R.y+R.width/2,R.height/2,R.width/2);
        root->UR=split(src,ur);

        Rect ll(R.x+R.height/2,R.y,R.height/2,R.width/2);
        root->LL=split(src,ll);

        Rect lr(R.x+R.height/2,R.y+R.width/2,R.height/2,R.width/2);
        root->LR=split(src,lr);
    }

    rectangle(src,R,Scalar(0));
    return root;
}
```

Merge

Nella fase di merge, parto dalla radice e verifico, regione per regione, se posso unire regioni adiacenti. In questo caso sto considerando solo regioni adiacenti che appartengono alla stessa regione (e dunque non considero le sottoregioni che si trovano in altre regioni).

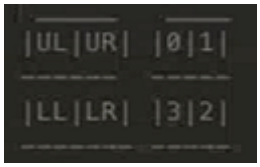
Prima di analizzare la regione verifico se l'ampiezza è maggiore di `tsize` e se la deviazione standard è maggiore della threshold: in tal caso, significa che la regione è stata splittata, e quindi ho delle sottoregioni che potrei fondere.
Altrimenti, significa che la regione non è stata splittata, dunque inserisco all'interno dell'array *merged* della regione l'intera regione stessa, in quanto deve essere considerata integralmente.
Come flag per indicare che tutta la regione deve essere unita imposto a true tutti i valori booleani dell'array MergedB.

Se la regione è stata splittata, verifico se posso fare delle unioni: il procedimento è analogo per ogni if.
Nel primo if provo ad unire Upper Left e Upper Righ. Per verificare se queste due regioni possono essere unite dobbiamo prendere in considerazione la deviazione standard `stddev` della regione e verificare se entrambe sono minori di una certa threshold.
Se così fosse, inserisco ambo le regioni nell'array `merged` e imposto ambo i corrispettivi booleani di `mergedB` a true.
Avendo unito queste posso anche verificare se posso unire Lower Left e Lower Right.
Se non posso, richiamo ricorsivamente il merge su di loro.

Notare che non posso effettuare un'unione fra tutte le regione (in questo caso solo UL-UR e LL-LR), questo perché se unissi a loro volta anche UL-UR e LL-LR andrei a violare il predicato (la regione sarebbe completamente unita in quanto è "abbastanza uniforme", ma questo non è possibile perché altrimenti non l'avremmo splittata a priori).

Ovviamente posso effettuare un'ulteriore miglioria e permette un'unione fra UL-UR con solo LL oppure LR, così da ottenere dettagli più fini.

Gli altri if fanno praticamente la stessa cosa, ma tenendo conto di altre possibilità di unione. Se non riesco ad effettuare alcuna unione, significa che devo tentare di unire a livelli successivi, e quindi richiamo merge sulle 4 sottoregioni.



```
void merge(TNode *root){

    if(root->region.width > tsize && root->stddev > th){
        if(root->UL->stddev <= th && root->UR->stddev <= th){ //UL-UR
            root->addRegion(root->UL); root->setMergedB(0);
            root->addRegion(root->UR); root->setMergedB(1);
            if(root->LL->stddev <= th && root->LR->stddev <= th){
                root->addRegion(root->LL); root->setMergedB(3);
                root->addRegion(root->LR); root->setMergedB(2);
            } else {
                merge(root->LL);
                merge(root->LR);
            }
        } else if(root->UR->stddev <= th && root->LR->stddev <= th){ //UR-LR
            root->addRegion(root->UR); root->setMergedB(1);
            root->addRegion(root->LR); root->setMergedB(2);
            if(root->UL->stddev <= th && root->LL->stddev <= th){
                root->addRegion(root->UL); root->setMergedB(0);
                root->addRegion(root->LL); root->setMergedB(3);
            } else {
                merge(root->UL);
                merge(root->LL);
            }
        } else if(root->LL->stddev <= th && root->LR->stddev <= th){ //LL-LR
            root->addRegion(root->LL); root->setMergedB(3);
            root->addRegion(root->LR); root->setMergedB(2);
            if(root->UL->stddev <= th && root->UR->stddev <= th){
                root->addRegion(root->UL); root->setMergedB(0);
                root->addRegion(root->UR); root->setMergedB(1);
            } else {
                merge(root->UL);
                merge(root->UR);
            }
        } else if(root->UL->stddev <= th && root->LL->stddev <= th){ //UL-LL
            root->addRegion(root->UL); root->setMergedB(0);
            root->addRegion(root->LL); root->setMergedB(3);
            if(root->UR->stddev <= th && root->LR->stddev <= th){
                root->addRegion(root->UR); root->setMergedB(1);
                root->addRegion(root->LR); root->setMergedB(2);
            } else {
                merge(root->UR);
                merge(root->LR);
            }
        } else {
            merge(root->UL);
            merge(root->UR);
            merge(root->LL);
            merge(root->LR);
        }
    } else {
        root->addRegion(root); for(int i=0; i<4; i++) root->setMergedB(i);
    }

}
```

Segmentation

La segmentazione inserisce dei valori fra le regioni che sono state unite. Anche in questo caso effettueremo delle chiamate ricorsive.

All'inizio recupererò l'array `merged`.

- Se l'array non contiene nulla significa che quella regione è stata splittata in 4, e dunque richiamo segment sulle 4 sottoregioni.
  Altrimenti, nell'array merged potrò avere 1 valore o >1 valori.
  All'interno vado a metterci la media dei pixel contenuti all'interno della regione, o per motivi di efficienza faccio direttamente la "media delle medie".
  Poi assegno questa approssimazione del valore medio `val` in tutte le regioni che sono state inserite in `tmp`.
- Se è 1, significa che ho inserito tutta la regione. In pratica si tratta di una foglia del nostro quadtree. Non faccio nessun altra operazione.
- Se è >1, significa che ho inserito più di una regione. Andrò dunque a verificare quali di queste non sono state unite, cioè vado a vedere in ogni posizione di `mergedB` (con cui mi segno proprio quali regioni ho unito) se il valore è false, perché in tal caso significa che la regione non è stata segmentata e dunque richiama ricorsivamente segment su quella regione.

DrawMerged

È simile a segmentation, solo che invece si inserire dei valori disegna dei rettangoli sulla regione. Anche in questo caso distinguiamo cosa fare in base alla dimensione dell'array `merged`:

- Se l'array non contiene nulla, significa che la regione è stata splittata in 4, e dunque richiamo drawMerged sulle 4 sottoregioni.
- Se è 1, significa che ho un'unica regione e dunque disegno un unico rettangolo;
- Se è > 1, effettuo dei controlli per vedere quali regioni posso unire.

```
void segment(TNode *root, Mat& src){

    vector<TNode*> tmp = root->merged;

    if(tmp.size() == 0){
        segment(root->UL, src);
        segment(root->UR, src);
        segment(root->LR, src);
        segment(root->LL, src);
    } else {
        double val=0;       //calc means
        for(auto x:tmp)
            val+=(int)x->mean;
        val/=tmp.size();

        for(auto x:tmp)     //assign mean value to regions
            src(x->region) = (int)val;

        if(tmp.size() > 1){
            if(!root->mergedB[0])
                segment(root->UL,src);
            if(!root->mergedB[1])
                segment(root->UR,src);
            if(!root->mergedB[2])
                segment(root->LR,src);
            if(!root->mergedB[3])
                segment(root->LL,src);
        }

    }
}
```

*

nb: le righe:

```
int exponent = log(min(src.rows,src.cols))/log(2);

int s = pow(2.0, double(exponent));
```

sono importanti perché non solo rendono l'immagine quadrata, ma assicurano anche che le dimensioni dell'immagine siano potenza di 2 (es. 256, 512, 1024, ecc).
Se non applicassimo questa "finezza", nel risultato finale potrebbero comparire degli artefatti.

FERONE LEZ 11

Segmentazione6_esercizi (kmeans.cpp)

Ricordiamo dagli “Appunti ELIM – Ferone”:

[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.

Ripetere i punti 2 e 3 finché il centro (media) di ogni cluster non viene più modificata (o comunque è minima). **]**

4. Assegnare ad ogni pixel del cluster i-simo il valore di c[i].

Dunque:

1.

Otterrò i centri selezionando a caso i valori di intensità presenti all’interno dell’immagine. Genererò per ogni centro un indice di riga e colonna a caso, e gli assegnerò il valore di intensità dell’elemento corrispondente della matrice.

> **Sezione 2/3**: si tratta di un ciclo while nel quale calcoliamo le medie ed aggiorniamo i centri. Se questi non sono variati rispetti alla medie precedenti usciamo dal ciclo. I pixel saranno contenuti nei rispettivi cluster, dove il cluster i-simo è quello rappresentato dal centro i-simo. All’inizio del ciclo ripuliamo i cluster (li riutilizziamo), in quanto ci interessano solo quelli finali, e le *vecchie medie* assumono il valore delle *nuove medie* dell’iterazione precedente.
>
> Inoltre, per evitare di rimanere intrappolati in un minimo locale, fissiamo un numero massimo di iterazioni (tipo 50);

2.

Scorro sulla matrice con un doppio ciclo for. Calcolo la distanza di ogni pixel da ognuno dei *k* centri e li conservo nel vector `dist`. Calcolo l’indice del minimo di queste *k* distanze attraverso la funzione predefinita (senza -dist.begin() alla fine restituisce solo l’elem min):

```
min_element(dist.begin(),dist.end())-dist.begin();
```

assegno le coordinate del pixel al cluster il quale centroide è più vicino.

3.

Completati i raggruppamenti, aggiorno i centri.
Calcolo la *nuova media* per ogni cluster. Recupererò i pixel della matrice che appartengono al cluster i-simo attraverso gli indici conservati nel cluster stesso.

Infine, controllo se le medie sono variate. Il modo migliore per confrontare due valori floating point è vedere se sono "abbastanza simili", per questo utilizziamo epsilon. Insomma, verifichiamo se variano poco.

Verifichiamo se NON sono abbastanza simili, in tal caso significa che c'è stata una variazione (dunque `is_c_varied = true`).

Il ciclo termina quando `is_c_varied = false`, ovvero quando le medie non si spostano ulteriormente.

Nb: non è possibile usare direttamente il size del vector, ma dovremo prima effettuare un cast statico ad intero.

4.

Assegno a tutti i pixel della matrice che appartengono ad un certo cluster il valore medio del cluster stesso (ovvero il valore di `c[i]`). Il risultato andrà nella matrice `dst`.

k-means a colori

Di base è lo stesso ragionamento, solo che in questo caso non dovremo considerare degli uchar ma dei Vec3b.

La cosa principale a cui stare attenti è che dobbiamo considerare tutti e 3 canali della matrice separatamente. Dunque dovremo creare un vector di Mat `srcChannels(3)` il quale conterrà i canali. L' "assegnazione" avverrà utilizzando l'istruzione `split(src, srcChannels)`.

Es. main

```
Mat greySrc,dst,dstColor;
cvtColor(src, greySrc, COLOR_BGR2GRAY);

kmeans(greySrc,dst,2);
kmeansRGB(src,dstColor,3);

imshow("src",src);
imshow("kmeans",dst);
imshow("kmeansRGB",dstColor);
```

```
void kmeans(Mat& src, Mat& dst, int k){

    srand(time(NULL));

    //1.
    vector<uchar> c(k,0);
    for(int i=0; i<k; i++){
        int randRow = rand()%src.rows;
        int randCol = rand()%src.cols;
        c[i] = src.at<uchar>(randRow,randCol);
    }

    //2 and 3
    double epsilon = 0.01f;
    bool is_c_varied = true;
    int maxIterations = 50;
    int it;
    vector<double> oldmean(k,0.0f);
    vector<double> newmean(k,0.0f);
    vector<vector<Point>> cluster(k);

    vector<uchar> dist(k,0);
    int minDistIdx;

    while(is_c_varied && it++ < maxIterations){

        is_c_varied = false;
        for(int i=0; i<k; i++) cluster[i].clear();
        for(int i=0; i<k; i++) oldmean[i] = newmean[i];

        for(int x=0; x<src.rows; x++){
            for(int y=0; y<src.cols; y++){
                for(int i=0; i<k; i++){
                    dist[i] = abs(c[i] - src.at<uchar>(x,y));
                }
                minDistIdx = min_element(dist.begin(),dist.end())-dist.begin();
                cluster[minDistIdx].push_back(Point(x,y));
            }
        }

        for(int i=0; i<k; i++){
            int csize = static_cast<int>(cluster[i].size());
            for(int j=0; j<csize; j++){
                int cx = cluster[i][j].x;
                int cy = cluster[i][j].y;
                newmean[i] += src.at<uchar>(cx,cy);
            }
            newmean[i] /= csize;
            c[i] = uchar(newmean[i]);
        }

        for(int i=0; i<k; i++)
            if( !(abs(newmean[i]-oldmean[i]) <= epsilon) )
                is_c_varied = true;
    }

    //4.
    dst = src.clone();
    for(int i=0; i<k; i++){
        int csize = static_cast<int>(cluster[i].size());
        for(int j=0; j<csize; j++){
            int cx = cluster[i][j].x;
            int cy = cluster[i][j].y;
            dst.at<uchar>(cx,cy) = c[i];
        }
    }

}
```

RGB version

```
void kmeansRGB(Mat& src, Mat& dst, int k){

    srand(time(NULL));

    //1.
    vector<Vec3b> c(k,0);
    for(int i=0; i<k; i++){
        int randRow = rand()%src.rows;
        int randCol = rand()%src.cols;
        c[i] = src.at<Vec3b>(randRow,randCol);
    }

    //2 and 3
    double epsilon = 0.01f;
    bool is_c_varied = true;
    int maxIterations = 50;
    int it;
    vector<Vec3d> oldmean(k,0.0f);
    vector<Vec3d> newmean(k,0.0f);
    vector<vector<Point>> cluster(k);

    double diffBlue, diffGreen, diffRed;
    vector<uchar> dist(k,0);
    int minDistIdx;

    vector<Mat> srcChannels(3);
    split(src, srcChannels);

```

Nb:

Possiamo fare l'**accesso ai canali** anche 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
```

```
    while(is_c_varied && it++ < maxIterations){

        is_c_varied = false;
        for(int i=0; i<k; i++) cluster[i].clear();
        for(int i=0; i<k; i++) oldmean[i] = newmean[i];

        for(int x=0; x<src.rows; x++){
            for(int y=0; y<src.cols; y++){
                for(int i=0; i<k; i++){
                    diffBlue  = c[i].val[0] - srcChannels[0].at<uchar>(x,y);
                    diffGreen = c[i].val[1] - srcChannels[1].at<uchar>(x,y);
                    diffRed   = c[i].val[2] - srcChannels[2].at<uchar>(x,y);
                    dist[i] = sqrt(pow(diffBlue, 2) + pow(diffGreen,2) + pow(diffRed,2));
                }
                minDistIdx = min_element(dist.begin(),dist.end())-dist.begin();
                cluster[minDistIdx].push_back(Point(x,y));
            }
        }

        for(int i=0; i<k; i++){
            int csize = static_cast<int>(cluster[i].size());
            for(int j=0; j<csize; j++){
                int cx = cluster[i][j].x;
                int cy = cluster[i][j].y;
                newmean[i].val[0] += srcChannels[0].at<uchar>(cx,cy);
                newmean[i].val[1] += srcChannels[1].at<uchar>(cx,cy);
                newmean[i].val[2] += srcChannels[2].at<uchar>(cx,cy);
            }
            newmean[i] /= csize;
            c[i] = newmean[i];
        }

        double val = 0.0f;
        for(int i=0; i<k; i++){
            for(int ch=0; ch<3; ch++)
                val += newmean[i].val[ch]-oldmean[i].val[ch];
            val /= 3;
            if( abs(val) <= epsilon)
                is_c_varied = true;
        }
    }

    //4.
    dst = src.clone();
    for(int i=0; i<k; i++){
        int csize = static_cast<int>(cluster[i].size());
        for(int j=0; j<csize; j++){
            int cx = cluster[i][j].x;
            int cy = cluster[i][j].y;
            dst.at<Vec3b>(cx,cy) = c[i];
        }
    }

}
```