

ALMA MATER STUDIORUM  
Università degli Studi di Bologna

---

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI  
Corso di Laurea in Matematica

Elaborazioni di Immagini  
con la Libreria  
OpenCV

Tesi in Analisi Numerica

Presentata da:  
Pietro Varano

Relatore:  
Chiar.mo Prof.  
Giulio Casciola

III Sessione  
Anno Accademico 2008-2009

---

# Indice

<b>Introduzione</b>	<b>6</b>
<b>1 OpenCV</b>	<b>8</b>
1.1 Informazioni preliminari . . . . .	8
1.1.1 Ambiente di sviluppo . . . . .	9
1.1.2 Convenzioni . . . . .	10
1.2 Organizzazione della libreria . . . . .	11
1.3 Strutture dati . . . . .	11
1.3.1 CvArr . . . . .	12
1.3.2 IplImage . . . . .	12
1.3.3 CvScalar . . . . .	15
1.4 Immagine e spazi colore . . . . .	15
1.5 Accesso ai pixel . . . . .	19
1.5.1 Metodo indiretto . . . . .	19
1.5.2 Metodo diretto . . . . .	20
<b>2 Operazioni sulle immagini</b>	<b>23</b>
2.1 Operazioni logiche . . . . .	23
2.2 Inversione di un'immagine . . . . .	25
2.3 Thresholding . . . . .	27
2.4 Trasformazioni geometriche . . . . .	33
2.4.1 Traslazione . . . . .	35
2.4.2 Trasformazioni di scala . . . . .	39
2.4.3 Rotazione . . . . .	41

---

2.4.4	Distorsione geometrica . . . . .	44
<b>3</b>	<b>Filtri spaziali</b>	<b>46</b>
3.1	Elementi base . . . . .	46
3.2	Filtri di smoothing . . . . .	50
3.3	Filtri derivativi . . . . .	56
3.3.1	Operatore di Sobel . . . . .	56
3.3.2	Operatore Laplaciano . . . . .	66
3.4	Operatore di Canny . . . . .	71
3.5	Trasformata di Hough . . . . .	76
3.5.1	Trasformata di Hough per le linee . . . . .	76
3.5.2	Trasformata di Hough per forme circolari . . . . .	82
<b>4</b>	<b>Morfologia</b>	<b>88</b>
4.1	Operazioni morfologiche . . . . .	88
4.1.1	Erosione . . . . .	89
4.1.2	Dilatazione . . . . .	93
4.2	Operazioni morfologiche complesse . . . . .	96

# Elenco delle figure

1.1	Configurazione dell'ambiente di sviluppo Dev-C++ . . . . .	10
1.2	Organizzazione della libreria . . . . .	12
1.3	Rappresentazione di un'immagine digitale . . . . .	17
1.4	Spazio colore . . . . .	18
2.1	a) Immagine in scala di grigio, b) maschera per isolare la regione di interesse, c) risultato dell'operazione AND. . . . .	25
2.2	Inversione di un'immagine medica. . . . .	26
2.3	Binarizzazione . . . . .	27
2.4	Impronta digitale e relativo Istogramma . . . . .	28
2.5	Immagini binarie ottenute mediante global thresholding . . . . .	29
2.6	Immagine corrotta da una variazione di intensità sinusoidale . . . . .	32
2.7	Risultato ottenuto mediante variable thresholding . . . . .	33
2.8	Trasformazioni geometriche . . . . .	34
2.9	Immagine di $640 \times 480$ pixel . . . . .	37
2.10	Traslazione . . . . .	38
2.11	Ridimensionamento . . . . .	41
2.12	Rotazione antioraria di $15^\circ$ . . . . .	42
2.13	Rotazione antioraria caratterizzata dalla replica dei bordi . . . . .	43
2.14	Distorsione . . . . .	45
3.1	Filtraggio . . . . .	47
3.2	Filtro mediano . . . . .	48
3.3	Immagine corrotta da rumore impulsivo . . . . .	53

---

3.4	Smoothing: a) tramite filtro di media, b) tramite filtro mediano	54
3.5	Immagine affetta da rumore e relativo istogramma . . . . .	55
3.6	Immagine di smoothing e relativo istogramma . . . . .	55
3.7	Risultato della sogliatura globale . . . . .	56
3.8	Modelli di edge ideali (Ramp Edge e Step Edge) e relativi profili di intensità . . . . .	57
3.9	Vettore gradiente . . . . .	57
3.10	Immagine originale di $320 \times 330$ pixel . . . . .	62
3.11	Immagine Sobel_V e Immagine Sobel_H . . . . .	64
3.12	Immagine Sobel_M . . . . .	66
3.13	Immagine originale e di sharpening . . . . .	69
3.14	Immagine originale di $256 \times 256$ pixel . . . . .	70
3.15	Valore assoluto dell'immagine laplaciana e Valori positivi . . .	70
3.16	Intervalli delle normali agli edge in un intorno $3 \times 3$ . . . . .	72
3.17	Immagine originale di $256 \times 256$ pixel . . . . .	73
3.18	Risultato dell'operatore di Canny: $T_l = 30$ e $T_h = 70$ . . . . .	74
3.19	Risultato ottimale dell'operatore di Canny: $T_l = 60$ e $T_h = 120$	74
3.20	Immagine originale di $319 \times 239$ pixel . . . . .	75
3.21	Risultato ottimale dell'operatore di Canny: $T_l = 45$ e $T_h = 150$	75
3.22	Spazio immagine e spazio dei parametri . . . . .	77
3.23	Punti e rette nello spazio immagine $xy$ e nello spazio dei parametri $mc$ . . . . .	77
3.24	Spazio immagine e punti collineari . . . . .	78
3.25	Spazio $\rho\theta$ di Hough . . . . .	78
3.26	Immagine originale affetta da rumore . . . . .	80
3.27	Trasformazione SHT . . . . .	81
3.28	Trasformazione PPHT . . . . .	82
3.29	Rappresentazione dei cerchi nello spazio immagine e nello spazio dei parametri. . . . .	83
3.30	Superficie conica nello spazio dei parametri. . . . .	84
3.31	Smoothing con filtro gaussiano $11 \times 11$ . . . . .	85

---

3.32	<i>HT</i> che individua le forme circolari . . . . .	86
3.33	Smoothing con filtro di media $15 \times 15$ . . . . .	87
3.34	<i>HT</i> che individua alcune forme in modo non corretto . . . . .	87
4.1	Immagine erosa tramite <i>SE</i> quadrato $3 \times 3$ . . . . .	90
4.2	Immagine erosa con 5 iterazioni tramite <i>SE</i> $3 \times 3$ . . . . .	93
4.3	Dilatazione di A tramite B . . . . .	94
4.4	Dilatazione tramite <i>SE</i> $3 \times 3$ quadrato . . . . .	94
4.5	Dilatazione ottenuta con 5 iterazioni tramite SE quadrato . . . . .	95
4.6	Immagine originale contenente un testo con caratteri rotti . . . . .	96
4.7	Immagine riparata tramite dilatazione con SE $3 \times 3$ a croce . . . . .	97
4.8	Gradiente morfologico . . . . .	99
4.9	Immagine in scala di grigio raffigurante le componenti meccaniche di un motore e relativo gradiente morfologico . . . . .	100
4.10	Opening e Closing per la rimozione del rumore impulsivo. . . . .	101
4.11	Opening attraverso un <i>SE</i> circolare di dimensione $11 \times 11$ . . . . .	103
4.12	Estrazione dei contorni . . . . .	103

# Introduzione

Le tecniche di image processing sono impiegate nel miglioramento della qualità delle immagini, nel controllo di prodotti industriali, nell'analisi di immagini satellitari, e più in generale come passo di pre-elaborazione nei procedimenti di computer vision. Nella seguente tesi saranno trattate operazioni come la sogliatura, la riduzione del rumore, il miglioramento della qualità, e l'estrazione di componenti. Presenteremo inoltre specifiche funzioni della libreria OpenCV che eseguono le operazioni esposte. La Open Source Computer Vision Library è una libreria scritta in C e C++, costituita da oltre 500 funzioni utili nel campo dell'image processing e della computer vision. Punti di forza della libreria sono la completezza e il suo essere open source. Il fatto di essere liberamente distribuita garantisce sicurezza al codice e la possibilità di apportare modifiche al codice stesso, assicurandone così una continua evoluzione. La licenza di distribuzione è priva di royalty e ciò consente il suo utilizzo anche in prodotti commerciali a condizione di mantenere le note di copyright. OpenCV costituisce quindi una infrastruttura aperta e gratuita, compatibile con la Intel Image Processing Library (IPL). Quest'ultima è una libreria non open source che esegue solo operazioni di image processing e dalla quale OpenCV ha ereditato originariamente alcune strutture dati. La portabilità della libreria OpenCV è completa, sono infatti disponibili le versioni per i sistemi MS-Windows, Linux, BSD, Unix e MacOSX.

### Organizzazione della tesi

- Il primo capitolo si apre fornendo al lettore le informazioni relative alla reperibilità, alle convenzioni adottate e all'organizzazione della libreria OpenCV. Dopo una sintetica descrizione sulle modalità di configurazione dell'ambiente di sviluppo scelto, introduciamo alcune strutture dati maggiormente utilizzate nelle applicazioni. Sono inoltre introdotti i concetti di immagine e spazi colore, per poi terminare il capitolo con la descrizione di due metodi utili a manipolare i pixel di un'immagine.
- Nel secondo capitolo sono trattate le operazioni spaziali eseguibili sulle immagini, come le operazioni logiche e di inversione, per passare poi alle operazioni di sogliatura ed infine terminare con le trasformazioni geometriche.
- Nel terzo capitolo viene introdotto il concetto di filtro spaziale. Sono trattati di seguito i filtri di smoothing, il filtro derivativo di Sobel e il filtro di Laplace. Gli ultimi due paragrafi sono dedicati rispettivamente all'operatore di Canny e alla trasformata di Hough.
- La morfologia matematica è l'argomento del quarto capitolo. Sono presentate le operazioni morfologiche di base come l'erosione e la dilatazione, per poi passare alle operazioni complesse ottenute come combinazioni delle operazioni base.

# Capitolo 1

## OpenCV

### 1.1 Informazioni preliminari

I siti internet relativi al progetto sono due, il primo facente capo a Intel (<http://intel.com/technology/computing/opencv>), il secondo è il classico hosting presso SourceForge (<http://SourceForge.net/projects/opencvlibray>) dal quale si può procedere al download sia dei pacchetti sorgenti che degli installativi, e dove sono disponibili tutte le versioni finora rilasciate a partire da quella iniziale. La versione utilizzata è la 1.1pre1, disponibile per i sistemi UNIX, Linux, BSD e MacOSX nel pacchetto `opencv-1.1pre1.tar.gz`, e per i sistemi Windows nel pacchetto `OpenCV_1.1pre1.exe`. In entrambi i pacchetti è presente il file di testo `INSTALL` contenente le informazioni relative alla procedura di installazione della libreria, in particolar modo per i sistemi UNIX e suoi derivati che richiedono maggiore accorgimento per tale operazione. Sono necessari infatti pacchetti aggiuntivi come `ffmpeg 0.4.9-pre1` o superiore (<http://ffmpeg.mplayerhq.hu/projects.html>), le librerie `GTK+ 2.x`, il linguaggio `Python 2.5` o superiore, e ulteriori librerie se non presenti nel sistema. Maggiori informazioni in merito si possono trovare presso le `OpenCV Wiki` del sito `SourceForge`.

### 1.1.1 Ambiente di sviluppo

La procedura di installazione, nei sistemi MS-Windows, consiste semplicemente nell'eseguire il file `OpenCV.exe`. Al termine del setup otteniamo l'apposita cartella di installazione `C:\Programmi\OpenCV`. Il passo successivo è quello di scegliere un ambiente di sviluppo, il quale deve essere settato specificando opportunamente i vari files include e le librerie statiche e dinamiche. In un ambiente come Dev-C++ l'operazione si effettua in pochi passi che possiamo così sintetizzare:

- Aprire Dev-C++;
- Andare sul menù strumenti, opzioni di compilazione;
- Nella scheda "Compilatore" scrivere il nome OpenCV come nuovo compilatore, mettere il segno di spunta su "Aggiungi questi comandi alla linea di comando del linker" e scrivere nella casella di testo associata quanto segue `-L"C:\\Programmi\\OpenCV\\lib" -lxcvcore -lcv -lcvaux -lhighgui -lml`
- Nella scheda "Cartelle", sottoscheda "Include C" aggiungere  
`C:\\Programmi\\OpenCV\\cxcore\\include`  
`C:\\Programmi\\OpenCV\\cv\\include`  
`C:\\Programmi\\OpenCV\\otherlibs\\highgui`  
`C:\\Programmi\\OpenCV\\cvaux\\include`  
`C:\\Programmi\\OpenCV\\ml\\include`
- Ripetere quanto fatto sopra nella sottoscheda "Include C++"

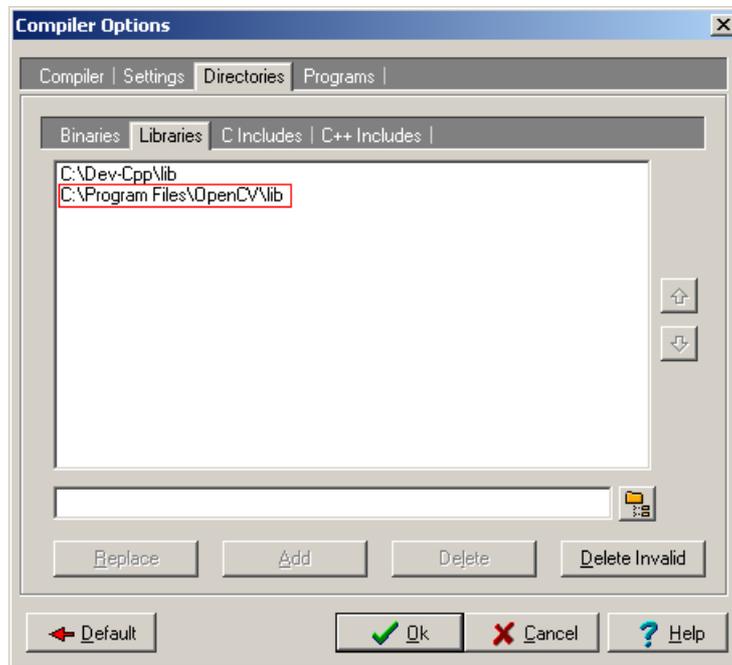


Figura 1.1: Configurazione dell'ambiente di sviluppo Dev-C++

- Nella scheda “Cartelle”, sottoschede “Librerie”, aggiungere  
C:\Programmi\OpenCV\lib  
C:\Programmi\OpenCV\otherlibs\highgui
- Nella scheda “Cartelle”, sottoscheda “File Binari” aggiungere  
C:\Programmi\OpenCV\bin

### 1.1.2 Convenzioni

Tutte le funzioni della libreria sono caratterizzate dal prefisso *cv*, mentre i nomi delle strutture hanno prefisso *Cv* ad eccezione della *IplImage*, la quale è ereditata dalla libreria IPL. Poichè le funzioni operano con immagini e con matrici numeriche, le dimensioni seguono rispettivamente la convenzione larghezza-altezza ( $X, Y$ ) comune nella grafica e l'ordine righe-colonne ( $R, C$ ). In genere i nomi delle funzioni hanno la forma *cv*<ActionName><Object>, dove ActionName rappresenta l'azione principale della funzione, ad esempio

`cvCreate<Object>`, `cvRelease<Object>`, mentre `Object` è l'oggetto su cui avviene l'azione, ad esempio `cvReleaseImage`. Altre funzioni assumono invece il nome dell'algoritmo che implementano, ad esempio `cvCanny`.

## 1.2 Organizzazione della libreria

La libreria è strutturata in cinque sottolibrerie (dette anche moduli) ciascuna con funzionalità specifiche. Il modulo `CXCORE` è quello principale nonché indispensabile. Contiene le strutture dati di base con le rispettive funzioni di inizializzazione, le funzioni matematiche, le funzioni di lettura, scrittura e memorizzazione dati, le funzioni di sistema e di gestione degli errori. Il modulo `CV` contiene le funzioni relative all'immagine processing, le funzioni di analisi strutturale e del moto, di object detection e ricostruzione 3D. `HIGHGUI` contiene le funzioni GUI e quelle di salvataggio e caricamento immagini, nonché le funzioni di acquisizione video e di gestione delle telecamere. Il modulo `ML` (Machine Learning) contiene classi e funzioni relative all'implementazione e gestione di reti neurali, in particolare di tipo multilayer perceptrons (MPL), di classificazione statistica e clustering di dati. Infine vi è il modulo `CVAUX` che contiene sia le funzioni basate su algoritmi ancora in fase di sperimentazione, il cui futuro è quello di migrare nel modulo `CV`, e sia le funzioni considerate obsolete e quindi non più supportate. Le funzionalità di tale modulo sono rivolte alla corrispondenza stereo, al tracking 3D, al riconoscimento degli oggetti (Eigen object).

## 1.3 Strutture dati

Introduciamo alcune delle strutture dati e funzioni base utili per iniziare ad operare con la libreria.

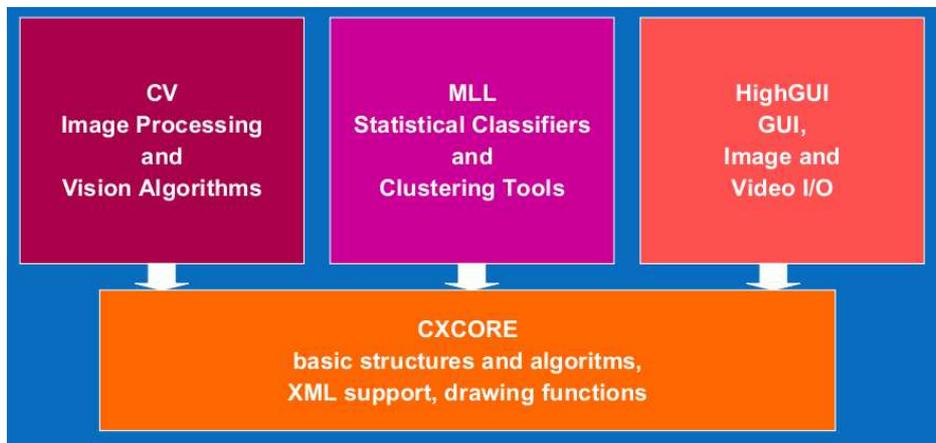


Figura 1.2: Organizzazione della libreria

### 1.3.1 CvArr

La maggior parte delle funzioni eseguono operazioni su strutture dati diverse tra loro, come le immagini (*IplImage*), le matrici (*CvMat*) e le liste dinamiche (*CvSeq*). Le funzioni che eseguono operazioni su questi tipi di dati ammettono sempre un virtuale oggetto array *CvArr*. Ad esempio, nella funzione *cvTranspose(const CvArr\* src, CvArr\* dst)* il parametro *src* può denotare rispettivamente una matrice o un'immagine da trasporre e *dst* la matrice o l'immagine trasposta. La relazione esistente tra le strutture *CvArr*, *cvMat* e *IplImage* è di tipo object-oriented, dove la *IplImage* deriva dalla *CvMat* la quale a sua volta deriva dalla *CvArr*.

### 1.3.2 IplImage

*IplImage* è l'acronimo di Image Processing Library Image, ossia il formato standard utilizzato da Intel per rappresentare un'immagine nelle API IPL e OpenCV. Tramite questa struttura dati è possibile gestire completamente tutto ciò che ruota intorno alle immagini, come caricamento e salvataggio, conversione di formato, elaborazione, filtraggio e visualizzazione. La struttura definisce sia immagini con origine top-left che immagini avente orig-

ine bottom-left, inoltre è costituita dai seguenti campi che nel loro insieme prendono il nome di image Header.

```
typedef struct _IplImage{  
  
    int nSize;  
  
    int ID;  
  
    int nChannels;  
  
    int alphaChannel;  
  
    int depth;  
  
    char colorModel[4];  
  
    char channelSeq[4];  
  
    int dataOrder;  
  
    int origin;  
  
    int align;  
  
    int width;  
  
    int height;  
  
    struct _IplROI * roi;  
  
    struct _IplImage * maskROI;  
  
    void * imageId;  
  
    struct _IplTileInfo * tileInfo;  
  
    int imageSize;  
  
    char * imageData;  
  
    int widthStep;  
  
    int BorderMode[4];  
  
    int BorderConst[4];  
  
};
```

```
char * imageDataOrigin;  
  
}  
  
IplImage;
```

La libreria OpenCV, a differenza della IPL, utilizza solo alcuni campi della struttura. Tra questi abbiamo *nChannels* che indica il numero di canali, e il campo *depth* che indica la profondità in bit. Per quest'ultimo i valori possibili sono: *IPL\_DEPTH\_8U* intero a 8-bit unsigned, *IPL\_DEPTH\_8S* intero a 8-bit con segno, *IPL\_DEPTH\_16S* e *IPL\_DEPTH\_16U* interi a 16-bit con e senza segno, *IPL\_DEPTH\_32S* intero a 32-bit con segno, *IPL\_DEPTH\_32F* floating point a 32-bit e *IPL\_DEPTH\_64F* floating point a 64-bit. Il campo *origin* assume il valore 0 quando l'origine delle coordinate è top-left, viceversa assume il valore 1 quando l'origine è bottom-left. I campi *width* e *height* indicano rispettivamente l'ampiezza e l'altezza in pixel dell'immagine, mentre il campo *struct\_IplROI\* roi* specifica una regione dell'immagine da elaborare (Region Of Interest). Il campo *imageSize* contiene le dimensioni in bytes dell'intera immagine.

La funzione che consente di creare e allocare un'immagine è la seguente

```
cvCreateImage(CvSize size, int depth, int channels);
```

Il parametro *size* permette di assegnare le dimensioni relative all'ampiezza e all'altezza, il parametro *depth* assegna la profondità in bit ed infine *channels* che assegna il numero di canali. Ad esempio, volendo allocare un'immagine (*img*) di dimensione  $640 \times 320$  con profondità a 8-bit con un solo canale, possiamo scrivere

```
CvSize size = cvSize(640, 320);
```

```
IplImage * img = cvCreateImage(size, IPL_DEPTH_8U, 1);
```

La struttura *CvSize* è quindi costituita da due membri di tipo intero utili per definire le dimensioni, espresse in pixel, dell'immagine. La funzione costruttore corrispondente è

$$cvSize(int\ width, int\ height);$$

e l'accesso ai campi si ottiene tramite l'operatore punto (.)

$$size.width = 640;$$
$$size.height = 320;$$

### 1.3.3 CvScalar

Per poter operare con valori scalari si utilizza la struttura dati *CvScalar* dove l'unico membro è un array costituito da quattro elementi di tipo double. La struttura può essere utilizzata non solo per le usuali operazioni aritmetiche, ma anche nelle operazioni di accesso ai singoli pixel di una immagine. In particolare la funzione

$$cvScalar(double\ val0, double\ val1 = 0, double\ val2 = 0, double\ val3 = 0);$$

consente di assegnare da uno a quattro valori ai corrispondenti elementi del membro *val[ ]* mediante una dichiarazione del tipo

$$CvScalar\ scalar = cvScalar(val0, val1, val2, val3);$$

L'accesso a un elemento si ottiene con l'usuale operatore punto

$$scalar.val[i]; \quad i = 0, 1, 2, 3$$

## 1.4 Immagine e spazi colore

Nel precedente paragrafo abbiamo introdotto la struttura che permette di gestire un'immagine. Per semplicità possiamo considerare quest'ultima come una distribuzione bidimensionale di intensità luminosa.

$$f = f(x, y) \tag{1.1}$$

Poichè essa è acquisita tramite sensori, l'uscita è normalmente una grandezza continua le cui variazioni spaziali di ampiezza seguono una legge dipendente dal tipo di sensore. Per poter essere trattata con un computer è necessario convertirla in formato numerico. La creazione dell'immagine digitale a partire dal segnale analogico richiede un processo di campionamento e di quantizzazione. Il campionamento è la discretizzazione dei valori delle coordinate spaziali, mentre la quantizzazione è la discretizzazione dei valori di intensità. Un semplice modello consiste nel rappresentare l'intensità mediante il prodotto di due termini, l'illuminazione  $i(x, y)$  e la riflettanza  $r(x, y)$ . L'immagine è quindi costituita da una componente dovuta alla luce proveniente dalla sorgente di illuminazione e da una componente dovuta alla luce riflessa dagli oggetti presenti sulla scena

$$f(x, y) = i(x, y) \cdot r(x, y) \quad (1.2)$$

con  $0 < f(x, y) < \infty$ ,  $0 < i(x, y) < \infty$ ,  $0 < r(x, y) < 1$ .

Tenuto conto del campionamento spaziale, che rende discreti gli intervalli di variazione di  $x$  e  $y$ , e assumendo che l'immagine continua sia approssimata mediante  $M \times N$  campioni equispaziati lungo le due dimensioni, si ha la sua rappresentazione classica sotto forma di matrice. Un'immagine digitale è quindi una matrice di valori discreti di intensità luminosa costituita da  $M \times N$  pixel ognuno dei quali ha valore compreso nell'intervallo  $[0, 2^k - 1]$ , dove  $k$  è il numero di bit usato per codificarlo. Un pixel è una regione rettangolare coincidente con una cella della griglia di campionamento, e il valore ad esso associato è l'intensità media nella cella. Dal campionamento si determina la risoluzione spaziale, la quale si indica come risoluzione lungo  $x$  e lungo  $y$ , e definisce il numero di pixel da usare. Maggiore è il numero, maggiori sono i particolari visibili. I formati generalmente usati nell'elaborazione di immagini sono: a due livelli e in scala di grigio. Nelle immagini a due livelli (binarie) ciascun pixel può assumere solo il valore 1 (bianco) o il valore 0 (nero). Nelle immagini monocromatiche ciascun elemento della matrice è un valore intero compreso tra 0 e 255, e rappresenta uno dei 256 livelli di grigio. In quelle

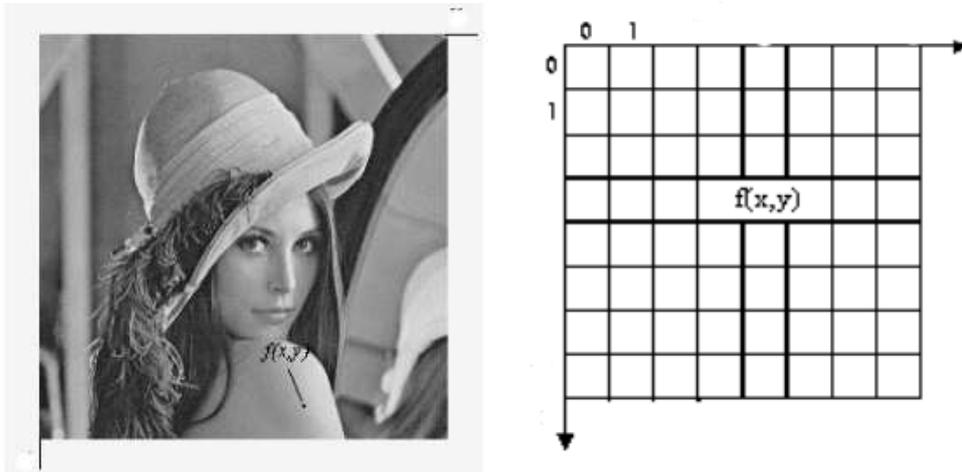


Figura 1.3: Rappresentazione di un'immagine digitale

a colori ogni elemento della matrice corrisponde a una terna ordinata di interi  $(R, G, B)$ , dove  $R, G, B$  indicano rispettivamente le intensità di rosso, verde e blu e assumono valori compresi tra 0 e 255, corrispondenti ai 256 diversi livelli di intensità delle tre sorgenti di colore. In tal caso un'immagine può essere vista come una terna di matrici, dove ognuna rappresenta uno dei colori fondamentali. Lo *spazio colore* è generalmente tridimensionale in modo da poter essere rappresentato da una terna di numeri. Nel sistema visivo umano la retina ha tre tipi di ricettori sensibili al colore, ciascuno dei quali risponde alla radiazione elettromagnetica nel campo del visibile (tra i 360  $nm$  e 800  $nm$ ), tre numeri sono quindi sufficienti a rappresentare la sensazione del colore. I colori fondamentali  $RGB$  sono additivi, ossia i contributi individuali di ciascuno sono sommati insieme per produrre il risultato. La forma dello spazio che contiene al suo interno i colori visibili è un cubo unitario, dove la diagonale principale rappresenta i livelli di grigio. Anche lo spazio colore  $CMY$  viene rappresentato come un cubo unitario allineato con gli assi. Si usano come colori fondamentali i complementari di  $RGB$ : ciano ( $C$ ), magenta ( $M$ ), giallo ( $Y$ ), e costituiscono una sintesi di tipo sottrattivo.

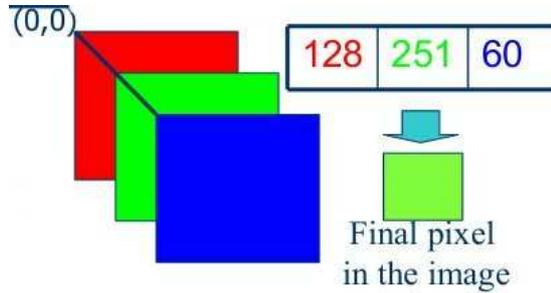


Figura 1.4: Spazio colore

Lo spazio  $CMY$  è utilizzato nei dispositivi di stampa a getto di inchiostro, mentre lo spazio  $RGB$  nei dispositivi di visualizzazione (monitor). La conversione si effettua nel seguente modo

$$C = 1 - R, \quad M = 1 - G, \quad Y = 1 - B; \quad (1.3)$$

$$R = 1 - C, \quad G = 1 - M, \quad B = 1 - Y; \quad (1.4)$$

OpenCV è in grado di gestire numerosi spazi colore, consentendo il passaggio da uno spazio all'altro tramite la funzione

```
void cvCvtColor(const CvArr * src, CvArr * dst, int code);
```

Le due immagini  $src$  e  $dst$  devono essere dello stesso tipo ( $8U$ ,  $16U$  e  $32F$ ) ma possono avere differente numero di canale. Il parametro  $code$  specifica il tipo di conversione che deve essere eseguita utilizzando l'espressione  $CV\_<spazio\_colore\_sorgente>2<spazio\_colore\_destinazione>$ . Ad esempio, con il codice di conversione  $CV\_RGB2GRAY$  la funzione converte un'immagine a colori ( $RGB$ ) in una a toni di grigio. Le intensità dell'immagine a toni di grigio sono calcolate secondo la formula

$$Y = (0.299)R + (0.587)G + (0.114)B \quad (1.5)$$

Viceversa, nella conversione di un'immagine a toni di grigio in una a colori ( $CV\_GRAY2RGB$ ) le componenti sono prese tutte con lo stesso valore

$$R = Y, \quad G = Y, \quad B = Y \quad (1.6)$$

Informazioni esaustive relative agli spazi colore e alle rispettive formule di conversione gestibili in OpenCV si possono trovare in [4].

## 1.5 Accesso ai pixel

Vediamo ora come si possono modificare i valori di intensità dei singoli pixel utilizzando due metodi classificabili come: metodo indiretto e metodo diretto.

### 1.5.1 Metodo indiretto

Questo metodo si basa sull'utilizzo di due funzioni, *cvGet2D()* e *cvSet2D()*. La prima restituisce il valore di uno specifico pixel

```
CvScalar cvGet2D(const CvArr * arr, int idx0, int idx1);
```

Il parametro *arr* è l'immagine da modificare, mentre *idx0* e *idx1* sono le coordinate relative allo specifico pixel. Per un'immagine con un solo canale di dimensioni  $width \times height$ , l'intensità del pixel di coordinate  $(i, j)$  dove  $i \in [0, height - 1]$  e  $j \in [0, width - 1]$ , si ottiene come

```
CvScalar scalar = cvGet2D(img, i, j);
```

```
scalar.val[0];
```

Per un'immagine multicanale si devono invece utilizzare tutti gli elementi del campo *val[ ]*, uno per ogni livello di colore:

```
scalar.val[0];
```

```
scalar.val[1];
```

```
scalar.val[2];
```

La modifica del valore di un pixel si effettua invece tramite la funzione

```
void cvSet2D(CvArr * arr, int idx0, int idx1, CvScalar value);
```

I parametri sono gli stessi di `cvGet2D()` con l'aggiunta di `value` il quale rappresenta il nuovo valore da assegnare al pixel. In definitiva, per assegnare uno specifico valore al pixel di coordinate  $(i, j)$  si definisce una variabile con la quale si accede all'elemento `val[0]` del campo relativo alla struttura `CvScalar`, si assegna il valore desiderato, e infine si configura il pixel  $(i, j)$  mediante la funzione `cvSet2D()`

```
CvScalar scalar;

scalar.val[0] = 'valore pixel';

cvSet2D(img, i, j, scalar);
```

In modo analogo si può procedere per un'immagine multicanale.

### 1.5.2 Metodo diretto

Questo metodo consiste nella manipolazione diretta di alcuni campi della struttura `IplImage`. Come abbiamo visto un'immagine può essere rappresentata come una matrice di dimensioni  $width \times height$ . OpenCV la gestisce come un array monodimensionale, ossia le righe sono allineate una dopo l'altra. Per poter indicizzare l'array immagine si utilizza il campo `widthStep` che contiene le dimensioni in bytes relative al passo con il quale le righe sono allineate. Il campo `widthStep` contiene quindi il numero di bytes necessari per passare dal pixel di coordinate  $(R, C)$  al pixel  $(R+1, C)$  della matrice. Per un'immagine ad un solo canale l'indice dell'array è dato da

$$i * widthStep + j, \quad \text{con } i \in [0, height - 1] \text{ e } j \in [0, width - 1]$$

Ad esempio considerando un'immagine a toni di grigio di ampiezza  $width = 3$  e altezza  $height = 4$ , l'array immagine sarà del tipo

0	1	2	P	3	4	5	P	6	7	8	P	9	10	11
---	---	---	---	---	---	---	---	---	---	---	---	---	----	----

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

| *widthStep* |

In questo modo il valore 7, che corrisponde all'elemento contenuto in posizione (2, 1) nella matrice di ordine  $3 \times 4$ , è individuato dall'indice di valore  $2 * 4 + 1$  nell'array immagine. Inoltre, essendo ogni riga della matrice allineata con passo *widthStep* si possono avere dei gap (*P*) tra la fine della riga *i*-esima e l'inizio della riga (*i*+1)-esima. Per un'immagine multicanale l'indice dell'array diventa

$$i * widthStep + j * nChannels + k$$

dove

$$k \in [0, nChannels - 1]$$

Ad esempio, per un'immagine a 8 bit unsigned e con un solo canale conveniamo nell'assegnare i seguenti campi

```
int height = img-> height;
```

```
int width = img-> width;
```

```
int step = img-> widthStep;
```

```
uchar * data = (uchar*)img-> imageData;
```

L'accesso ai pixel (*i*, *j*) si ottiene come

```
for(int i = 0; i < height; i++)
```

```
for(int j = 0; j < width; j++)
```

```
data [i * step + j] = 'valore pixel';
```

Nel caso di un'immagine multicanale è necessario utilizzare, oltre ai campi precedenti, anche il campo *nChannels*

```
int channels = img->nChannels;
```

e l'accesso ai pixel si ottiene come

```
for (int i = 0; i < height; i ++);  
for (int j = 0; j < width; j ++);  
for (int k = 0; k < nChannels; k ++);  
data [i * step + j * channels + k] = 'valore pixel';
```

## Capitolo 2

# Operazioni sulle immagini

In questo capitolo e in quelli successivi saranno trattate alcune delle tecniche di image processing basate su *operazioni spaziali*, ossia operazioni eseguibili direttamente sui pixel di un'immagine. Le operazioni spaziali sono classificabili come *operazioni puntuali* quando eseguite pixel per pixel, *operazioni locali* quando trasformano il valore di un pixel in base ai valori dei pixel in un suo intorno, e in *operazioni globali* quando trasformano il valore di un pixel sulla base dei valori assunti da tutti i pixel dell'immagine.

### 2.1 Operazioni logiche

Gli operatori logici *AND*, *OR* e *NOT* sono funzionalmente completi, in quanto ogni altro operatore può essere definito mediante questi tre. Vengono usati ad esempio nelle operazioni morfologiche, nell'operazione di masking, e in quella di inversione di un'immagine. Si tratta in ogni caso di operazioni effettuate bit per bit tra pixel omologhi. Disponendo di due immagini composte entrambe da una regione di pixel di foreground, il risultato dell'operazione *OR* consiste nell'insieme dei pixel appartenenti a una o all'altra regione, oppure ad entrambe, mentre l'operazione *AND* corrisponde all'insieme dei pixel in comune tra le due regioni. L'operazione *NOT* individua

invece tutti i pixel che non sono nella regione. In OpenCV, l'operatore *AND* è implementato dalla seguente funzione

```
void cvAnd(const CvArr * src1, const CvArr * src2, CvArr * dst,  
           const CvArr * mask = NULL);
```

I parametri *src1* e *src2* sono le due immagini in input dello stesso tipo e dimensione, mentre *dst* è l'immagine destinazione che deve contenere il risultato dell'operazione. Se diverso da *NULL*, il parametro opzionale *mask* permette di restringere l'elaborazione ad una regione di forma arbitraria. La funzione che implementa l'operatore logico *OR* ha argomenti identici alla funzione precedente

```
void cvOr(const CvArr * src1, const CvArr * src2, CvArr * dst,  
          const CvArr * mask = NULL);
```

In modo del tutto analogo è definita anche la funzione logica *XOR*, ossia *cvXor()*. Il prototipo dell'operatore *NOT* è costituito invece da due soli argomenti

```
void cvNot(const CvArr * src, CvArr * dst);
```

La funzione inverte i bit di ciascun punto dell'immagine ottenendone una nuova rispetto all'originale, in cui i valori del foreground sono scambiati con quelli del background. Possiamo ad esempio utilizzare l'operatore *AND* per eseguire l'operazione di masking rappresentata in Figura 2.1. La maschera ha valori di intensità pari a 255 nella regione di interesse, e 0 nel resto. L'elaborazione produce come risultato l'immagine 2.1c in cui è evidenziata la regione di interesse. Lo stesso risultato si può ottenere mediante l'operatore *OR* utilizzando una maschera con valori di intensità pari a 0 nella regione di interesse e 255 nel resto.

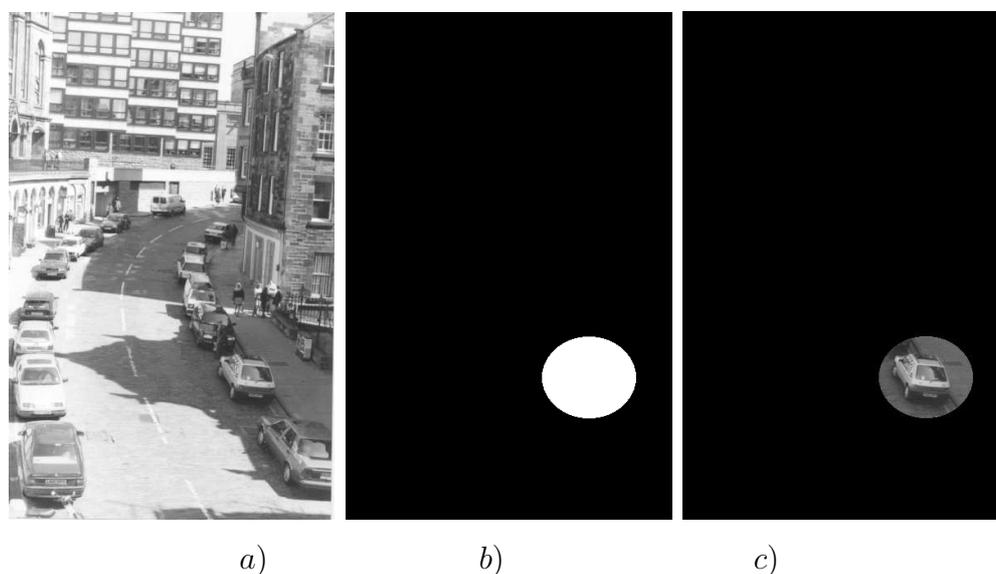


Figura 2.1: a) Immagine in scala di grigio, b) maschera per isolare la regione di interesse, c) risultato dell'operazione AND.

## 2.2 Inversione di un'immagine

Una generica operazione può essere espressa nel seguente modo

$$g(x, y) = T[f(x, y)] \quad (2.1)$$

dove  $f(x, y)$  è l'immagine da elaborare,  $T$  è l'operatore che agisce su  $f(x, y)$ , mentre  $g(x, y)$  è l'immagine risultante. La più semplice elaborazione puntuale è l'inversione, in senso fotografico, di un'immagine. Se indichiamo con  $r$  l'intensità del pixel originale, con  $s$  l'intensità del pixel elaborato e con  $[0, L - 1]$  l'intervallo dei livelli di intensità, il negativo di un'immagine si ottiene dalla trasformazione

$$s = T(r) = L - 1 - r \quad (2.2)$$

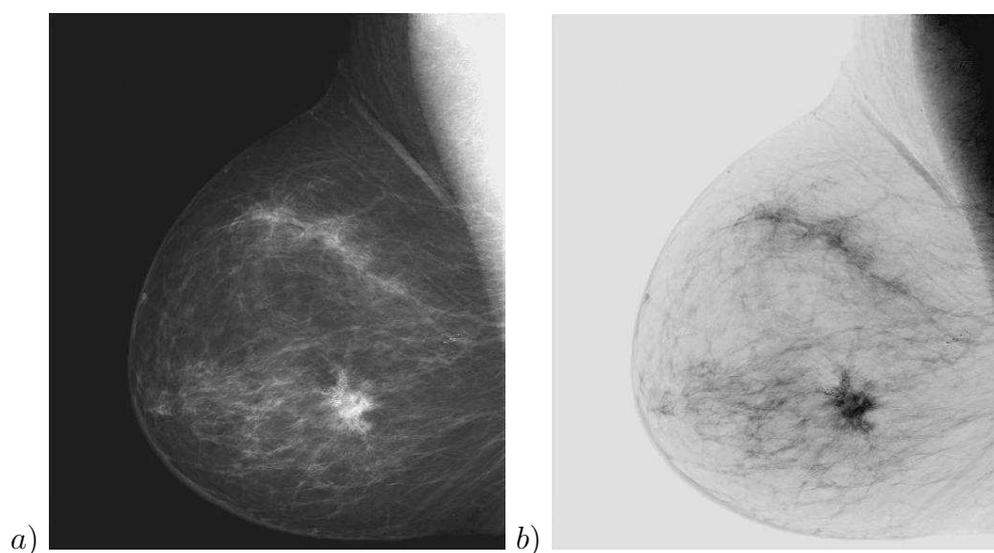


Figura 2.2: Inversione di un'immagine medica.

Questa operazione è particolarmente utile quando si vogliono evidenziare dettagli di bianco o di grigio compresi in regioni più scure. In pratica la costruzione del negativo di un'immagine a toni di grigio si ottiene sostituendo l'intensità  $I \in [0, 255]$  di ciascun pixel con il valore  $(255 - I)$ . Analogamente, per un'immagine a colori si sostituisce la tripletta  $(R, G, B) = (a, b, c)$  con la sua inversa  $(255 - a, 255 - b, 255 - c)$ . In OpenCV l'operazione di inversione si può ottenere agendo direttamente sui pixel dell'immagine alterandone il loro valore mediante un accesso diretto, oppure utilizzando semplicemente la funzione `cvNot()`. L'immagine 2.2a rappresenta il tipico caso in cui l'operazione di inversione può essere applicata, si tratta infatti di una radiografia medica che evidenzia una lesione. Il negativo, ottenuto mediante la trasformazione di intensità (2.2), consente un'analisi migliore rispetto all'immagine originale.

## 2.3 Thresholding

La trasformazione di intensità

$$s = T(r) = \begin{cases} 0 & \text{se } r < m \\ 255 & \text{altrimenti} \end{cases} \quad (2.3)$$

con  $m \in [0, 255]$ , riduce un'immagine a toni di grigio in un'immagine binaria.

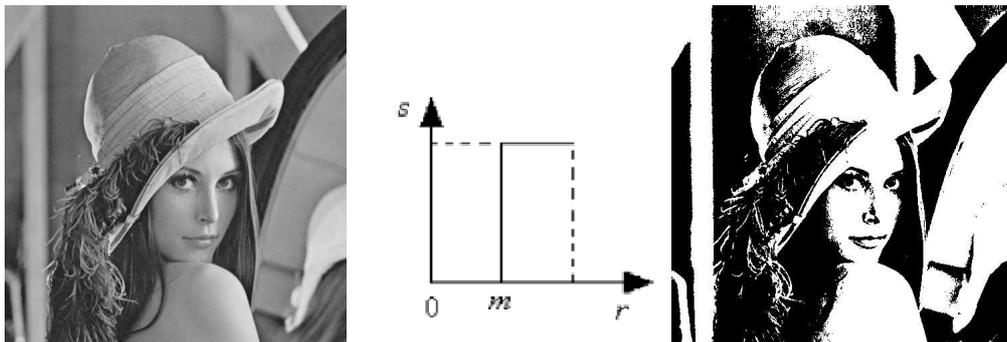


Figura 2.3: Binarizzazione

Adottando una soglia unica per l'intera immagine la trasformazione è nota come *global thresholding*. Quando la soglia varia in funzione di una qualche proprietà la trasformazione è nota invece come *variable thresholding*. In entrambi i casi queste operazioni permettono di separare gli oggetti o le regioni di interesse dallo sfondo dell'immagine. Nelle elaborazioni puntuali assume notevole importanza la conoscenza dell'istogramma, il quale consente di analizzare l'immagine fornendo una serie di informazioni statistiche utili in operazioni come l'enhancement e la segmentazione. L'istogramma di un'immagine a toni di grigio indica il numero di pixel per ciascun livello di grigio. È una funzione discreta  $h(r_k) = n_k$ , dove  $r_k$  è il  $k$ -esimo valore di intensità ed  $n_k$  è il numero di pixel con intensità  $r_k$ . È in pratica una stima dell'occorrenza dei livelli di grigio, utile perchè fornisce una descrizione globale dell'immagine. Gli istogrammi sono visualizzati graficamente come semplici grafici dove l'asse orizzontale corrisponde ai valori  $r_k$ , mentre l'asse verticale ai valori di intensità  $h(r_k)$ . La sogliatura dell'immagine 2.4 si ottiene dall'osservazione dell'istogramma, dove la distribuzione di intensità dei



Figura 2.4: Impronta digitale e relativo Istogramma

pixel dell'oggetto e dello sfondo sono distinte. Ponendo la soglia al valore  $r_k$  contenuto nella valle compresa tra i due picchi, l'immagine 2.5a è il risultato che si ottiene assegnando ai pixel di intensità maggiore di 127 il valore 255, e ai restanti il valore 0. Viceversa la 2.5b si ottiene assegnando il valore 0 ai pixel di intensità maggiore di 127, e ai restanti il valore 255. Generalmente la binarizzazione tramite soglia globale si esegue quando è presente un unico oggetto di interesse caratterizzato da uno sfondo uniforme. La funzione di thresholding ha il seguente prototipo

```
double cvThreshold (const CvArr * src, CvArr * dst,  
double threshold, double max_value, int threshold_type);
```

L'immagine in input deve avere un solo canale con profondità pari a 8 bit oppure a 32 bit floating point, mentre l'immagine di destinazione deve essere a 8 bit. Il parametro *threshold* è ovviamente la soglia, mentre *max\_value* è il valore massimo da assegnare ai pixel. La modalità operativa della funzione è così di seguito sintetizzata.



Figura 2.5: Immagini binarie ottenute mediante global thresholding

$threshold\_type = CV\_THRESH\_BINARY$

$$dst(x, y) = \begin{cases} max\_value & \text{se } src(x, y) > threshold \\ 0 & \text{altrimenti} \end{cases}$$

$threshold\_type = CV\_THRESH\_BINARY\_INV$

$$dst(x, y) = \begin{cases} max\_value & \text{altrimenti} \\ 0 & \text{se } src(x, y) > threshold \end{cases}$$

*threshold\_type* = *CV\_THRESH\_TRUNC*

$$dst(x, y) = \begin{cases} threshold & \text{se } src(x, y) > threshold \\ src(x, y) & \text{altrimenti} \end{cases}$$

*threshold\_type* = *CV\_THRESH\_TOZERO*

$$dst(x, y) = \begin{cases} src(x, y) & \text{se } src(x, y) > threshold \\ 0 & \text{altrimenti} \end{cases}$$

*threshold\_type* = *CV\_THRESH\_TOZERO\_INV*

$$dst(x, y) = \begin{cases} 0 & \text{se } src(x, y) > threshold \\ src(x, y) & \text{altrimenti} \end{cases}$$

Anche per le tecniche di *variable thresholding* esiste una funzione specifica. Questa determina la soglia in due modi diversi, basati entrambi sul calcolo della media dell'intensità dei pixel di un intorno.

```

void cvAdaptiveThreshold(CvArr * src, CvArr * dst, double max_val,
    int adaptive_method = CV_ADAPTIVE_THRESH_MEAN_C,
    int threshold_type = CV_THRESH_BINARY,
    int block_size = 3, double param1 = 5);

```

Il parametro *max\_val* corrisponde al valore da assegnare ai pixel quando *threshold\_type* è settato a CV\_THRESH\_BINARY o CV\_THRESH\_BINARY\_INV. Il parametro *adaptive\_method* è posto per default a CV\_ADAPTIVE\_THRESH\_MEAN\_C, quindi la soglia adattiva locale è calcolata in ogni pixel effettuando la media in un suo intorno  $block\_size \times block\_size$  e sottraendo al risultato così ottenuto il valore fissato *param1*. Se invece il parametro *adaptive\_method* è settato a CV\_ADAPTIVE\_THRESH\_GAUSSIAN\_C, la soglia locale viene calcolata come la media pesata in un intorno  $block\_size \times block\_size$  per poi sottrarre al risultato il valore *param1*. I pesi corrispondono ai coefficienti di una maschera Gaussiana di dimensione pari all'intorno. Si nota che al parametro *block\_size* si può assegnare solo un valore dispari maggiore o uguale a 3, in modo tale che il centro dell'intorno possa coincidere con il pixel rispetto al quale viene calcolata la soglia.

Ad esempio: siano  $threshold\_type = CV\_THRESH\_BINARY$  e  $param1 = 10$ . Se il pixel di coordinate  $(x, y)$  ha intensità 100 e la media delle intensità nel suo intorno vale 105, allora al pixel di coordinate  $(x, y)$  verrà assegnato il valore *max\_value*, poichè  $105 - 10 = 95 = T(x, y) < 100$ . In definitiva, il modo di operare della funzione è così sintetizzato.

$threshold\_type = CV\_THRESH\_BINARY$

$$dst(x, y) = \begin{cases} max\_value & \text{se } src(x, y) > T(x, y) \\ 0 & \text{altrimenti} \end{cases}$$

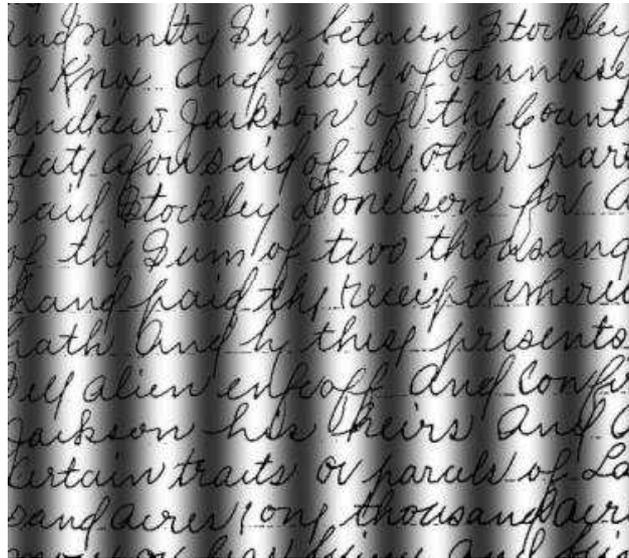


Figura 2.6: Immagine corrotta da una variazione di intensità sinusoidale

*threshold\_type = CV\_THRESH\_BINARY\_INV*

$$dst(x, y) = \begin{cases} 0 & \text{se } src(x, y) > T(x, y) \\ max\_value & \text{altrimenti} \end{cases}$$

Generalmente le tecniche di soglia adattiva sono efficaci nel trattamento di immagini che presentano una elevata variazione di intensità luminosa. In Figura 2.6 abbiamo l'immagine di un documento di testo corrotto da una variazione di intensità sinusoidale. L'effetto è dovuto all'errato funzionamento del dispositivo di scansione. L'immagine 2.7 è il risultato accettabile ottenuto dall'operazione di sogliatura con il metodo `CV_ADAPTIVE_THRESH_MEAN_C`. Le variazioni di intensità sono state isolate assegnando alla funzione i valori: *block\_size = 9*, *param1 = 25*.

Indimently Six between Stockley  
 of Knox And State of Tennessee  
 Andrew Jackson of the County  
 State of said of the other part  
 said Stockley Donelson for a  
 of the sum of two thousand  
 hand paid the receipt where  
 hath and by these presents  
 self alien enfeof And confir  
 Jackson his heirs And a  
 certain tracts or parcels of La  
 sand acres one thousand are

Figura 2.7: Risultato ottenuto mediante variable thresholding

## 2.4 Trasformazioni geometriche

Le trasformazioni geometriche elementari eseguibili su un'immagine sono le traslazioni, le rotazioni, i cambiamenti di scala, e le distorsioni (orizzontale e verticale). Queste vengono utilizzate prevalentemente in fase di pre-processing e nelle operazioni di computer graphics. Le trasformazioni, pur modificando le relazioni spaziali tra i pixel, sono vincolate in modo da preservare la continuità delle linee e i rapporti reciproci di posizione e proporzione degli oggetti di scena. Consistono in particolare di due operazioni: la trasformazione spaziale delle coordinate, e l'interpolazione che assegna i valori di intensità alle nuove posizioni ottenute dalla trasformazione.

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & t_x \\ a_{21} & a_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} v \\ w \\ 1 \end{bmatrix} \quad (2.4)$$

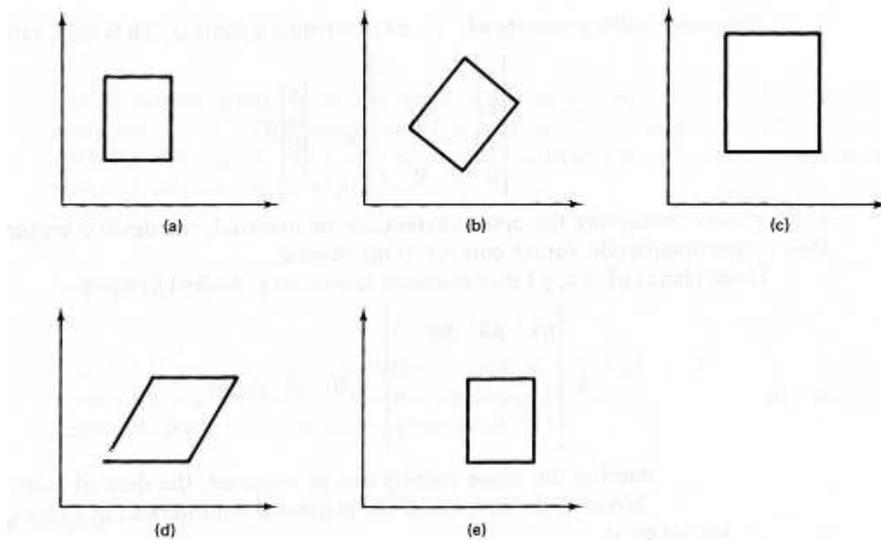


Figura 2.8: Trasformazioni geometriche

La trasformazione affine (2.4) permette di deformare i rettangoli in parallelogrammi in base ai valori degli elementi della matrice. L'introduzione delle coordinate omogenee permette anche di concatenare una sequenza di trasformazioni geometriche elementari in un'unica matrice, la quale è ottenuta dal prodotto (nell'ordine opportuno) delle singole matrici di trasformazione. Le trasformazioni possono essere di tipo *forward mapping*

$$(x, y) = T(v, w) \quad (2.5)$$

dove per ogni posizione  $(v, w)$  dei pixel dell'immagine sorgente viene calcolata la corrispondente posizione spaziale  $(x, y)$  dei pixel dell'immagine trasformata. In questo caso alcune coordinate potrebbero non essere coperte da nessun pixel di input, oppure potrebbero essere destinazione di più pixel dell'immagine sorgente. Con una trasformazione di tipo *inverse mapping* si evita il problema della copertura non uniforme. Tutte le posizioni spaziali dell'immagine di destinazione vengono visitate, e per ciascuna di esse si determinano le coordinate corrispondenti nell'immagine sorgente

$$(v, w) = T^{-1}(x, y) \quad (2.6)$$

Le tecniche di interpolazione come *nearest neighbor*, *bilineare* e *bicubica*, sono utilizzate nelle operazioni di rotazione, trasformazione di scala e distorsione. In genere introducono nell'immagine trasformata degli artefatti, i quali sono più o meno evidenti in base alla tecnica utilizzata. Ad esempio, si possono avere immagini trasformate i cui bordi appaiono frastagliati o immagini che presentano un certo grado di blurring.

### 2.4.1 Traslazione

La traslazione a corpo rigido di un'immagine consiste nello spostamento di ogni suo punto di una certa distanza lungo una direzione prestabilita. Considerando  $f(v, w)$  come l'immagine sorgente e  $g(x, y)$  come l'immagine traslata, si può scrivere

$$g(x, y) = f(v + t_x, w + t_y) \quad (2.7)$$

dove  $t_x$  e  $t_y$  sono rispettivamente lo spostamento lungo l'asse orizzontale e verticale. Per eseguire una trasformazione geometrica si utilizza la funzione generica `cvWarpAffine()`, la quale è caratterizzata da una matrice di trasformazione di dimensione  $2 \times 3$ . Nel caso particolare di una traslazione, la matrice deve avere la seguente forma

$$T = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \end{bmatrix}$$

Il prototipo della funzione è invece il seguente

```
void cvWarpAffine( const CvArr * src, CvArr * dst,
                  const CvMat * map_matrix, int flags = CV_INTER_LINEAR+
                  CV_WARP_FILL_OUTLIERS, CvScalar fillval = cvScalarAll(0));
```

Le immagini di input e output devono possedere entrambe uno o tre canali. Il parametro *map\_matrix* è la matrice di trasformazione, mentre il parametro *flags* definisce la tecnica di interpolazione usata. L'aggiunta di CV\_WARP\_FILL\_OUTLIERS consente di assegnare lo stesso valore di intensità, tramite l'argomento *fillval*, ai pixel dell'immagine trasformata che non trovano corrispondenza nell'immagine sorgente. L'interpolazione bilineare, applicata per default dalla funzione, stima il valore del pixel sconosciuto mediante i quattro pixel più vicini. In pratica, il valore di intensità  $h(x, y)$  del punto di coordinate  $(x, y)$  interno ad un quadrato di lato unitario si ottiene dall'equazione

$$h(x, y) = ax + by + cxy + d \quad (2.8)$$

che definisce un paraboloide iperbolico. I coefficienti  $a, b, c, d$  sono determinati dalle quattro equazioni nelle quattro incognite che si ottengono dai quattro pixel più vicini al punto  $(x, y)$ . Considerando le coordinate dei quattro vertici  $(0, 0), (0, 1), (1, 0), (1, 1)$ , avremo

$$a \cdot 0 + b \cdot 0 + c \cdot 0 \cdot 0 + d = h(i, j) \quad (2.9)$$

$$a \cdot 0 + b \cdot 1 + c \cdot 0 \cdot 1 + d = h(i, j + 1) \quad (2.10)$$

$$a \cdot 1 + b \cdot 0 + c \cdot 1 \cdot 0 + d = h(j + 1, j) \quad (2.11)$$

$$a \cdot 1 + b \cdot 1 + c \cdot 1 \cdot 1 + d = h(i + 1, j + 1) \quad (2.12)$$

Dalle quattro equazioni (2.9) – (2.12) otteniamo i coefficienti cercati

$$d = h(i, j) \quad (2.13)$$

$$b = h(i, j + 1) - h(i, j) \quad (2.14)$$

$$a = h(i + 1, j) - h(i, j) \quad (2.15)$$

$$c = h(i + 1, j + 1) + h(i, j) - h(i + 1, j) - h(i, j + 1) \quad (2.16)$$

Con il seguente codice si costruisce la matrice di trasformazione  $T$  che consente di traslare l'immagine di Figura 2.9 lungo le direzioni  $x$  e  $y$ .

Figura 2.9: Immagine di  $640 \times 480$  pixel

$$T = \begin{bmatrix} 1 & 0 & 60 \\ 0 & 1 & 50 \end{bmatrix}$$

```
//spostamento in pixel lungo l'asse X e Y
int  $t_x$  = 60;
int  $t_y$  = 50;

// matrice di trasformazione T con elementi  $T_{ij}$ 
CvMat *T = cvCreateMat( 2, 3, CV_32FC1);

//assegna il valore 1 all'elemento (0,0)
cvmSet(T,0,0,1);
//assegna il valore 0 all'elemento (0,1)
cvmSet(T,0,1,0);
```

```
//assegna il valore 0 all'elemento (1,0)
cvmSet(T,1,0,0);
//assegna il valore 1 all'elemento (1,1)
cvmSet(T,1,1,1);

//assegna il valore  $t_x$  all'elemento (0,2)
cvmSet(T,0,2,tx);
//assegna il valore  $t_y$  all'elemento (1,2)
cvmSet(T,1,2,ty);

// traslazione
cvWarpAffine (src, dst, T, CV_INTER_LINEAR +
              CV_WARP_FILL_OUTLIERS, cvScalarAll(0) );
```



Figura 2.10: Traslazione

Come si può osservare dalla Figura 2.10, la traslazione ha prodotto una perdita di informazione. L'origine dell'immagine è traslata di 60 pixel lungo

l'asse  $x$  e di 50 pixel lungo l'asse  $y$ , così come tutti gli altri punti. Operando in questo modo, alcuni punti dell'immagine sorgente vengono mappati in altrettanti punti che cadono al di fuori dell'immagine di destinazione. Nel nostro caso lo spazio vuoto che si viene a creare è colmato assegnando il valore di intensità pari a 0.

### 2.4.2 Trasformazioni di scala

Si possono modificare le dimensioni di un'immagine moltiplicando ciascuna coordinata per un fattore di scala. Le equazioni delle coordinate sono

$$x = av, \quad y = bw \quad (2.17)$$

con  $a$  e  $b$  reali e positivi. Per uno stesso fattore ( $a = b$ ) maggiore di uno si ha un ingrandimento dell'immagine (zooming), mentre per uno stesso fattore minore di uno si ha un rimpicciolimento (shrinking). Il ridimensionamento ( $a \neq b$ ) si può eseguire utilizzando la precedente funzione `cvWarpAffine()` con una matrice di trasformazione  $2 \times 3$  del tipo

$$T = \begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \end{bmatrix}$$

oppure mediante la specifica funzione

```
void cvResize(const CvArr * src, CvArr * dst,
              int interpolation = CV_INTER_LINEAR);
```

I primi due parametri sono le solite immagini sorgente e destinazione, mentre il terzo definisce il tipo di interpolazione. Ponendo *interpolation* al valore CV\_INTER\_NN la funzione applica il metodo nearest neighbor, che consiste nel porre il valore di ogni pixel sconosciuto  $(x_D, y_D)$  uguale al valore del pixel noto più vicino  $h(x_D, y_D) = (\text{round}(x_s), \text{round}(y_s))$ . Si tratta di una tecnica semplice che richiede un tempo di esecuzione inferiore alle altre, ma ha il difetto di introdurre distorsioni lungo i bordi. L'interpolazione bilineare fornisce risultati migliori rispetto al metodo precedente, ma introduce un certo

grado di blurring in quanto si vengono a creare nuovi livelli di grigio rispetto all'immagine sorgente. Il tipico effetto che si può riscontrare in questo caso è la scomparsa dei dettagli fini, soprattutto in caso di zooming dell'immagine. Con CV\_INTER\_CUBIC si applica invece l'interpolazione bicubica. Il valore di intensità assegnato al punto  $(x, y)$  nell'intorno dei sedici pixel più vicini si ottiene dall'equazione

$$h(x, y) = \sum_{i=0}^3 \sum_{j=0}^3 a_{ij} x^i y^j \quad (2.18)$$

I coefficienti si determinano dalle sedici equazioni in sedici incognite. Generalmente l'interpolazione bicubica preserva meglio i dettagli rispetto alle due tecniche precedenti, e l'effetto tipico è quello di rendere i contorni più sfumati e dolci. Inoltre, l'ottimo rapporto tra qualità e complessità computazionale lo rendono uno dei metodi più utilizzati. La Figura 2.11 rappresenta il ridimensionamento dell'immagine campione ottenuto tramite la funzione *cvResize()*. Da notare che in tal caso è necessario definire direttamente le dimensioni dell'immagine trasformata.

```
// fattori di scala relativi agli assi X e Y
double i=1.5;
double j=0.5;
//nuove dimensioni dell'immagine
int width=cvRound((src->width)*i);
int height=cvRound((src->height)*j);
//crea l'immagine di destinazione di dimensioni opportune
IplImage* dst=cvCreateImage(cvSize(width,height),
                             IPL_DEPTH_8U, 1);

//funzione ridimensionamento
cvResize(src, dst, CV_INTER_LINEAR);
```



Figura 2.11: Ridimensionamento

### 2.4.3 Rotazione

La rotazione di un'immagine si esprime, nell'equazione delle coordinate, come

$$(x, y) = (v \cos \theta - w \sin \theta, v \sin \theta + w \cos \theta) \quad (2.19)$$

dove  $\theta$  è l'angolo di rotazione rispetto all'asse orizzontale. Si può ruotare e ridimensionare l'immagine utilizzando la seguente funzione che calcola l'apposita matrice di trasformazione

```
cv2DRotationMatrix(CvPoint2D32f center, double angle,  
double scale, CvMat * map_matrix);
```

Il parametro *center* è il centro di rotazione espresso in coordinate 2D, mentre *angle* definisce l'angolo di rotazione espresso in gradi. Assegnando a quest'ultimo un valore positivo, la rotazione avviene in senso antiorario. Il parametro *scale* corrisponde al fattore di scala dell'immagine ruotata, mentre il parametro *map\_matrix* è la matrice di trasformazione  $2 \times 3$ . In generale, ponendo  $\alpha = scale \cdot \cos \theta$ ,  $\beta = scale \cdot \sin \theta$ , e  $(c_x, c_y)$  come centro di rotazione, si ottiene la matrice di trasformazione

$$T = \begin{bmatrix} \alpha & \beta & (1 - \alpha) \cdot c_x - \beta \cdot c_y \\ -\beta & \alpha & \beta \cdot c_x + (1 - \alpha) \cdot c_y \end{bmatrix}$$



Figura 2.12: Rotazione antioraria di 15°

Utilizziamo questa funzione per ruotare di 15° l'immagine di Figura 2.9, facendo coincidere il centro di rotazione con quello geometrico. Da notare che i bordi nell'immagine ruotata sono frastagliati. Questo effetto si presenta in modo più evidente con la tecnica di interpolazione nearest neighbor.

```
...
// centro di rotazione
CvPoint2D32f center=cvPoint2D32f(src->width/2, dst->height/2);
...
// matrice di trasformazione T
CvMat* T = cvCreateMat(2,3,CV_32FC1);

// calcola la matrice di rotazione T
cv2DRotationMatrix(center, angolo, 1.0, T);

// trasformazione affine
cvWarpAffine(src, dst, T, CV_INTER_LINEAR+

CV_WARP_FILL_OUTLIERS, cvScalarAll(0));
```

In alternativa alla funzione di trasformazione *cvWarpAffine()* si può utilizzare la seguente

```
void cvGetQuadrangleSubPix(const CvArr * src, CvArr * dst,  
  
const CvMat * map_matrix);
```

Questa funzione può operare con immagini multicanale, anche di tipo diverso tra loro. Quindi è possibile avere l'immagine di input a 8 bit e quella di output a 32 bit floating point. La tecnica di interpolazione implementata è quella bilineare. Si osserva, a differenza del caso precedente, che i triangoli adiacenti ai lati dell'immagine non risultano più vuoti. La funzione ricostruisce l'intensità dei pixel mancanti mediante la replica dei bordi.



Figura 2.13: Rotazione antioraria caratterizzata dalla replica dei bordi

### 2.4.4 Distorsione geometrica

La Figura 2.14 mostra la distorsione geometrica di un'immagine lungo le due direzioni, orizzontale e verticale. La trasformazione affine che consente di deformare un rettangolo in un parallelogramma è definita mappando tre punti dell'immagine sorgente, ad esempio i vertici opposti  $(0, 0)$ ,  $(width - 1, 0)$ ,  $(0, height - 1)$ , in tre punti specifici dell'immagine di destinazione. La matrice di trasformazione *map\_matrix* si costruisce tramite la funzione

```
cvGetAffineTransform(const CvPoint2D32f * pts_src,
                    const CvPoint2D32f * pts_dst, CvMat * map_matrix);
```

I parametri *pts\_src* e *pts\_dst* sono i rispettivi array che contengono le coordinate dei tre punti bidimensionali delle due immagini. La distorsione si ottiene passando la matrice di trasformazione alla già nota *cvWarpAffine()*. Nel nostro caso, disponendo di un'immagine di  $640 \times 480$  pixel, la trasformazione è stabilita dalla seguente corrispondenza tra punti:

$$(0, 0) \mapsto (0, 0), \quad (639, 0) \mapsto (609, 100), \quad (0, 479) \mapsto (29, 379)$$

...

```
CvMat* map_matrix=cvCreateMat(2,3,CV_32FC1);
```

```
// definizione della trasformazione
```

```
CvPoint2D pts_src[3], pts_dst[3];
```

```
pts_src[0]=cvPoint2D32f(0.0, 0.0);
```

```
pts_src[1]=cvPoint2D32f(639.0, 0.0);
```

```
pts_src[2]=cvPoint2D32f(0.0, 479.0);
```

```
pts_dst[0]=cvPoint2D32f(0.0, 0.0);

pts_dst[1]=cvPoint2D32f(609.0, 100.0);

pts_dst[2]=cvPoint2D32f(29.0, 379.0);

//matrice di trasformazione
cvGetAffineTransform(pts_src, pts_dst, map_matrix);

//funzione di trasformazione
cvWarpAffine(src, dst, map_matrix, CV_INTER_LINEAR +
CV_WARP_FILL_OUTLIERS, cvScalarAll (0));
```



Figura 2.14: Distorzione

# Capitolo 3

## Filtri spaziali

I filtri spaziali sono ampiamente utilizzati nelle operazioni di miglioramento della qualità delle immagini, nell'estrazione delle caratteristiche, e nella riduzione del rumore.

### 3.1 Elementi base

Un filtro spaziale, detto anche *maschera* o *kernel*, è una regione rettangolare caratterizzata da un'operazione predefinita. L'operazione viene eseguita sui pixel dell'immagine corrispondenti alla suddetta regione, e le modifiche indotte dal filtro generano la cosiddetta immagine filtrata. Il filtro opera sovrapponendosi all'immagine partendo generalmente dall'angolo in alto a sinistra. Quando il punto centrale (*anchor point*) coincide con il pixel da modificare, quest'ultimo verrà trasformato nel nuovo pixel dell'immagine di output. Il kernel si sposta da sinistra verso destra, e procede in questo modo fino a terminare riga per poi ripartire dalla riga successiva modificando i valori di input nel modo indicato. Il procedimento si ripete in questo modo per tutti i punti dell'immagine.

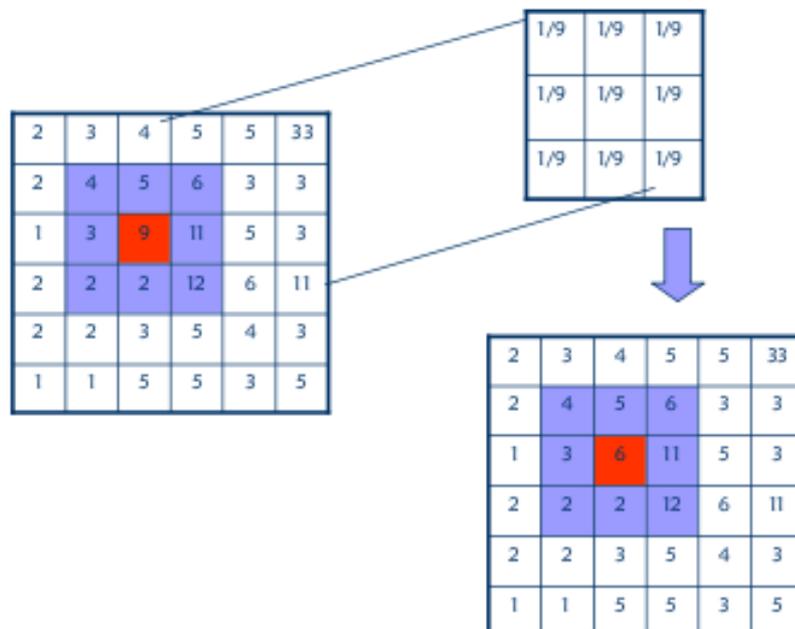


Figura 3.1: Filtraggio

Il filtro viene così applicato mediante un procedimento di convoluzione espresso dalla seguente equazione

$$g(x, y) = \sum_{s=-a}^{s=a} \sum_{t=-b}^{t=b} w(s, t) f(x + s, y + t) \quad (3.1)$$

dove  $f(x, y)$  è l'immagine originale di dimensioni  $n \times n$ ,  $w(s, t)$  è il kernel di convoluzione  $m \times m$ , mentre  $g(x, y)$  è l'immagine di output. L'equazione è calcolata per tutti i valori  $x$  e  $y$ , e l'inserimento di  $m - 1$  zeri (*zero padding*) sui lati della  $f$  consente ad ogni elemento del kernel di visitare tutti i pixel dell'immagine. La complessità computazionale della convoluzione è elevata, un kernel di dimensioni  $m \times m$  richiede  $m^2$  somme e altrettanti prodotti per ogni pixel dell'immagine. In genere per una maschera di dimensioni dispari e con due assi di simmetria si può eseguire il procedimento in modo efficiente tramite tecniche di separazione, applicando cioè due filtri monodimensionali in serie invece di un'unico filtro bidimensionale. In Figura 3.1 è rappresentato un semplice filtro particolarmente efficiente. Si tratta di un *filtro di*

*media*, il quale sostituisce ad ogni pixel il valor medio dei suoi vicini incluso il pixel stesso. Essendo i coefficienti unitari l'operazione non richiede alcuna moltiplicazione, è sufficiente sommare le intensità dei pixel 8-vicini dell'immagine e dividere per 9. Questo tipo di elaborazione provoca una perdita di definizione dei contorni, ottenendo così un'immagine sfocata. Poichè l'operazione è lineare si parla anche di *filtro spaziale lineare*. Un *filtro spaziale non lineare* esegue operazioni più complesse rispetto alle semplici operazioni lineari. Il filtro non lineare maggiormente usato è quello *mediano*, il quale sostituisce ad ogni pixel dell'immagine il valore mediano dei suoi vicini, incluso il pixel stesso. Ha quindi il vantaggio di non introdurre nuovi valori di grigio nell'immagine elaborata. Il valore è calcolato ordinando tutti i pixel dell'intorno prescelto, e sostituendo al pixel in esame il valore centrale dell'insieme ordinato. In particolare, se l'intorno contiene un numero pari di pixel si prende la media dei due valori centrali. Questo filtro viene applicato principalmente per ridurre il rumore di tipo impulsivo (salt e pepper). In genere, i filtri non lineari permettono di operare selettivamente attenuando il rumore e preservando le strutture principali dell'immagine, come i bordi.

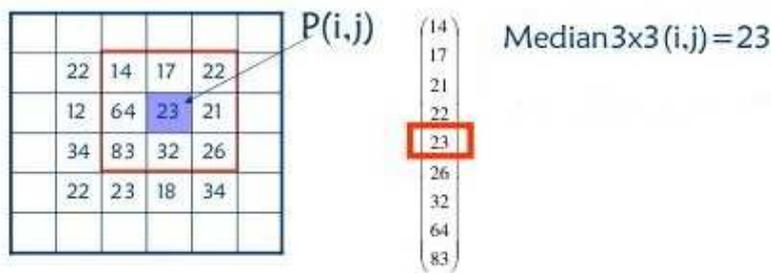


Figura 3.2: Filtro mediano

Ogni filtro lineare può essere costruito e applicato mediante la funzione

```
void cvFilter2D(const CvArr * src, CvArr * dst, const CvMat * kernel,
```

```
    CvPoint anchor = cvPoint(-1, -1));
```

In pratica, il filtro viene costruito come una matrice numerica. Per default il centro della matrice coincide con l'anchor point, il quale può essere riposizionato internamente al filtro modificando in modo opportuno gli argomenti della funzione *cvPoint()*. La seguente porzione di codice costruisce il filtro rappresentato nella Tabella 3.1, il quale sarà utilizzato più avanti nelle operazioni di miglioramento della qualità dell'immagine.

```
// dimensioni del kernel  $w \times h$ 
int w=3; int h=3;

// coefficienti del kernel
float coeff [3][3]={{-1,-1,-1},{-1,9,-1},{-1,-1,-1}};

// matrice  $h \times w$ 
CvMat* kernel= cvCreateMat(h, w, CV_32FC1);

//carica i coefficienti assegnati alla matrice
for (int i=0; i<h; i++)
for (int j=0; j<w; j++)
cvSet2D(kernel, i, j, cvScalar(coeff[i][j]));

//filtraggio mediante il kernel opportunamente costruito
cvFilter2D(src, dst, kernel);
```

-1	-1	-1
-1	9	-1
-1	-1	-1

Tabella 3.1: Filtro di sharpening

## 3.2 Filtri di smoothing

I filtri di smoothing si utilizzano generalmente per sfocare le immagini e per ridurre l'eventuale presenza di rumore. È possibile sfocare un'immagine sia per far emergere oggetti di interesse che per ridurre dettagli irrilevanti, dove per dettagli irrilevanti si considerano quelle regioni piccole rispetto alle dimensioni del filtro. Si possono usare filtri che calcolano la media dei valori dei pixel in un intorno simmetrico, in questo modo le intensità degli oggetti piccoli tendono ad essere inglobate nello sfondo mentre gli oggetti più grandi diventano più semplici da individuare. Il filtro mediano, ad esempio, permette di ridurre il rumore producendo una minore sfocatura rispetto ai filtri lineari di dimensioni simili. La seguente funzione implementa cinque differenti filtri di smoothing

```
void cvSmooth( const CvArr * src, CvArr * dst,  
  
int smoothtype = CV_GAUSSIAN, int size1 = 3,  
int size2 = 0, double sigma1 = 0, double sigma2 = 0);
```

I parametri *size1*, *size2*, *sigma1* e *sigma2* dipendono dal filtro scelto, ossia dal valore passato al parametro *smoothtype*. Assegnando il valore CV\_BLUR, la funzione applica un filtro di media di dimensione  $size1 \times size2$ . Il parametro *size1* può assumere solo valori dispari, in questo modo ogni pixel dell'immagine di input coinciderà con il centro del filtro. Se non assegniamo nessun valore al parametro *size2* questo assume lo stesso valore di *size1*. La funzione applica il filtro di media alle immagini con uno o tre canali e con profondità pari a 8 bit, 16 bit e 32 bit floating point. Se al parametro *smoothtype* si assegna CV\_BLUR\_NO\_SCALE la funzione applica un filtro la cui risposta è data dalla somma dei valori compresi nell'intorno  $size1 \times size2$ . In questo caso l'assenza del fattore di normalizzazione  $\frac{1}{size1 \times size2}$  impone che l'immagine di output deve avere differente profondità rispetto a quella di input, in modo da evitare l'overflow dei valori di intensità. In particolare, per un'immagine di input con un solo canale e con profondità a 8 bit unsigned,

quella di output deve essere a 16 bit signed oppure a 32 bit floating point. Un'altro valore che si può assegnare al parametro è CV\_MEDIAN, con il quale la funzione applica il filtro mediano di dimensione  $size1 \times size1$ . Il filtro lavora solo con immagini a uno o tre canali e con profondità pari a 8 bit unsigned. Con il parametro *smoothtype* posto a CV\_GAUSSIAN, l'immagine viene elaborata mediante un filtro gaussiano di dimensione  $size1 \times size2$ , il quale si basa sull'approssimazione discreta della funzione gaussiana bidimensionale a valor medio nullo

$$h(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \quad (3.2)$$

La deviazione standard  $\sigma$  è il parametro che modella la funzione e definisce l'area di influenza del filtro, dove il peso dei coefficienti è inversamente proporzionale alla distanza del pixel rispetto a quello centrale. Tra le proprietà vi è la simmetria circolare, ossia il filtro esegue l'operazione di smoothing in modo identico in tutte le direzioni, e la separabilità. Siano  $h(i, j)$  e  $f(i, j)$  rispettivamente la funzione di trasferimento e l'immagine da filtrare. Ne segue che

$$\begin{aligned} g(i, j) &= h(i, j) \star f(i, j) = \sum_l \sum_k h(l, k) f(i-l, j-k) \\ &= \sum_l \sum_k e^{-\frac{(l^2+k^2)}{2\sigma^2}} f(i-l, j-k) \\ &= \sum_l e^{-\frac{l^2}{2\sigma^2}} \left[ \sum_k e^{-\frac{k^2}{2\sigma^2}} f(i-l, j-k) \right] \end{aligned} \quad (3.3)$$

dove l'espressione nelle parentesi indica la convoluzione dell'immagine  $f(i, j)$  con la funzione gaussiana monodimensionale  $h(k)$  che rappresenta la componente verticale. Il risultato del filtraggio verticale produce l'immagine di output  $g_v(i, j)$  che viene data in input all'operatore di convoluzione orizzontale  $h(l)$ . Scambiando l'ordine, prima quella orizzontale e poi quella verticale, i risultati non cambiano in virtù della proprietà associativa e commutativa della convoluzione. La deviazione standard della funzione di

$$\frac{1}{16} \times \begin{array}{|c|c|c|} \hline 1 & 2 & 1 \\ \hline 2 & 4 & 2 \\ \hline 1 & 2 & 1 \\ \hline \end{array} \iff \frac{1}{16} \times \begin{array}{|c|c|c|} \hline 1 & 2 & 1 \\ \hline \end{array} \otimes \begin{array}{|c|} \hline 1 \\ \hline 2 \\ \hline 1 \\ \hline \end{array}$$

Tabella 3.2: Separabilità del filtro gaussiano

smoothing è definita dal parametro *sigma1*. Se non specificato la funzione calcola automaticamente il valore di  $\sigma$  dalle dimensioni della maschera, ossia

$$\sigma = \left(\frac{n}{2} - 1\right) * 0.3 + 0.80 \quad (3.4)$$

con  $n = size1$  per il kernel orizzontale e  $n = size2$  per il kernel verticale. Al parametro *sigma2* si assegna un valore diverso da quello di default quando si vuole una maschera non quadrata. In questo caso *sigma1* e *sigma2* saranno rispettivamente i valori di  $\sigma$  lungo le direzioni orizzontali e verticali. Il filtro gaussiano può operare con immagini a uno o tre canali, con profondità pari a 8 bit, 16 bit o 32 bit floating point. Con il valore CV\_BILATERAL assegnato al parametro *smoothtype*, la funzione applica un filtro non lineare denominato *filtro bilaterale*. A differenza di quello gaussiano, il filtro bilaterale è in grado di eseguire lo smoothing dell'immagine senza attenuare i contorni (*edge preserving smoothing*), e si basa sull'azione congiunta di un filtro lineare e di un filtro non lineare. Quello lineare, detto *domain filter*, è un filtro gaussiano che si ottiene dall'equazione (3.2). Il filtro non lineare, detto *range filter*, è definito dalla funzione

$$r(a_i) = e^{-\frac{(f(a_i) - f(a_0))^2}{2\sigma_R^2}} \quad (3.5)$$

dove  $f$  è l'intensità luminosa,  $a_i$  è il pixel  $i$ -esimo in un'intorno di dimensione  $n \times n$ , e  $a_0$  il pixel centrale. Il range filter si ottiene dalla differenza di intensità tra i pixel dell'intorno e il pixel centrale, mentre quello bilaterale si ottiene dalla moltiplicazione punto per punto di entrambi i filtri. Il nuovo valore del pixel centrale  $a_0$  dell'intorno è dato da

$$h(a_0) = \frac{1}{k} \sum_{i=0}^{n-1} f(a_i) d(a_i) r(a_i) \quad (3.6)$$

con  $k = \sum_{i=0}^{n-1} d(a_i)r(a_i)$  costante di normalizzazione. In particolare, il filtro implementato dalla funzione ha dimensione  $3 \times 3$  e può agire su immagini che hanno uno o tre canali con profondità a 8 bit oppure a 32 bit floating point. Le caratteristiche si assegnano mediante il parametro *sigma1* per il range filter e *sigma2* per il domain filter. Più è grande il valore assegnato a *sigma1*, maggiore è l'effetto watercolor sull'immagine filtrata, analogamente più è grande il valore *sigma2*, maggiore è l'effetto di blurring. Consideriamo ora l'immagine di Figura 3.3, corrotta da rumore impulsivo, sulla quale si esegue lo smoothing utilizzando sia un filtro di media che un filtro mediano entrambi di dimensioni  $5 \times 5$ . Come si può notare dal confronto tra le due immagini 3.4, il filtro mediano produce il risultato migliore. Infatti, il filtro di media attenua il rumore ma tende anche a creare nuovi livelli di grigio prima non esistenti, inoltre attenua le intensità elevate in modo indiscriminato causando una sfocatura con perdita di dettaglio fine. Il filtro mediano elimina invece i picchi di intensità elevate senza influenzare i pixel vicini.

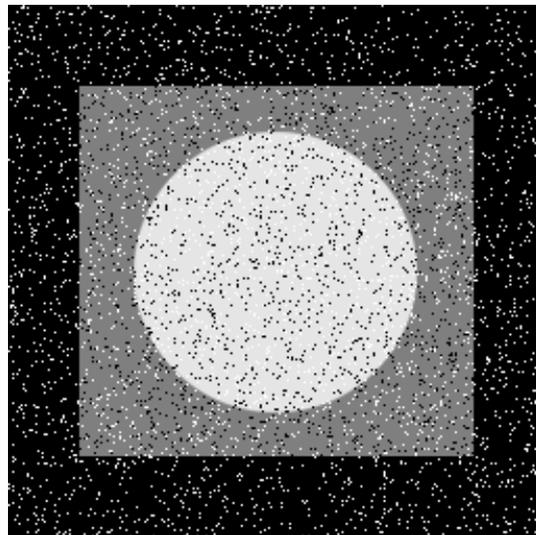


Figura 3.3: Immagine corrotta da rumore impulsivo

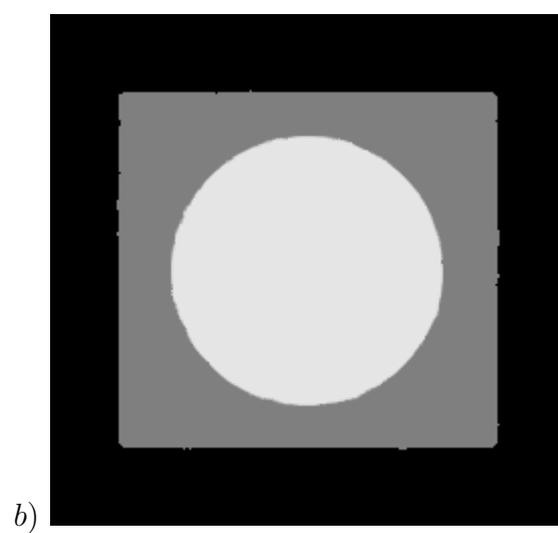
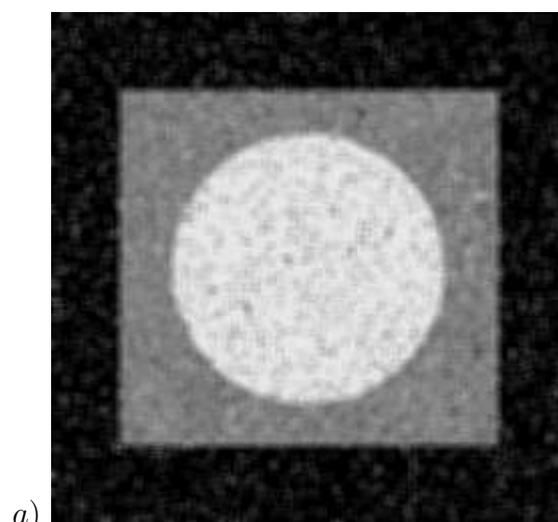


Figura 3.4: Smoothing: a) tramite filtro di media, b) tramite filtro mediano

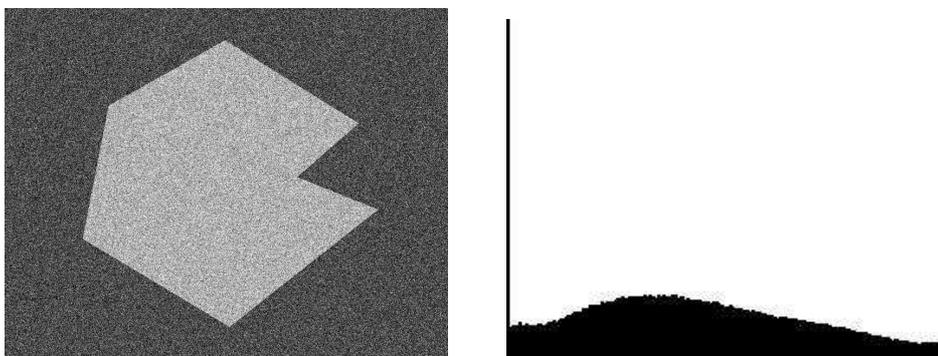


Figura 3.5: Immagine affetta da rumore e relativo istogramma

Lo smoothing è particolarmente utile anche nelle operazioni di sogliatura. Si osserva dall'istogramma in Figura 3.5, che per un'immagine affetta da rumore risulta difficile poter individuare dei valori corretti di soglia. Eseguendo lo smoothing con un filtro mediano  $5 \times 5$  si ottiene l'immagine filtrata 3.6. Notiamo che la riduzione del rumore appare evidente sia dall'immagine filtrata che dal relativo istogramma. Da quest'ultimo si può dedurre anche il valore di soglia ottimale. In particolare, con la soglia pari a 127 otteniamo l'immagine binaria 3.7.

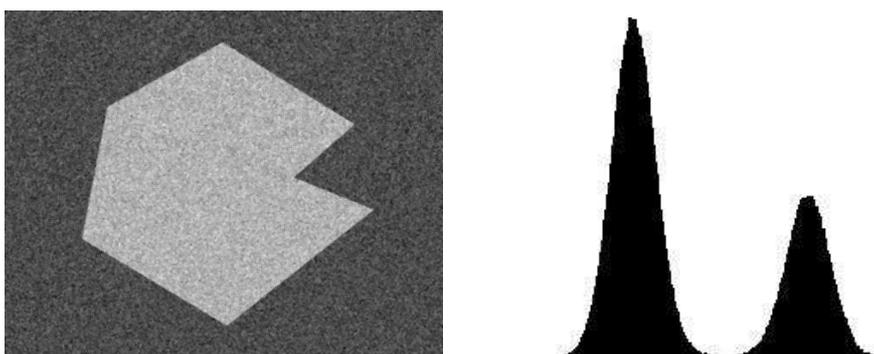


Figura 3.6: Immagine di smoothing e relativo istogramma

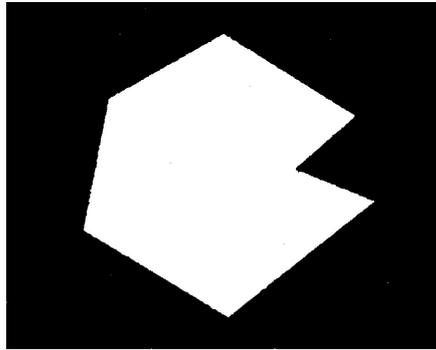


Figura 3.7: Risultato della sogliatura globale

### 3.3 Filtri derivativi

I filtri derivativi sono utilizzati principalmente nelle operazioni di *sharpening* e di *edge detection*. L'operazione di *sharpening* produce generalmente una immagine più nitida dell'originale evidenziando i dettagli offuscati durante il processo di acquisizione. L'uso dei filtri come *edge detector* permette invece di individuare i pixel di edge, ossia quei pixel in cui la funzione di intensità cambia bruscamente.

#### 3.3.1 Operatore di Sobel

Le operazioni di edge detection evidenziano quindi gli *edge* di un'immagine digitale, ossia gli insiemi di pixel di edge connessi tra loro e che costituiscono i bordi degli oggetti rappresentati. I modelli di edge sono classificati in base ai loro profili di intensità. Per determinare l'intensità e la direzione di un edge nella posizione  $(x, y)$  si utilizza il vettore gradiente il quale contiene le derivate parziali dell'immagine lungo le direzioni principali. Il gradiente è un indice della velocità di variazione della grandezza a cui si applica, appare quindi naturale ricercare i punti di edge tra quelli caratterizzati dai valori elevati del gradiente.

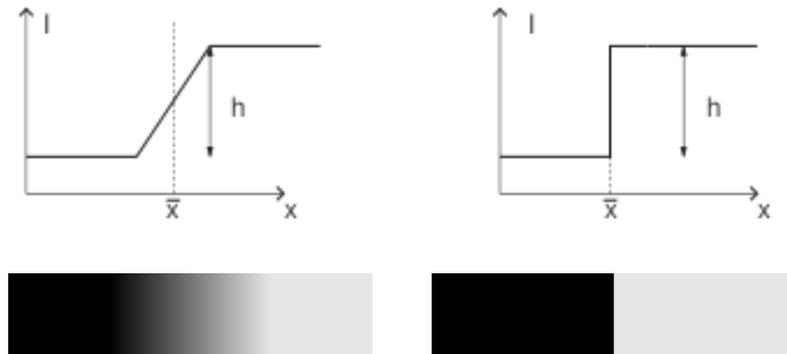


Figura 3.8: Modelli di edge ideali (Ramp Edge e Step Edge) e relativi profili di intensità

La direzione del vettore gradiente

$$\nabla f = [g_x, g_y] = \left[ \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right] \quad (3.7)$$

è quella di massima variazione di  $f(x, y)$  nel punto  $(x, y)$ . Tale direzione è data dall'angolo  $\alpha(x, y) = \arctan \left[ \frac{g_y}{g_x} \right]$  calcolato rispetto all'asse  $x$ . Il vettore ha quindi direzione normale all'edge, verso positivo andando dalla regione più scura a quella più chiara, e magnitudo  $M(x, y) = \sqrt{g_x^2 + g_y^2}$ .

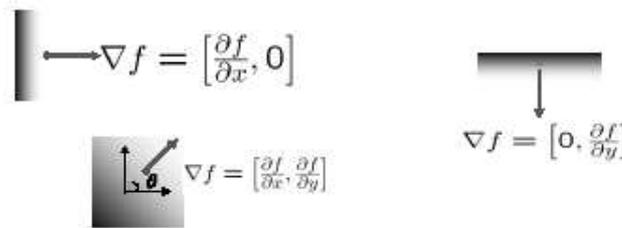


Figura 3.9: Vettore gradiente

Poichè si opera con immagini digitali, le derivate della funzione sono definite in termini di differenze locali. In corrispondenza di forti variazioni locali la derivata assume valore elevato, mentre nelle aree di intensità costante la derivata è nulla.

Discretizzando direttamente il gradiente, considerato che nelle immagini la minima distanza è di un pixel, cioè  $\Delta x = 1$ ,  $\Delta y = 1$ , si ha

$$g_x = \frac{\partial f(x, y)}{\partial x} = \frac{f(x + \Delta x, y) - f(x, y)}{\Delta x} = f(x + 1, y) - f(x, y) \quad (3.8)$$

$$g_y = \frac{\partial f(x, y)}{\partial y} = \frac{f(x, y + \Delta y) - f(x, y)}{\Delta y} = f(x, y + 1) - f(x, y) \quad (3.9)$$

Le equazioni (3.8) e (3.9) sono allora implementate dalle seguenti maschere

-1		-1	1
1			

Tabella 3.3: Maschere monodimensionali

Un limite applicativo per questo tipo di maschere è rappresentato dalla loro incapacità di individuare gli edge lungo la direzione diagonale, e dalla incapacità a ridurre l'eventuale rumore presente nell'immagine. In questi casi è possibile utilizzare maschere di convoluzione che selezionano più pixel. In caso di rumore si può assumere che questo coinvolga un pixel ma non quelli adiacenti, così elaborando più pixel alla volta il rumore presente in uno di essi si distribuisce sugli altri riducendo così il suo effetto. Per l'individuazione di edge diagonali si utilizzano maschere che hanno gli elementi delle diagonali uguali a 0. Si possono in definitiva applicare maschere  $3 \times 3$  che calcolano il gradiente del pixel centrale.

I <sub>1</sub>	I <sub>2</sub>	I <sub>3</sub>
I <sub>4</sub>	I <sub>5</sub>	I <sub>6</sub>
I <sub>7</sub>	I <sub>8</sub>	I <sub>9</sub>

Tabella 3.4: Regione  $3 \times 3$  di un'immagine con intensità  $I_i$

La differenza tra la terza e la prima riga della regione  $3 \times 3$  approssima la  $\frac{\partial f}{\partial y}$ , mentre la differenza tra la terza e la prima colonna approssima la  $\frac{\partial f}{\partial x}$ .

Nel caso specifico dell'operatore di Sobel, tali differenze si effettuano considerando il peso di valore 2 nel coefficiente centrale in modo da rimuovere anche il rumore.

$$g_x = \frac{\partial f}{\partial x} = (I_3 + 2I_6 + I_9) - (I_1 + 2I_4 + I_7) \quad (3.10)$$

$$g_y = \frac{\partial f}{\partial y} = (I_7 + 2I_8 + I_9) - (I_1 + 2I_2 + I_3) \quad (3.11)$$

L'operatore così ottenuto realizza la derivazione dell'immagine introducendo anche un leggero smoothing, e consiste in due maschere  $3 \times 3$  una per determinare la componente del gradiente nella direzione dell'asse  $x$  e l'altra nella direzione dell'asse  $y$ .

a)	<table border="1" style="border-collapse: collapse; text-align: center;"> <tr><td>-1</td><td>0</td><td>1</td></tr> <tr><td>-2</td><td>0</td><td>2</td></tr> <tr><td>-1</td><td>0</td><td>1</td></tr> </table>	-1	0	1	-2	0	2	-1	0	1	b)	<table border="1" style="border-collapse: collapse; text-align: center;"> <tr><td>-1</td><td>-2</td><td>-1</td></tr> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>2</td><td>1</td></tr> </table>	-1	-2	-1	0	0	0	1	2	1	c)	<table border="1" style="border-collapse: collapse; text-align: center;"> <tr><td>-2</td><td>-1</td><td>0</td></tr> <tr><td>-1</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>2</td></tr> </table>	-2	-1	0	-1	0	1	0	1	2	d)	<table border="1" style="border-collapse: collapse; text-align: center;"> <tr><td>0</td><td>1</td><td>2</td></tr> <tr><td>-1</td><td>0</td><td>1</td></tr> <tr><td>-2</td><td>-1</td><td>0</td></tr> </table>	0	1	2	-1	0	1	-2	-1	0
-1	0	1																																									
-2	0	2																																									
-1	0	1																																									
-1	-2	-1																																									
0	0	0																																									
1	2	1																																									
-2	-1	0																																									
-1	0	1																																									
0	1	2																																									
0	1	2																																									
-1	0	1																																									
-2	-1	0																																									

Tabella 3.5: Kernel di Sobel che individuano rispettivamente gli edge verticali e orizzontali (a)-(b); e gli edge diagonali (c)-(d)

Quindi  $g_x = \frac{\partial f}{\partial x}$  evidenzia i bordi verticali degli oggetti dell'immagine, mentre  $g_y = \frac{\partial f}{\partial y}$  quelli orizzontali. L'operatore di Sobel è implementato dalla seguente funzione, la quale permette di calcolare la derivata prima, seconda, terza e mista mediante un opportuno kernel predefinito

```
void cvSobel(const CvArr * src, CvArr * dst, int xorder,
            int yorder, int aperture_size = 3);
```

In particolare, se l'immagine sorgente ha profondità pari a 8 bit, l'immagine destinazione deve essere a 16 bit signed oppure a 32 bit floating point. Come possiamo notare dalla Tabella 3.6, l'operatore può fornire come risposta un valore esterno all'intervallo  $[0, \dots, 255]$ .

$$\begin{array}{|c|c|c|} \hline 20 & 20 & 200 \\ \hline 20 & 20 & 200 \\ \hline 20 & 20 & 200 \\ \hline \end{array} * \begin{array}{|c|c|c|} \hline -1 & 0 & 1 \\ \hline -2 & 0 & 2 \\ \hline -1 & 0 & 1 \\ \hline \end{array} = \\
 -20 - 40 - 20 + 200 + 400 + 200 = 720$$

Tabella 3.6: Risposta del filtro di Sobel verticale

Al termine dell'elaborazione, l'immagine a 16 bit può essere riconvertita a 8 bit tramite la funzione di conversione. I parametri interi,  $xorder$  e  $yorder$ , corrispondono rispettivamente all'ordine di derivazione lungo la direzione dell'asse  $x$  e dell'asse  $y$ , e il parametro  $aperture\_size$  definisce le dimensioni del filtro. In definitiva, ponendo  $xorder = 1$ ,  $yorder = 0$  e  $aperture\_size = 1$ , la funzione applica la seguente maschera, la quale rileva i bordi verticali e rappresenta il gradiente più semplice nella sua traduzione digitale, dove il nuovo valore del pixel  $i$  è calcolato in relazione al pixel  $(i - 1)$ .

$$\begin{array}{|c|c|c|} \hline -1 & 0 & 1 \\ \hline \end{array}$$

Mentre, per  $xorder = 0$ ,  $yorder = 1$  e  $aperture\_size = 1$  si ha la seguente maschera

$$\begin{array}{|c|} \hline -1 \\ \hline 0 \\ \hline 1 \\ \hline \end{array}$$

la quale permette di rilevare i bordi lungo la direzione orizzontale dell'immagine.

Con i valori  $xorder = 1$ ,  $yorder = 0$  e  $aperture\_size = 3$ , la funzione applica il filtro di Sobel verticale corrispondente alla derivata  $\frac{\partial f}{\partial x}$ . Viceversa, con i parametri  $xorder = 0$ ,  $yorder = 1$  e  $aperture\_size = 3$ , si ha il filtro orizzontale corrispondente alla derivata  $\frac{\partial f}{\partial y}$ . In questo caso il valore del campo origin della struttura IplImage è uguale a 0, ossia l'immagine è top-left-corner. Mentre, se l'immagine è bottom-left il campo origin è posto uguale a 1. Viene allora applicata la seguente maschera

$$\frac{\partial f}{\partial y} = \begin{array}{|c|c|c|} \hline 1 & 2 & 1 \\ \hline 0 & 0 & 0 \\ \hline -1 & -2 & -1 \\ \hline \end{array}$$

Ponendo  $aperture\_size = CV\_SCHARR$ , la funzione applica il *filtro di Scharr*

$$\frac{\partial f}{\partial x} = \begin{array}{|c|c|c|} \hline -3 & 0 & 3 \\ \hline -10 & 0 & 10 \\ \hline -3 & 0 & 3 \\ \hline \end{array} \quad \frac{\partial f}{\partial y} = \begin{array}{|c|c|c|} \hline -3 & -10 & -3 \\ \hline 0 & 0 & 0 \\ \hline 3 & 10 & 3 \\ \hline \end{array}$$

Tabella 3.7: Filtro di Scharr

Le derivate di ordine 2 si ottengono ponendo rispettivamente  $xorder = 2$ ,  $yorder = 0$ ; e  $xorder = 0$ ,  $yorder = 2$ .

$$\frac{\partial^2 f}{\partial x^2} = \begin{array}{|c|c|c|} \hline 1 & -2 & 1 \\ \hline 2 & -4 & 2 \\ \hline 1 & -2 & 1 \\ \hline \end{array} \quad \frac{\partial^2 f}{\partial y^2} = \begin{array}{|c|c|c|} \hline 1 & 2 & 1 \\ \hline -2 & -4 & -2 \\ \hline 1 & 2 & 1 \\ \hline \end{array}$$

Tabella 3.8: Operatore di Sobel del secondo ordine

Presentiamo ora due esempi in cui l'operatore di Sobel è impiegato come edge detector. Il primo consiste nell'individuare i bordi degli oggetti raffigurati in Figura 3.10 lungo le due direzioni verticale e orizzontale.

Il filtraggio viene eseguito con un kernel  $3 \times 3$ , ottenendo due immagini denominate *Sobel\_V* e *Sobel\_H*.

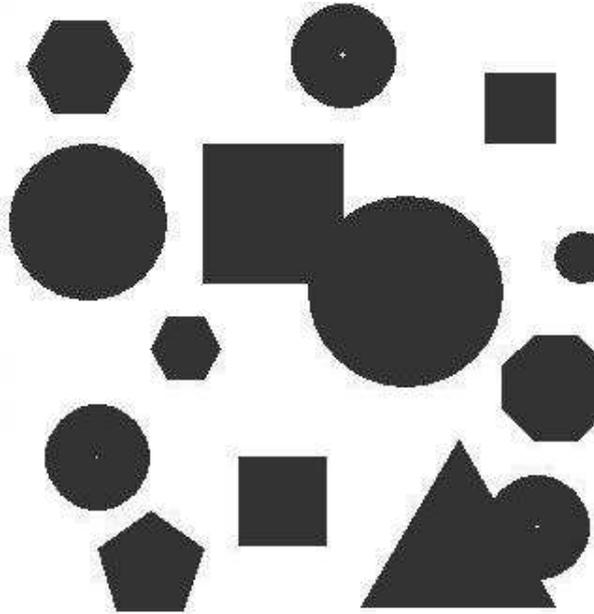


Figura 3.10: Immagine originale di  $320 \times 330$  pixel

Nel programma che realizza il primo esempio si utilizzano due immagini temporanee, le quali hanno un solo canale e profondità pari a 16 bit signed. Queste contengono il risultato delle operazioni di convoluzione del filtro verticale e orizzontale con l'immagine sorgente. La successiva conversione delle due, nelle immagini con profondità a 8 bit unsigned, si ottiene tramite la funzione

```
void cvConvertScaleAbs(const CvArr * src, CvArr * dst,  
                      double scale = 1.0, double shift = 0.0);
```

In generale, la funzione consente di convertire gli elementi di un array in elementi interi ad 8 bit unsigned. Il parametro *scale* rappresenta il fattore di scala della trasformazione, mentre *shift* è il valore costante da sommare agli elementi scalati dell'array.

L'operazione di conversione è definita nel seguente modo

$$dst = abs( src \cdot scale + (shift, shift, \dots) )$$

Ad esempio, considerando  $A$  come l'immagine a 32 bit floating point

$$A = \begin{array}{|c|c|c|} \hline 9.89949512 & 14 & 37.12142181 \\ \hline 5.83095169 & 40.52159882 & 37.12142181 \\ \hline 10.29563046 & 15.03329659 & 28.07133675 \\ \hline \end{array}$$

la funzione  $cvConvertScaleAbs( A, B, 1, 0)$  corrisponde alla trasformazione  $B = A \cdot 1$ , dove ciascun numero in virgola mobile è convertito all'intero più vicino.

$$B = \begin{array}{|c|c|c|} \hline 10 & 14 & 37 \\ \hline 6 & 41 & 37 \\ \hline 10 & 15 & 28 \\ \hline \end{array}$$

La funzione  $cvConvertScaleAbs( A, B, 1, 1)$  corrisponde invece alla trasformazione  $B = A \cdot 1 + 1$ , da cui si ottiene l'immagine seguente

$$B = \begin{array}{|c|c|c|} \hline 11 & 15 & 38 \\ \hline 7 & 42 & 38 \\ \hline 11 & 16 & 29 \\ \hline \end{array}$$

Come possiamo notare dall'immagine filtrata  $Sobel_V$  (3.11a) i segmenti di linea verticali sono evidenziati correttamente, mentre nell'immagine  $Sobel_H$  (3.11b) sono evidenziati quelli orizzontali.

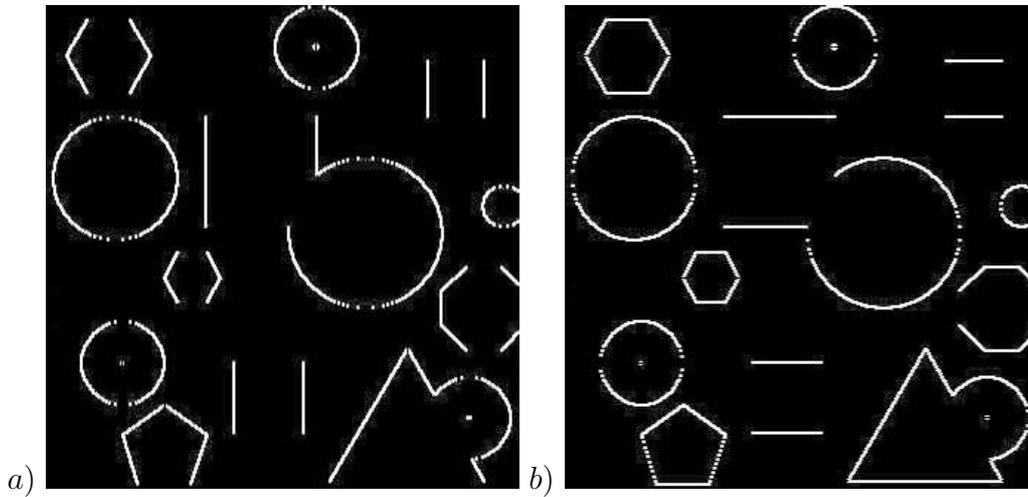


Figura 3.11: Immagine Sobel\_V e Immagine Sobel\_H

Con il secondo esempio vogliamo determinare la presenza di un edge in un punto mediante il calcolo della magnitudo del vettore gradiente. Oltre alla funzione di Sobel, necessaria per il calcolo delle derivate, usiamo le funzioni *cvPow()* e *cvAdd()*. Dalla funzione potenza *cvPow()* si ottengono le immagini  $(\frac{\partial f}{\partial x})^2$  e  $(\frac{\partial f}{\partial y})^2$  utilizzando come base i valori dei singoli pixel delle immagini  $\frac{\partial f}{\partial x}$  e  $\frac{\partial f}{\partial y}$ , e come esponente il valore 2. La funzione potenza ha il seguente prototipo

```
void cvPow(const CvArr * src, CvArr * dst, double power);
```

Con le due immagini  $\frac{\partial f}{\partial x}$  e  $\frac{\partial f}{\partial y}$  avremo rispettivamente:

$$cvPow(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial x}, 2) = (\frac{\partial f}{\partial x})^2 \quad \text{e} \quad cvPow(\frac{\partial f}{\partial y}, \frac{\partial f}{\partial y}, 2) = (\frac{\partial f}{\partial y})^2$$

Dalla funzione *cvAdd()* otteniamo invece la nuova immagine sommando i valori di intensità delle due precedenti:

$$(\frac{\partial f}{\partial x})^2 + (\frac{\partial f}{\partial y})^2 \tag{3.12}$$

Infine, dalla (3.12) si ottiene la magnitudo

$$cvPow\left(\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2, \left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2, 0.5\right)$$

dove il valore di ciascun pixel dell'immagine corrisponde al modulo del vettore gradiente nel punto.

```
...
// kernel di Sobel verticale
cvSobel (img_gray, img_tmp[0], 1, 0, 3);

// potenza della funzione img_[0] con esponente 2
cvPow(img_tmp[0], img_tmp[0], 2.0);

// kernel di Sobel orizzontale
cvSobel (img_gray, img_tmp[1], 0, 1, 3);

// potenza della funzione img_tmp[1] con esponente 2
cvPow(img_tmp[1], img_tmp[1], 2.0);

//calcolo della magnitudo
cvAdd(img_tmp[0], img_tmp[1], img_tmp[2], 0);
cvPow(img_tmp[2], img_tmp[2], 0.5);

//conversione immagini da 32 bit floating point
//a 8 bit unsigned con fattore di scala 1
cvConvertScaleAbs (img_tmp[0], img_dst[0], 1);
cvConvertScaleAbs (img_tmp[1], img_dst[1], 1);
cvConvertScaleAbs (img_tmp[2], img_dst[2], 1);
```

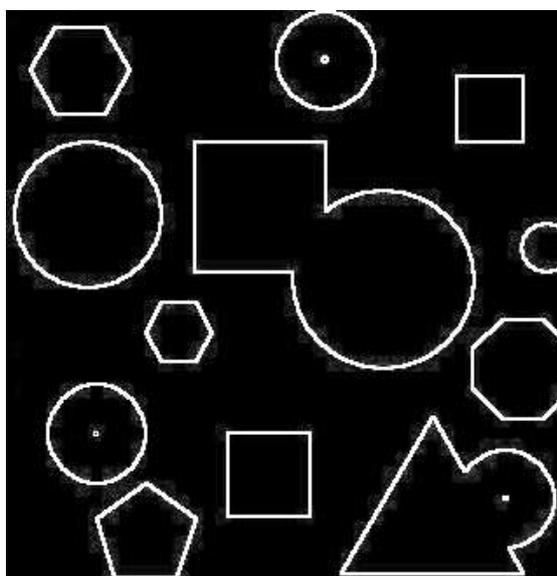


Figura 3.12: Immagine Sobel\_M

Come possiamo osservare, l'immagine filtrata Sobel\_M presenta i bordi delle figure geometriche correttamente individuati in tutte le direzioni.

### 3.3.2 Operatore Laplaciano

Il *laplaciano* si ottiene dalla formulazione discreta di derivata seconda della funzione  $f(x, y)$

$$\nabla^2 f(x, y) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \quad (3.13)$$

$$\frac{\partial^2 f}{\partial x^2} = f(x+1, y) + f(x-1, y) - 2f(x, y) \quad (3.14)$$

$$\frac{\partial^2 f}{\partial y^2} = f(x, y+1) + f(x, y-1) - 2f(x, y) \quad (3.15)$$

L'equazione laplaciana è

$$\nabla^2 f(x, y) = f(x+1, y) + f(x-1, y) + f(x, y+1) + f(x, y-1) - 4f(x, y) \quad (3.16)$$

la quale corrisponde alla seguente maschera di convoluzione

$$\nabla^2 f(x, y) = \begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline 1 & -2 & 1 \\ \hline 0 & 0 & 0 \\ \hline \end{array} + \begin{array}{|c|c|c|} \hline 0 & 1 & 0 \\ \hline 0 & -2 & 0 \\ \hline 0 & 1 & 0 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline 0 & 1 & 0 \\ \hline 1 & -4 & 1 \\ \hline 0 & 1 & 0 \\ \hline \end{array}$$

Il laplaciano si può estendere aggiungendo due termini diagonali ognuno dei quali contiene  $-2f(x, y)$ . Il termine totale da sottrarre all'equazione sarà  $-8f(x, y)$ , ottenendo il laplaciano a 8-vicini.

1	1	1
1	-8	1
1	1	1

Tabella 3.9: Laplaciano a 8-vicini

Essendo un operatore derivativo la somma dei coefficienti è zero. Ciò significa che la risposta sarà nulla nelle aree ad intensità costante, e diversa da zero nelle aree che presentano discontinuità, tipo linee o punti isolati. Il laplaciano può essere utilizzato nelle operazioni di miglioramento della qualità dell'immagine evidenziando e rendendo più nitidi i dettagli. In particolare, l'azione dell'operatore restituisce un'immagine contenente solo le discontinuità rilevate, provocando la perdita dell'informazione di sfondo. Il ripristino di tale informazione si ottiene sommando l'immagine filtrata con l'originale, ossia

$$g(x, y) = \begin{cases} f(x, y) - \nabla^2 f(x, y) & \text{se il coefficiente centrale è } < 0 \\ f(x, y) + \nabla^2 f(x, y) & \text{se il coefficiente centrale è } > 0 \end{cases} \quad (3.17)$$

Dalla (3.17) si ottiene in particolare l'equazione

$$g(x, y) = f(x, y) - [f(x+1, y) + f(x-1, y) + f(x, y+1) + f(x, y-1)] + 4f(x, y) = \\ 5f(x, y) - [f(x+1, y) + f(x-1, y) + f(x, y+1) + f(x, y-1)] \quad (3.18)$$

0	-1	0
-1	5	-1
0	-1	0

Tabella 3.10: Maschera che implementa l'equazione 3.18

Analogamente, se consideriamo anche i termini diagonali si ottiene la maschera rappresentata in Tabella 3.11. Queste maschere, la cui somma dei coefficienti è pari a 1, producono sull'immagine un aumento del contrasto locale evidenziando così i dettagli più fini. In ogni caso i filtri del secondo ordine operano aumentando le differenze tra i pixel vicini, e per accrescere tali differenze devono avere i coefficienti centrali di segno opposto a quelli periferici. L'utilizzo del laplaciano come edge detector è giustificato dal fatto che la derivata prima e seconda di un profilo siano diverse da zero solo in corrispondenza alle transizioni. In particolare la derivata seconda è positiva in prossimità della parte scura di un bordo, negativa dalla parte chiara e nulla nelle zone a livello di grigio costante. Ammette un passaggio per lo zero (*zero crossing*) esattamente in corrispondenza delle transizioni, e ciò consente la localizzazione degli edge. I limiti del laplaciano consistono in una eccessiva sensibilità al rumore in quanto operatore di derivata seconda, e nell'incapacità di rilevare la direzione dei bordi in quanto entità scalare. La funzione di libreria che consente di calcolare il laplaciano dell'immagine mediante operatori predefiniti è la seguente

```
void cvLaplace( const CvArr * src, CvArr * dst, int aperture_size = 3);
```

Le due immagini, sorgente e destinazione, devono avere entrambe un solo canale. Il parametro *aperture\_size* definisce le dimensioni del filtro, le quali possono essere 3, 5, 7. In particolare con il valore posto a 1 la funzione applica quello relativo all'equazione (3.16). Ovviamente l'azione del laplaciano genera un'immagine contenente anche valori negativi, quindi per un'immagine di input a 8 bit quella di output deve essere a 16 bit signed oppure a 32 bit floating point. La riconversione a 8 bit si può ottenere dalla funzione

*cvConvertScaleAbs()*, la quale converte il valore di ciascun pixel in valore assoluto e satura a 255 tutti i pixel ad esso superiore. Consideriamo ora l'operazione di sharpening, la quale evidenzia i dettagli fini dell'immagine e le variazioni di luminosità sommando la risposta del filtro all'immagine originale. Avremo nel nostro caso:

$$g(x, y) = f(x, y) + \nabla^2 f(x, y) = \begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & 0 & 0 \\ \hline \end{array} + \begin{array}{|c|c|c|} \hline -1 & -1 & -1 \\ \hline -1 & 8 & -1 \\ \hline -1 & -1 & -1 \\ \hline \end{array}$$

Tabella 3.11: Maschera di Sharpening



Figura 3.13: Immagine originale e di sharpening

Il filtro viene costruito e applicato tramite la già nota *cvFilter2D()*. Come si può notare l'immagine di sharpening 3.13b risulta notevolmente migliorata rispetto all'originale, i dettagli sono più nitidi e la tonalità dello sfondo è rimasta invariata.

Utilizziamo ora il laplaciano come edge detector. Il filtro che agisce sull'immagine 3.14 corrisponde all'equazione (3.16). Poichè dal filtraggio si ottiene

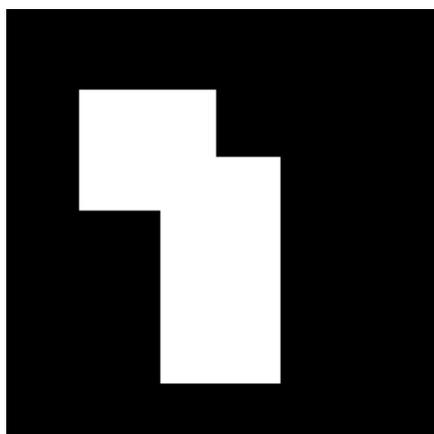


Figura 3.14: Immagine originale di  $256 \times 256$  pixel

un'immagine contenente sia pixel a valori positivi che negativi, per consentirne la visualizzazione è necessario che tutti i valori siano non negativi. Otteniamo la Figura 3.15a prendendo il valore assoluto dell'immagine laplaciana, mentre la 3.15b si ottiene prendendo solo i valori positivi.

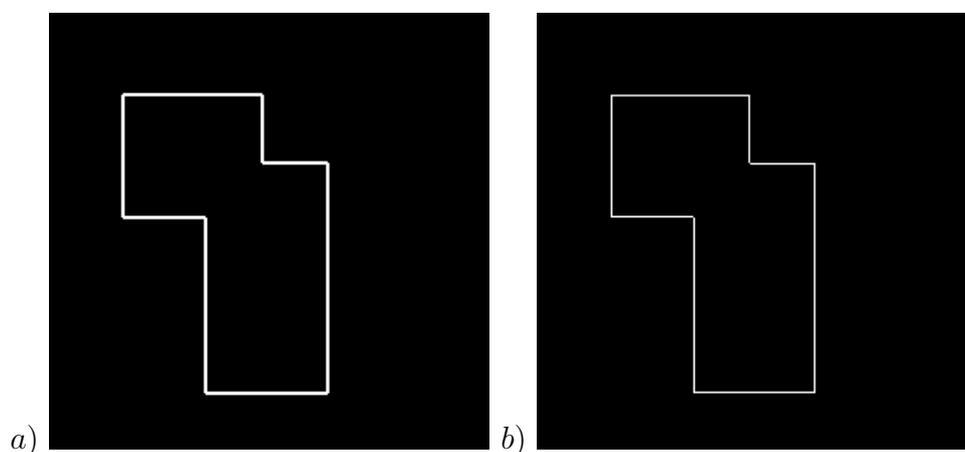


Figura 3.15: Valore assoluto dell'immagine laplaciana e Valori positivi

Le due immagini differiscono solo per lo spessore delle linee individuate, infatti come era prevedibile quella in valore assoluto presenta delle linee di spessore doppio rispetto all'immagine a valori positivi.

## 3.4 Operatore di Canny

L'operatore di Canny è utilizzato nel campo dell'elaborazione delle immagini come edge detector. Gli obiettivi che il metodo si propone sono:

- Buon riconoscimento. L'algoritmo deve individuare e marcare il maggior numero possibile di edge presenti nell'immagine.
- Buona localizzazione. I punti marcati come edge devono essere il più vicino possibile al centro di un edge reale.
- Risposta minima. Un edge deve essere marcato una sola volta. Quando si hanno due risposte per uno stesso edge, uno dei due deve essere considerato falso.

L'algoritmo proposto da Canny consiste in quattro passi. Il primo passo, detto *Image Smoothing*, effettua lo smoothing dell'immagine di input mediante un filtro Gaussiano. Questo influenza i risultati generati dall'algoritmo, poichè filtri di piccole dimensioni producono una minore sfocatura e consentono di riconoscere dettagli più fini, mentre filtri più grandi producono una maggiore sfocatura e sono indicati per riconoscere dettagli più ampi. La *differenziazione* è il passo successivo dell'algoritmo e permette di calcolare il gradiente dell'immagine ottenuta dal filtraggio gaussiano. Ne segue il calcolo della magnitudo  $M(x, y)$  e dell'angolo  $\alpha(x, y)$  del vettore gradiente. Il terzo passo consiste nella *soppressione dei non massimi*. Per un pixel di edge, il valore del gradiente nel punto è superiore ai valori dei vicini nella direzione del gradiente stesso. Per ogni punto bisogna quindi individuare la direzione del gradiente e confrontare il suo modulo con quello dei vicini giacenti lungo la direzione stessa. Se almeno uno dei due vicini ha modulo maggiore del pixel in esame, questo viene soppresso ponendo il modulo a zero. In una regione  $3 \times 3$  si possono definire quattro orientazioni per un edge passante attraverso il punto centrale, ossia la direzione orizzontale, verticale e le due diagonali ( $45^\circ$  e  $135^\circ$ ). La direzione dell'edge si ottiene quindi dalla direzione del vettore gradiente, la quale a sua volta è ottenuta dal valore angolare  $\alpha(x, y)$ . L'angolo del gradiente sarà quindi contenuto in uno dei quattro settori rappresentati in Figura 3.16.

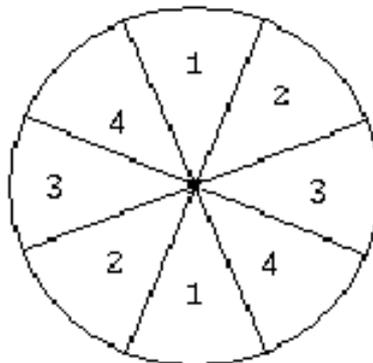


Figura 3.16: Intervalli delle normali agli edge in un intorno  $3 \times 3$

In particolare, se il vettore ha valore angolare compreso nel settore 1 si ha un edge orizzontale, nel settore 3 si ha un edge verticale, e infine nei rimanenti un edge diagonale. Nel quarto passo, *edge thresholding*, si esegue la sogliatura dell'immagine ottenuta al passo precedente. Si devono definire due soglie, una bassa  $T_L$  ed una alta  $T_H$ , le quali sono confrontate con i punti dell'immagine. Se il valore è inferiore alla soglia bassa  $T_L$ , il punto viene scartato, se è superiore alla soglia alta  $T_H$ , il punto viene accettato come edge, se invece è compreso tra le soglie il punto è accettato solo se contiguo ad un punto precedentemente accettato. L'uso delle soglie  $T_L$  e  $T_H$  permette di ridurre la presenza di falsi edge rispetto ad una soglia singola  $T$ . Utilizzando una soglia singola si rischia infatti di settarla ad un valore troppo basso ottenendo dei falsi edge, in questo modo un semplice disturbo potrebbe essere interpretato come un elemento importante dell'immagine. Se invece la soglia è settata ad un valore troppo alto, punti di edge validi potrebbero essere eliminati provocando in questo modo la perdita di informazione significativa. Canny suggerisce che i risultati migliori si ottengono scegliendo il valore  $T_H$  due o tre volte più grande del valore  $T_L$ . In pratica la soglia  $T_H$  serve a localizzare le strutture significative dell'immagine, mentre quella inferiore serve a connetterle. In OpenCV tale metodo è implementato dalla seguente funzione, la quale prende in input un'immagine a toni di grigio e restituisce

come output un'immagine binaria in cui i pixel non nulli marcano gli edge individuati.

```
void cvCanny( const CvArr * image, CvArr * edges, double threshold1,  
             double threshold2, int aperture_size = 3 );
```

Il parametro *image* è l'immagine da elaborare, mentre il parametro *edges* è l'immagine che dovrà contenere gli edge individuati. Entrambe devono essere a un solo canale e con profondità pari a 8 bit unsigned. I parametri *threshold1* e *threshold2* sono rispettivamente la soglia inferiore e superiore. La funzione è stata progettata in modo da eseguire un controllo sui valori di soglia. In questo modo se *threshold1* è maggiore di *threshold2* l'errore viene corretto mediante lo scambio dei valori. Il parametro *aperture\_size* corrisponde alle dimensioni dell'operatore di Sobel, il quale è impiegato dalla funzione per eseguire le operazioni di smoothing e di differenziazione previste dall'algoritmo. Alle immagini di Figura 3.17 e 3.20 abbiamo applicato rispettivamente un filtro di Canny di dimensione  $3 \times 3$  in modo da evidenziare gli edge dell'oggetto di scena. L'operazione ottimale consiste nell'estrarre un bordo che sia il più continuo possibile e che delimiti al meglio la forma dell'oggetto. I risultati migliori si sono ottenuti con le soglie  $T_L$  e  $T_H$  settate rispettivamente ai valori 60 e 120 per l'immagine 3.17, e ai valori 45 e 150 per l'immagine 3.20.



Figura 3.17: Immagine originale di  $256 \times 256$  pixel

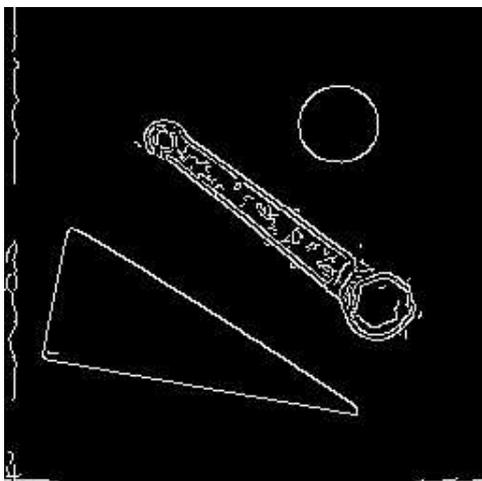


Figura 3.18: Risultato dell'operatore di Canny:  $T_l = 30$  e  $T_h = 70$

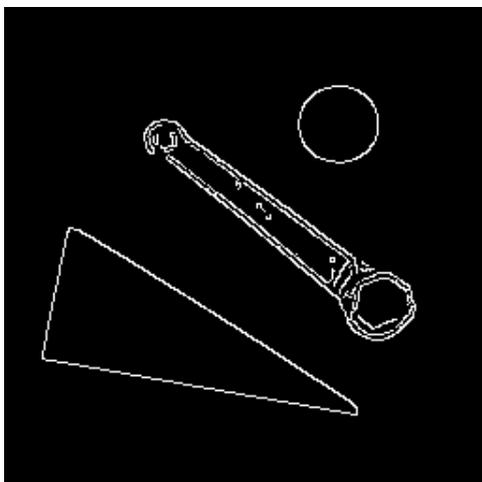


Figura 3.19: Risultato ottimale dell'operatore di Canny:  $T_l = 60$  e  $T_h = 120$

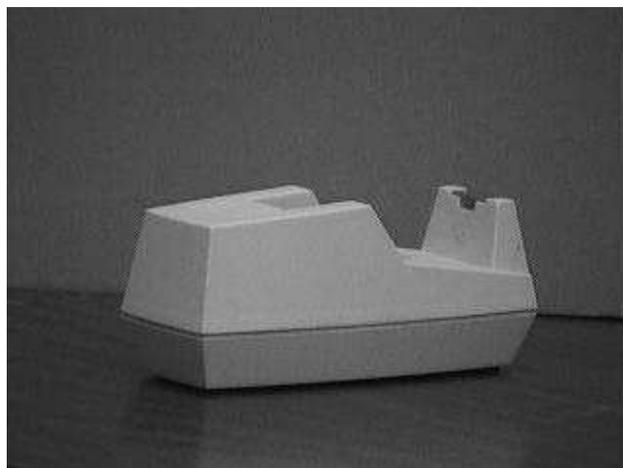


Figura 3.20: Immagine originale di  $319 \times 239$  pixel

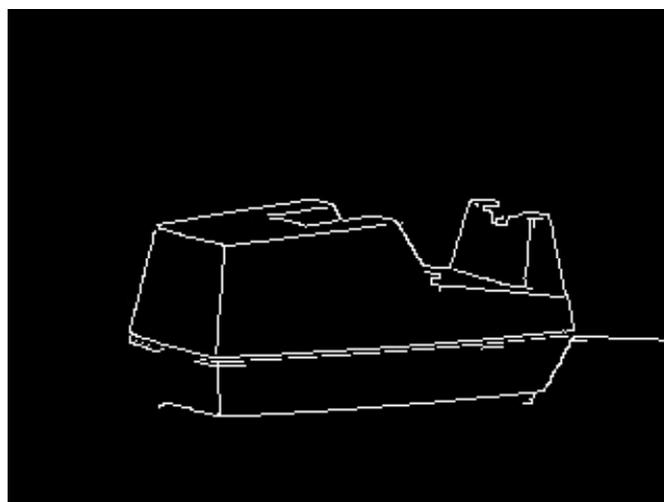


Figura 3.21: Risultato ottimale dell'operatore di Canny:  $T_l = 45$  e  $T_h = 150$

## 3.5 Trasformata di Hough

La *trasformata di Hough* (*HT*) permette di stabilire se particolari insiemi di pixel si trovano oppure no su una specifica curva che costituisce i contorni di interesse di un'immagine. Si può impiegare come *edge linking*, ossia come metodo che consente di collegare tra loro i pixel di edge. In generale la *HT* permette il riconoscimento di configurazioni globali come segmenti, curve e forme prestabilite, trasformando i punti dello spazio immagine in punti del nuovo spazio detto *spazio dei parametri*. La tecnica è basata sulla validazione delle ipotesi. Definita la curva che si intende cercare nella scena, per ogni punto si calcolano i parametri relativi alle curve che potrebbero passare per quel punto. Si ottiene quindi una funzione di accumulazione definita nello spazio dei parametri. Saranno i massimi di questa funzione, ovvero i punti nello spazio dei parametri che hanno accumulato il maggior numero di voti, a rappresentare le curve che hanno maggiore probabilità di essere presenti nell'immagine.

### 3.5.1 Trasformata di Hough per le linee

Nella ricerca delle rette si considera inizialmente un punto  $(x', y')$  nel piano cartesiano  $xy$ , detto *spazio immagine*, e la generica retta di equazione

$$y' = mx' + c \quad (3.19)$$

passante per il punto. L'equazione, al variare di  $m$  e  $c$ , rappresenta tutte le possibili rette del piano passanti per il punto dato. È possibile ottenere una legge che dal punto  $(x', y')$  dello spazio immagine consente di tracciare una curva nello *spazio dei parametri*  $mc$ . L'equazione 3.19 si può scrivere come

$$c = -mx' + y' \quad (3.20)$$

dove  $x'$  e  $y'$  sono costanti mentre  $m$  e  $c$  sono variabili.

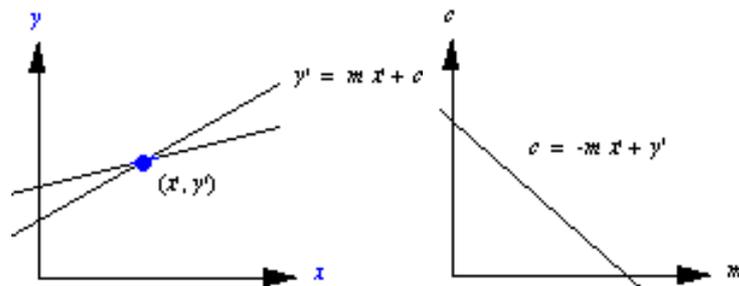
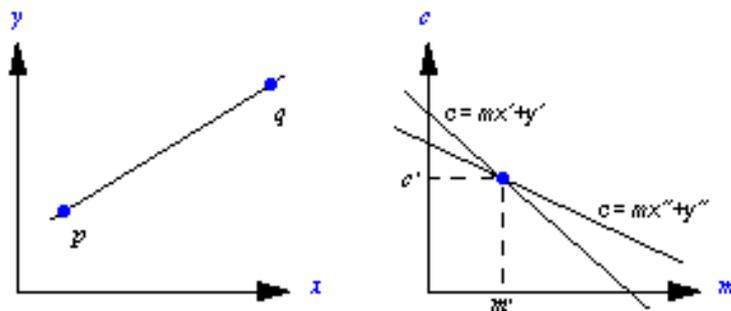


Figura 3.22: Spazio immagine e spazio dei parametri

Considerati due punti  $P = (x', y')$  e  $Q = (x'', y'')$  di una stessa retta nello spazio immagine  $xy$ , avremo che ogni retta di tale spazio è rappresentata da un punto nello spazio dei parametri  $mc$ , e per ogni punto  $P$  nello spazio  $xy$  il fascio di rette passanti per  $P$  è rappresentato da una singola retta nello spazio  $mc$ . Nello spazio dei parametri  $mc$  la retta 3.20 descrive i valori  $mc$  relativi alle rette passanti per  $P$ , mentre la retta

$$c = -mx'' + y'' \quad (3.21)$$

descrive i valori di  $mc$  relativi alle rette passanti per  $Q$ . Il punto di intersezione  $(m', c')$  delle due rette nello spazio  $mc$  descrive i valori dei parametri  $(m, c)$  della retta passante per  $P$  e  $Q$  nello spazio immagine  $xy$ .

Figura 3.23: Punti e rette nello spazio immagine  $xy$  e nello spazio dei parametri  $mc$

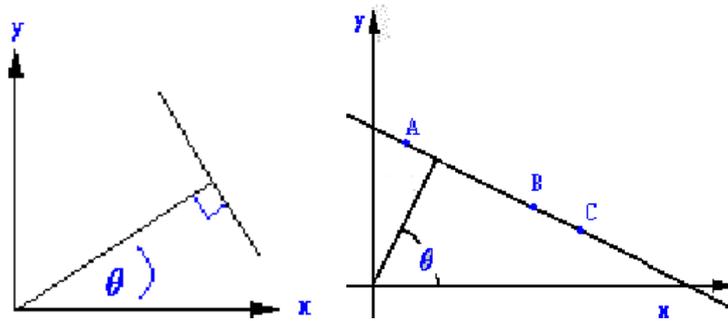
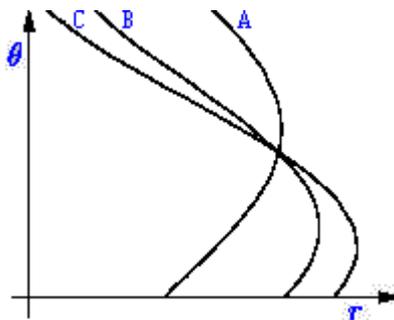


Figura 3.24: Spazio immagine e punti collineari

Ai fini pratici questo tipo di approccio non è adeguato, in quanto man mano che la retta si avvicina alla direzione verticale il parametro  $m$  tende ad infinito. Del resto imporre una limitazione a tale parametro significa ridurre le possibili rette riconoscibili. Un modo per superare tale inconveniente è quello di adottare la rappresentazione normale

$$\rho = x \cos \theta + y \sin \theta \quad (3.22)$$

dove  $\rho$  è la distanza (limitata dalla diagonale dell'immagine) dalla retta all'origine, mentre  $\theta \in [0, 2\pi]$  è l'angolo tra la normale alla retta dall'origine e l'asse  $x$ . Quindi, le sinusoidi che si intersecano in un punto del piano  $\rho\theta$  corrispondono ad altrettanti punti allineati del piano  $xy$ .

Figura 3.25: Spazio  $\rho\theta$  di Hough

La *HT* prevede la suddivisione dello spazio dei parametri  $\rho\theta$  in celle di accumulazione, con i parametri che possono assumere i valori  $-90^\circ \leq \theta \leq 90^\circ$  e  $-d \leq \rho \leq d$ , in cui  $d$  è la distanza tra due vertici opposti dell'immagine. La procedura per l'estrazione delle rette inizia ponendo a zero tutte le celle dell'accumulatore. Per ogni pixel di edge  $(x_p, y_q)$  nello spazio immagine, e con  $\theta$  corrispondente a ciascuno dei valori di suddivisione consentiti, si calcola

$$\rho = x_p \cos\theta + y_q \sin\theta$$

approssimando i valori di  $\rho$  a quello della cella più vicina lungo l'asse stesso. Si incrementa poi  $A(m, n)$  di una unità quando la scelta di  $\theta_n$  produce la soluzione  $\rho_m$ . Si determinano infine i valori più elevati presenti nell'accumulatore. Un punto di massimo corrisponde ad una retta nello spazio immagine. La *HT* consente di individuare la forma di interesse anche in presenza di rumore sui dati di input. È infatti improbabile che dei punti casuali dovuti proprio al rumore si accumulino in modo coerente nell'accumulatore producendo false evidenze della forma da individuare.

In OpenCV la funzione relativa alla trasformata di Hough per le linee implementa non solo l'algoritmo standard *SHT* appena visto, ma anche due sue varianti: progressive probabilistic Hough transform (*PPHT*) e multiscale variant of classical Hough transform (*MHT*). La funzione ha il seguente prototipo

```
cvHoughLines2(CvArr * image, void * line_storage,  
int method, double rho, double theta, int threshold,  
double param1 = 0, double param2 = 0);
```

Il parametro *image* è l'immagine di input a 8 bit con un solo canale, la quale viene considerata come binaria. Il secondo parametro, *line\_storage*, è la struttura dati necessaria a contenere le linee individuate. La struttura dati può essere la matrice  $N \times 1$ , dove  $N$  corrisponde al numero massimo di linee restituite dalla funzione, oppure la struttura *CvSeq* che permette



Figura 3.26: Immagine originale affetta da rumore

di allocare dinamicamente i dati in una sequenza di elementi(es: deque). Mediante i parametri  $\rho$  e  $\theta$  si definiscono le dimensioni delle celle che caratterizzano l'accumulatore bidimensionale, mentre  $threshold$  rappresenta la soglia rispetto alla quale la funzione restituisce una linea. Ponendo il parametro  $method$  al valore `CV_HOUGH_STANDARD` viene eseguito l'algoritmo *SHT*. Con `CV_HOUGH_PROBABILISTIC` si esegue invece il *PPHT* che è una versione randomizzata del precedente, dove l'accumulatore viene aggiornato dai singoli pixel selezionati in modo casuale. Questo metodo, diversamente dall'*SHT*, restituisce segmenti di linea invece che rette. Il suo obiettivo è quello di ridurre al minimo i punti utilizzati nella fase di voto con conseguente riduzione del tempo di calcolo e della quantità di memoria impiegata. Con il valore `CV_HOUGH_MULTISCALE` la funzione esegue l'algoritmo *MHT*. Le linee sono determinate allo stesso modo del metodo classico, ma in questo caso si utilizzano anche i parametri  $param1$  e  $param2$  che permettono di ottenere un livello di accuratezza maggiore nella suddivisione dell'accumulatore in celle. Inizialmente si calcolano le posizioni delle linee con precisione

data da  $\rho$  e  $\theta$ , per poi affinare ulteriormente i risultati mediante  $param1$  e  $param2$ , ottenendo così  $\frac{\rho}{param1}$  e  $\frac{\theta}{param2}$  come dimensioni delle celle. Gli argomenti  $param1$  e  $param2$  della funzione dipendono quindi dal metodo impiegato. Nell'algoritmo *SHT* tali parametri non vengono utilizzati, viceversa nel *PPHT* il valore  $param1$  corrisponde alla lunghezza minima del segmento di linea che deve essere restituito, mentre  $param2$  corrisponde al massimo gap che può intercorrere tra due segmenti di linea affinché l'algoritmo li colleghi formando un segmento unico.

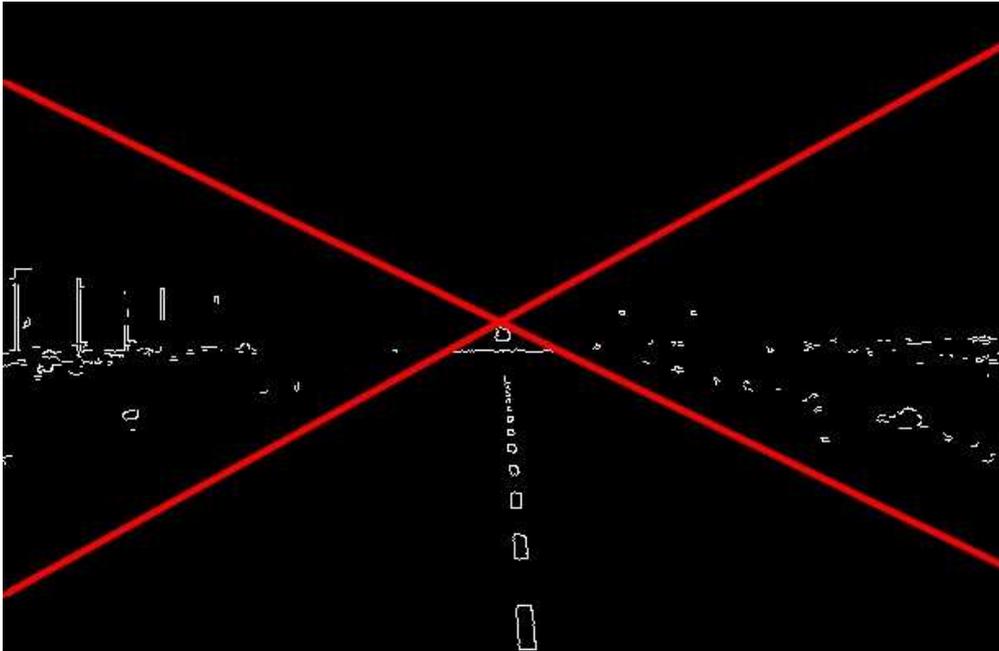


Figura 3.27: Trasformazione SHT

Consideriamo ora l'immagine di Figura 3.26 caratterizzata da una distribuzione uniforme di rumore. Applichiamo la trasformata di Hough in modo da estrarre i due edge della strada. A questo scopo si esegue preliminarmente un filtraggio aggressivo mediante l'operatore di Canny. In Figura 3.27 abbiamo il risultato relativo alla trasformata di Hough standard (*SHT*).

In realtà l'effetto del disturbo si riflette anche nello spazio dei parametri il quale diventa più denso di punti rispetto alla stessa immagine priva di rumore. Nonostante ciò i punti di massimo tra le due immagini sono gli stessi. L'operazione è ripetuta sull'immagine di input utilizzando il metodo *PPHT*. Possiamo osservare nella 3.28 che anche in questo caso gli edge sono correttamente individuati, ma a differenza del metodo precedente sono marcati con segmenti di linea.

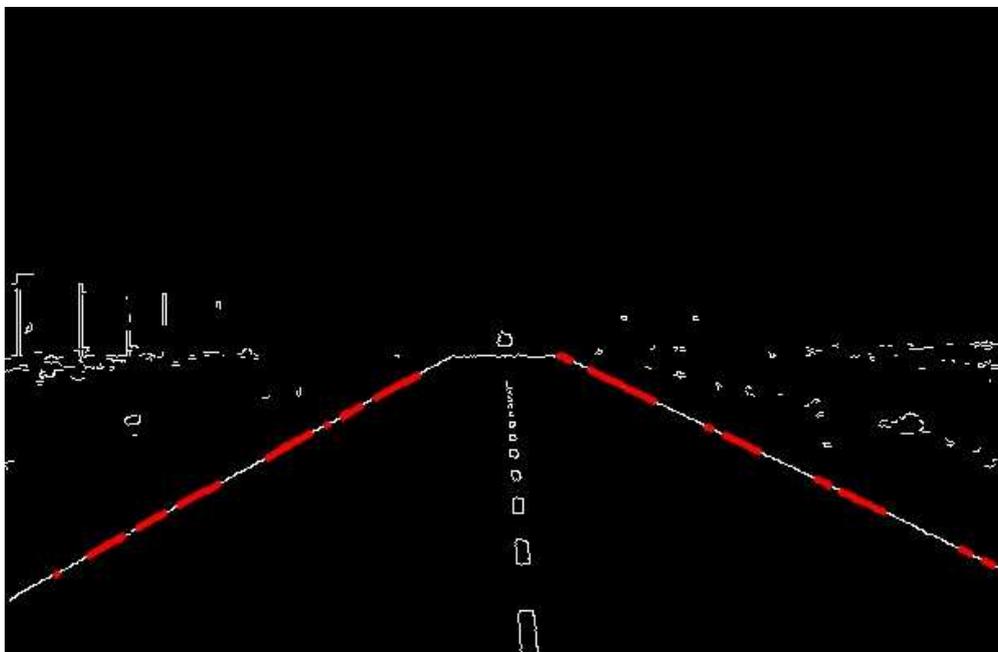


Figura 3.28: Trasformazione PPHT

### 3.5.2 Trasformata di Hough per forme circolari

La trasformata di Hough può essere usata anche per il riconoscimento di forme geometriche circolari

$$(x - a)^2 + (y - b)^2 = r^2 \quad (3.23)$$

Il principio base è analogo, con qualche modifica, a quello visto per il riconoscimento delle rette. In tal caso si hanno due parametri  $a$ ,  $b$  per il centro del cerchio e un parametro  $r$  per il raggio. Il fatto di avere una terna di parametri  $(a, b, r)$  comporta che l'array accumulatore deve essere tridimensionale  $A(i, j, k)$ . Se la tecnica di Hough è impiegata per localizzare circonferenze di raggio noto  $r_0$ , allora lo spazio dei parametri si riduce a due dimensioni, essendo necessario stimare solo le coordinate dei centri. Per ogni punto  $(x, y)$  dello spazio immagine corrisponderà, nello spazio dei parametri, un cerchio di raggio  $r_0$  avente centro proprio nel punto  $(x, y)$ . Come si può osservare dalla Figura 3.29 ogni punto dello spazio immagine genera un cerchio nello spazio dei parametri. I cerchi nello spazio dei parametri si intersecano nel punto  $(a, b)$ , il quale corrisponde al centro del cerchio nello spazio immagine. Il centro di un cerchio è quindi il punto comune a tutte quelle circonferenze di raggio  $r_0$  centrate su un punto qualsiasi  $(x, y)$  del suo perimetro.

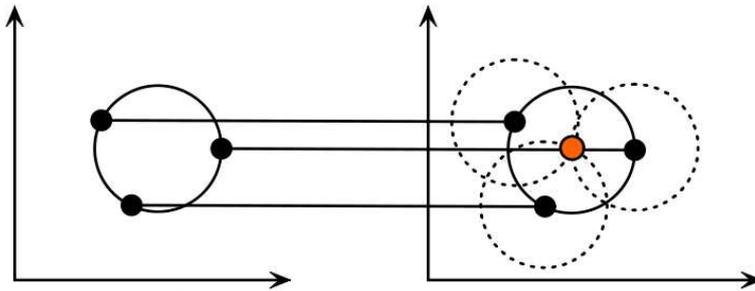


Figura 3.29: Rappresentazione dei cerchi nello spazio immagine e nello spazio dei parametri.

Se il raggio non è noto a priori, ma è compreso in un range di valori ammissibili, ogni punto di coordinate  $(x, y)$  nello spazio immagine genera un cono nello spazio dei parametri. In questo caso, per tutti i punti dello spazio immagine appartenenti ad una circonferenza di raggio  $r$ , la terna  $(x, y, r)$  da stimare si troverà in corrispondenza della cella accumulatrice in cui si interseca il maggior numero di coni. Questo si traduce in una richiesta maggiore

di memoria (l'array accumulatore è tridimensionale). Un modo per ridurre la richiesta di memoria consiste nell'utilizzare il cosiddetto *metodo del gradiente*, ossia nello sfruttare l'informazione del gradiente lungo la direzione di un edge. Inizialmente sull'immagine di input si esegue un'operazione di edge detection, generalmente mediante l'operatore di Canny. Successivamente, per ogni punto non nullo dell'immagine filtrata si determina il gradiente locale, utilizzando ad esempio l'operatore di Sobel. Poiché un edge è perpendicolare alla direzione del vettore gradiente nel punto in cui il gradiente viene calcolato, il centro di un cerchio sarà dato dalle intersezioni delle rette normali alla circonferenza. Un accumulatore bidimensionale conterrà i voti dei candidati centri, mentre il raggio del cerchio si ottiene dal calcolo della distanza dai pixel di edge al centro del cerchio.

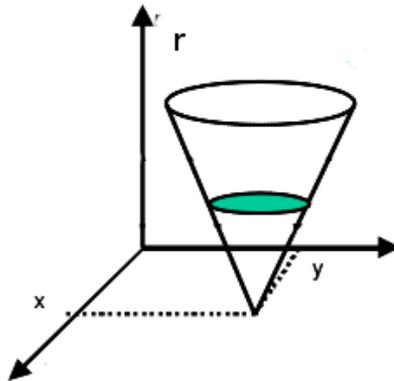


Figura 3.30: Superficie conica nello spazio dei parametri.

Il prototipo della funzione per le forme circolari è

```
cvHoughCircles(CvArr * image, void * circle_storage, int method,
```

```
double dp, double min_dist, double param1 = 100,
```

```
double param2 = 300, int min_radius = 0, int max_radius = 0);
```

Anche in questo caso l'immagine di input deve essere a 8 bit, ma diversamente dalla funzione di Hough per le linee non viene considerata come binaria. Il

parametro *circle\_storage* è la struttura dati necessaria a contenere la forma circolare. Il parametro *method* ammette attualmente solo il valore predefinito CV\_HOUGH\_GRADIENT che definisce il metodo del gradiente. Se il parametro *dp* è impostato ad un valore maggiore di 1 permette di creare un accumulatore con risoluzione inferiore, di tale fattore, all'immagine sorgente. Il parametro *min\_dist* rappresenta la distanza minima che deve intercorrere tra due cerchi affinché l'algoritmo li consideri come distinti. Gli argomenti *param1* e *param2* sono rispettivamente la soglia utilizzata nell'algoritmo di Canny e la soglia utilizzata dall'accumulatore. L'operatore di Canny, invocato internamente dalla funzione di Hough, utilizza due differenti soglie. Quindi il valore *param1* corrisponde alla soglia alta mentre la soglia bassa corrisponde esattamente alla metà del valore di *param1*. Il parametro *param2* è invece la soglia dell'accumulatore e ha significato analogo all'argomento *threshold* della funzione di Hough per le linee. Gli ultimi due parametri sono rispettivamente il raggio minimo e il raggio massimo rispetto ai quali è ammessa una rappresentazione del cerchio nell'accumulatore.

Nell'immagine di Figura 3.9, è possibile individuare le forme circolari mediante la *HT*. In genere è necessario eseguire lo smoothing in modo da limitare la rilevazione di falsi cerchi. Nel caso specifico abbiamo utilizzato un filtro gaussiano di dimensione  $11 \times 11$  ottenendo l'immagine filtrata 3.31.



Figura 3.31: Smoothing con filtro gaussiano  $11 \times 11$

Dalla seguente funzione di Hough si ottiene l'immagine 3.32, dove le forme circolari sono correttamente individuate.

```
cvHoughCircles(src, storage, CV_HOUGH_GRADIENT,  
3, src->height/10, 100, 100, 1, 100);
```

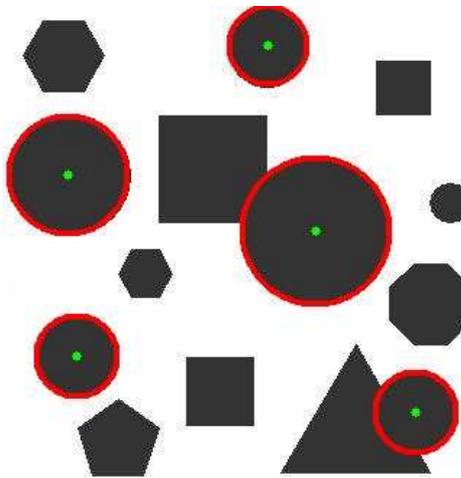


Figura 3.32: *HT* che individua le forme circolari

Lo smoothing dell'immagine sorgente può essere a volte controproducente ai fini di una corretta individuazione delle forme. Effettuando un filtraggio più aggressivo, ad esempio con un filtro di media di dimensioni  $15 \times 15$ , si ottiene la 3.33. Se applichiamo a questa immagine la stessa funzione di Hough definita nell'esempio precedente otteniamo la 3.34. Anche in questo caso le forme circolari sono individuate in modo corretto, ma vengono considerati come cerchi anche delle forme che non sono tali nell'immagine sorgente. Questo è dovuto all'operazione di smoothing che ha prodotto una deformazione eccessiva delle forme poligonali.



Figura 3.33: Smoothing con filtro di media  $15 \times 15$

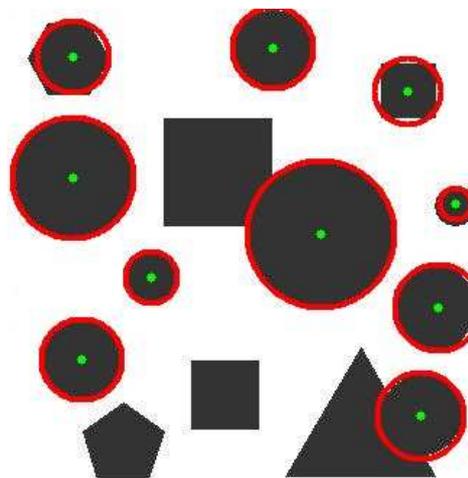


Figura 3.34: *HT* che individua alcune forme in modo non corretto

# Capitolo 4

## Morfologia

### 4.1 Operazioni morfologiche

La *morfologia matematica* consiste nelle operazioni di image processing che permettono di estrarre e più in generale modificare le componenti di una immagine. Un'operazione base come l'*erosione* può essere utilizzata per eliminare o assottigliare gli oggetti di scena, mentre la *dilatazione* può essere utilizzata per accrescere gli oggetti o riempire delle aree vuote. In morfologia un'immagine binaria è considerata come un sottoinsieme dell'insieme  $2D$  dei numeri interi, in questo modo un'insieme  $A$  di pixel a valore 1 costituisce l'oggetto all'interno di un'immagine mentre i pixel a valore 0 costituiscono gli elementi di sfondo. Gli oggetti sono manipolati mediante una forma elementare costituita da un piccolo insieme  $B$  detto *elemento strutturante* ( $SE$ ), nel quale è specificato un punto che costituisce la sua *origine*. Gli  $SE$  possono avere qualunque forma, ma si utilizzano prevalentemente quelli simmetrici con origine coincidente con il centro di simmetria. Per le operazioni morfologiche in scala di grigio si considerano le funzioni  $f(x, y)$  come immagine e  $b(s, t)$  come elemento strutturante, entrambe assegnano un valore di intensità per ogni coppia diversa di coordinate intere  $(x, y)$ . L'elemento strutturante ha le stesse proprietà del corrispondente  $SE$  per un'immagine binaria, e nonostante sia definito come una quantità continua la sua implementazione

0	1	0	1	1	1	1
1	1	1	1	1	1	1
0	1	0	1	1	1	1

Tabella 4.1: SE a croce, quadrato e rettangolare

su computer si basa sempre su approssimazioni digitali.

### 4.1.1 Erosione

L'erosione di una immagine binaria  $A$  è definita formalmente in termini di traslazione dell'elemento strutturante  $B$

$$A \ominus B = \{z \mid (B_z) \subseteq A\} \quad (4.1)$$

ossia,  $A \ominus B$  è l'insieme dei punti  $z$  dell'immagine tale che  $B$  traslato in  $z$  sia contenuto interamente in  $A$ . L'effetto della trasformazione è quello di contrarre le regioni corrispondenti agli oggetti. Ad esempio, questo tipo di operazione si può eseguire sulle immagini ottenute da una binarizzazione nel caso in cui oggetti che dovrebbero essere separati risultano erroneamente connessi. Mediante l'operazione di erosione si possono anche estrarre i contorni degli oggetti, erodendo inizialmente l'immagine con un opportuno  $SE$  per poi eseguire la sottrazione tra l'immagine originale e il risultato dell'erosione. In particolare, l'erosione di una immagine binaria per un  $SE$  quadrato  $3 \times 3$  consiste nel rimuovere dall'oggetto tutti i punti che hanno almeno un vicino, nel senso della 8-connettività, appartenente allo sfondo. In pratica, i contorni dell'oggetto vengono contratti di un pixel in tutte le direzioni. Per le immagini in scala di grigio l'elemento strutturante  $b(s, t)$  ha la forma del suo profilo di intensità. L'operazione di erosione sulla  $f(x, y)$  è eseguita in ogni punto  $(x, y)$ , in modo da ottenere il valore minimo nella regione coincidente con  $b(s, t)$  quando l'origine è posta nel punto  $(x, y)$ .

$$[f \ominus b](x, y) = \min\{f(x + s, y + t)\} \quad \text{con } (s, t) \in b \quad (4.2)$$

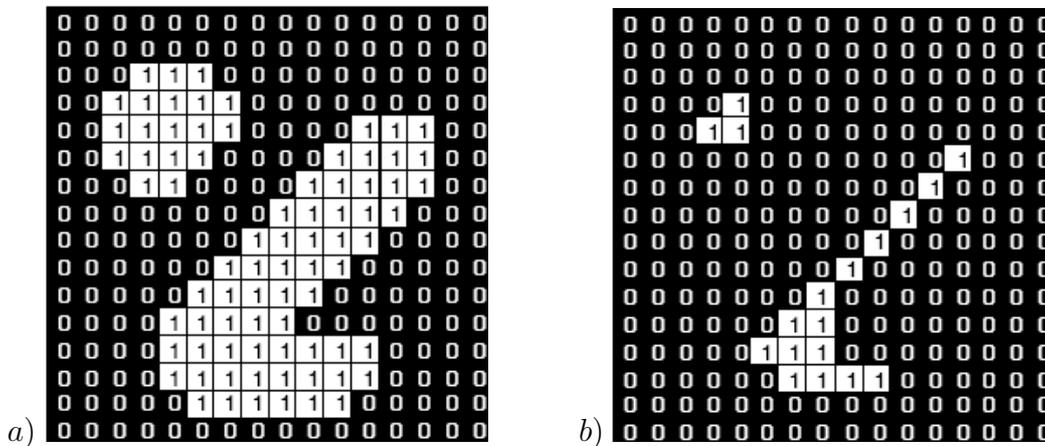


Figura 4.1: Immagine erosa tramite  $SE$  quadrato  $3 \times 3$

L'effetto prodotto dall'erosione consiste nel rendere l'immagine globalmente piú scura, attenuando i dettagli piú chiari e aumentando le dimensioni di quelli piú scuri.

La libreria dispone di apposite funzioni che consentono di eseguire le operazioni morfologiche. In ogni caso consistono in operazioni di convoluzione tra l'immagine e un elemento strutturante. La seguente funzione consente di creare e allocare un  $SE$  mediante la struttura *IplConKernel*.

```
cvCreateStructuringElementEx(int cols, int rows, int anchor_x,
                             int anchor_y, int shape, int * values = NULL);
```

I parametri interi *cols* e *rows* sono rispettivamente il numero di colonne e righe da assegnare all'elemento strutturante, mentre *anchor\_x* e *anchor\_y* sono le coordinate orizzontali e verticali dell'origine. In particolare, per tali coordinate il punto in basso a destra di un  $SE$  è  $[cols - 1, rows - 1]$ , così per una forma quadrata di ordine  $3 \times 3$  le coordinate dell'origine coincidenti con il centro geometrico sono  $[1, 1]$ . Naturalmente le coordinate dell'origine si possono scegliere in modo da non coincidere con il centro geometrico. Mediante il parametro *shape* si definisce la forma geometrica dell' $SE$ . Se al parametro si assegna il valore `CV_SHAPE_RECT` l'elemento strutturante corrisponde ad

una forma rettangolare, mentre con `CV_SHAPE_CROSS` si ha un *SE* a forma di croce. Ad esempio, la costruzione dell'elemento di dimensione  $45 \times 45$  e con origine  $[22, 22]$  nel centro di simmetria si ottiene ponendo:

```
IplConvKernel * element = cvCreateStructuringElementEx(45, 45,
                22, 22, CV_SHAPE_CROSS, NULL);
```

Con `CV_SHAPE_ELLIPSE` si definisce la generica forma ellittica, mentre `CV_SHAPE_CUSTOM` è il valore che bisogna passare alla funzione quando si vuole definire un *SE* con forma diversa da quelle predefinite. In tal caso il parametro *values* contiene l'indirizzo di memoria di un'array di interi che definisce l'elemento strutturante. Ad esempio, *int*  $a[9] = \{0, 1, 0, 0, 1, 0, 0, 1, 0\}$  corrisponde all'*SE* della seguente Tabella.

0	1	0
0	1	0
0	1	0

Tabella 4.2: Elemento strutturante non predefinito

In definitiva un elemento strutturante si può costruire tramite un array di interi  $a[\text{cols} \times \text{rows}]$ , in cui gli elementi non nulli opportunamente disposti tra loro ne definiscono la forma.

L'operazione di erosione morfologica viene eseguita dalla funzione

```
void cvErode(const CvArr * src, CvArr * dst,
```

```
IplConvKernel * element = NULL, int iterations = 1);
```

Quando il parametro *element* è mantenuto al valore di default, la funzione applica l'elemento strutturante predefinito di forma quadrata e dimensione  $3 \times 3$ . Il parametro *iterations* specifica il numero di volte che l'operazione di erosione deve essere eseguita.

Forniamo ora la porzione di codice che permette di reiterare per 5 volte l'operazione di erosione sull'immagine di Figura 4.2a utilizzando un elemento strutturante  $3 \times 3$  di forma quadrata.

```
...
//crea l'elemento strutturante
IplConvKernel* element= cvCreateStructuringElementEx(3, 3, 1, 1,
CV_SHAPE_RECT, NULL);

// erosione con 5 iterazioni
cvErode(src, dst, element, 5);

// rilascia l'elemento strutturante
cvReleaseStructuringElement( &element );
...
```

Osserviamo dall'immagine 4.2b come le regioni vengono modificate in funzione della forma geometrica scelta per l'elemento strutturante. Le regioni quadrate vengono contratte mantenendo invariata la loro forma, mentre quelle circolari e esagonali oltre a essere contratte sono anche deformate secondo la forma dell'elemento strutturante. In particolare si nota l'azione dell'elemento *SE* sui piccoli contorni interni alle due regioni circolari, in basso a sinistra e in alto a destra, così come l'azione di separazione messa in atto dall'erosione tra le due regioni connesse (quadrata e circolare). La separazione completa si ottiene con 17 iterazioni.

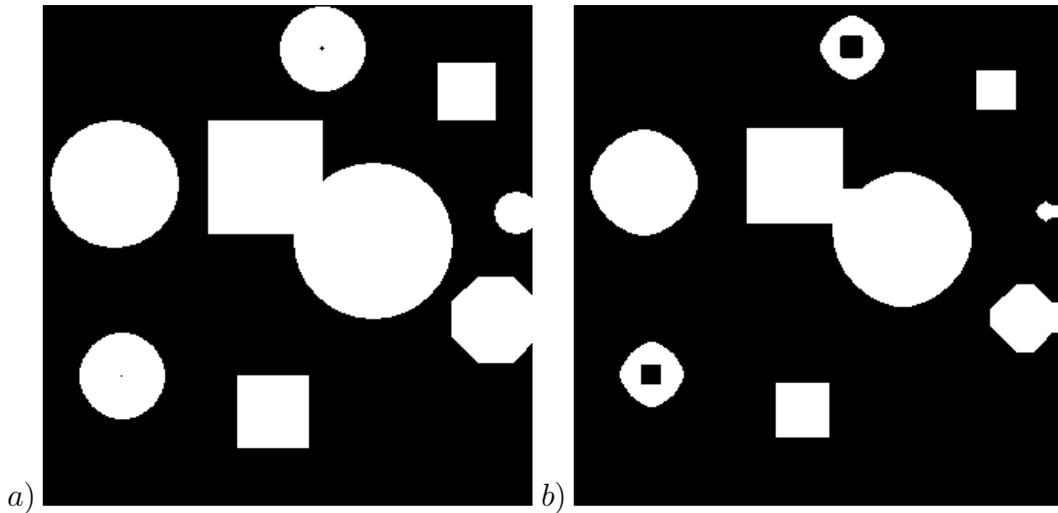


Figura 4.2: Immagine erosa con 5 iterazioni tramite  $SE\ 3 \times 3$

#### 4.1.2 Dilatazione

La *dilatazione* di una immagine binaria  $A$  tramite un elemento strutturante  $B$  è definita come

$$A \oplus B = \{z \mid (B')_z \cap A \neq \emptyset\} \quad (4.3)$$

L'elemento strutturante viene riflesso rispetto alla sua origine e spostato di  $z$  posizioni mediante una traslazione. La Figura 4.3 mostra un esempio di dilatazione dell'insieme  $A$  tramite un  $SE$  simmetrico ( $B' = B$ ). Il risultato è l'insieme di tutti i punti di posizione  $z$  per cui  $B'$  ed  $A$  si sovrappongono almeno in un punto. La trasformazione è estensiva poichè l'insieme originario è contenuto nell'insieme dilatato. La dilatazione può essere utilizzata per migliorare la qualità delle immagini ottenute dalla binarizzazione nei casi in cui delle regioni presentano lacune, oppure quando parti di un'oggetto che dovrebbero essere connesse risultano invece frammentate. In particolare, la dilatazione eseguita tramite un  $SE$  quadrato  $3 \times 3$  consiste semplicemente nell'espandere i contorni degli oggetti di un pixel in tutte le direzioni. La Figura 4.4 rappresenta la dilatazione eseguita sull'immagine 4.1a. Il risultato ottenuto consiste nel riunificare le due parti dell'oggetto. Per un'immagine in scala di grigio la dilatazione tramite un'elemento strutturante riflesso rispetto



La funzione che implementa l'operazione di dilatazione è

```
void cvDilate (const CvArr * src, CvArr * dst,  
IplConvKernel * element = NULL, int iterations = 1);
```

Anche in questo caso si può operare sia con un *SE* predefinito che con un elemento opportunamente creato dall'utente. Il significato dei parametri è analogo a quanto esposto per la *cvErode()*. La Figura 4.5 è il risultato che si

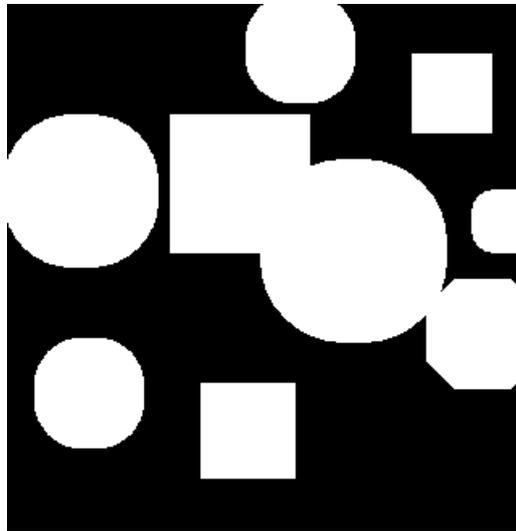


Figura 4.5: Dilatazione ottenuta con 5 iterazioni tramite *SE* quadrato

ottiene reiterando per 5 volte l'operazione di dilatazione sull'immagine 4.2a tramite un *SE* quadrato di dimensione  $3 \times 3$ . Si può notare che la regione avente forma esagonale si connette alla regione centrale dell'immagine. Un esempio di come applicare in modo utile tale caratteristica è rappresentato dall'immagine 4.6, in cui i caratteri del testo presentano delle interruzioni di lunghezza massima di due pixel. È possibile infatti riparare i segmenti rotti mediante il riempimento delle aree vuote. Si utilizza l'operazione di dilatazione tramite un elemento strutturante  $3 \times 3$  avente forma a croce. Il risultato ottenuto è rappresentato in Figura 4.7, dove possiamo notare la tipica forma a croce che riempie gli spazi vuoti.

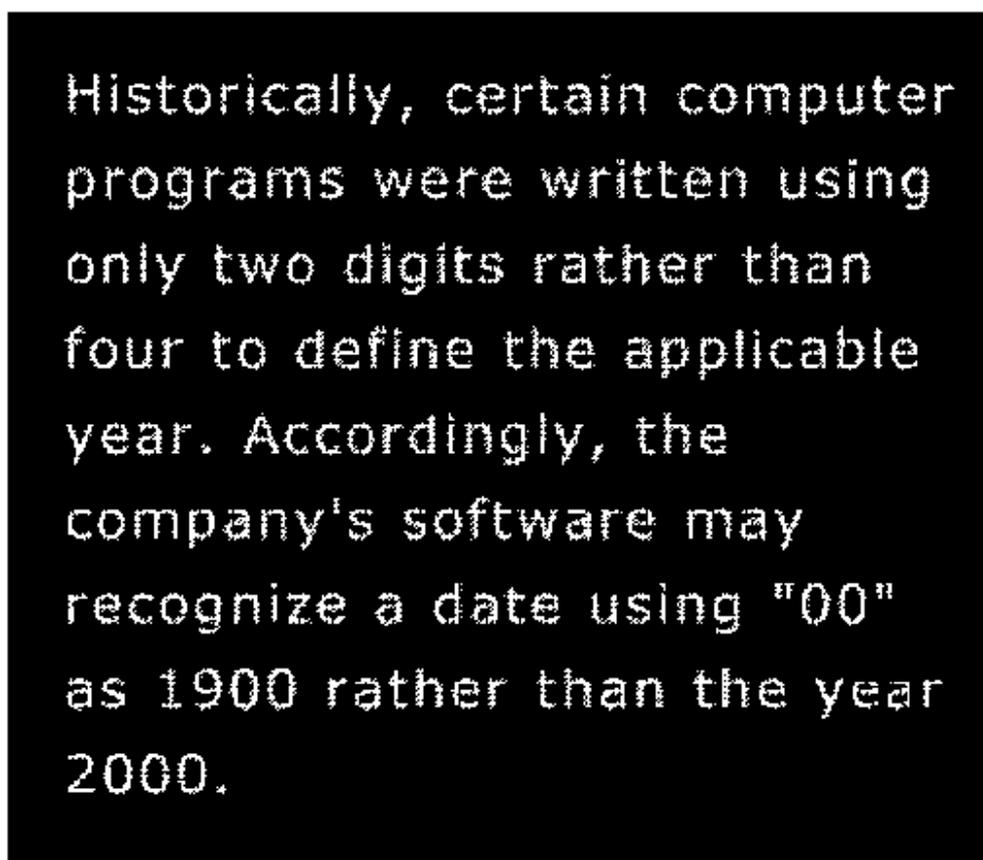
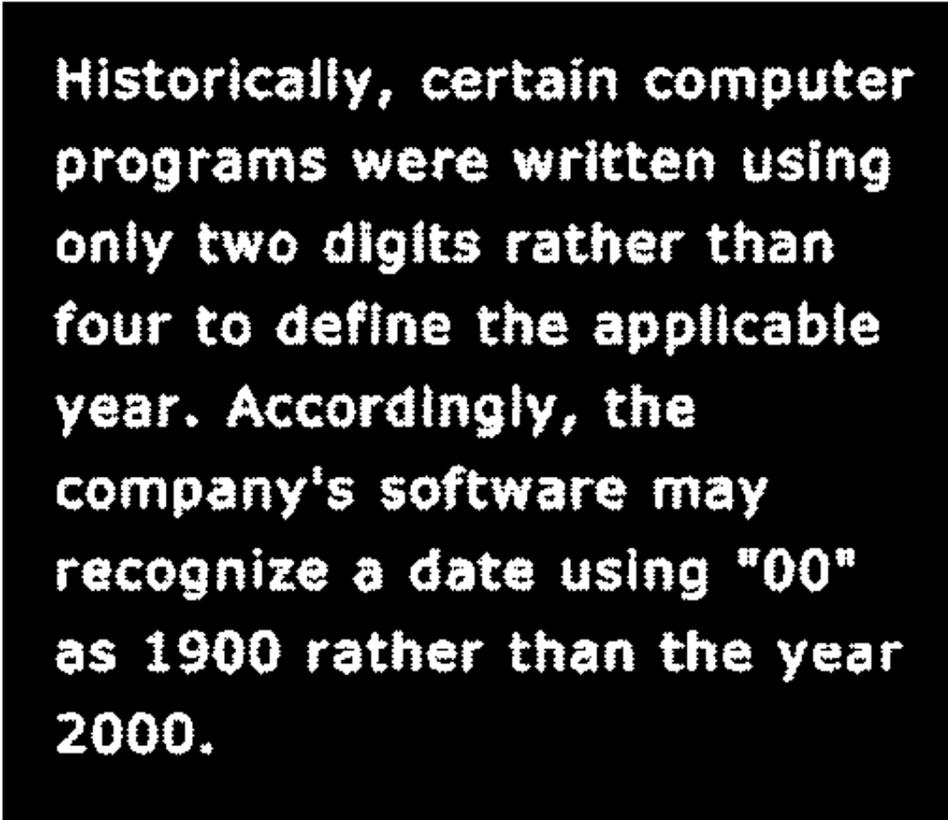


Figura 4.6: Immagine originale contenente un testo con caratteri rotti

## 4.2 Operazioni morfologiche complesse

Le operazioni di erosione e dilatazione per uno stesso elemento strutturante permettono di definire operatori più complessi. Eseguendo l'erosione di un insieme  $A$  attraverso un elemento strutturante  $B$  e la dilatazione del risultato nuovamente tramite  $B$ , otteniamo l'operazione di *opening*. L'effetto è quello di preservare le regioni di forma simile all'elemento strutturante, e di eliminare quelle differenti. L'operazione di apertura elimina i dettagli chiari più piccoli dell'elemento strutturante, rende più omogenei i contorni degli oggetti, ed elimina le piccole interruzioni.

$$A \circ B = (A \ominus B) \oplus B \quad (4.5)$$



**Historically, certain computer programs were written using only two digits rather than four to define the applicable year. Accordingly, the company's software may recognize a date using "00" as 1900 rather than the year 2000.**

Figura 4.7: Immagine riparata tramite dilatazione con SE  $3 \times 3$  a croce

La dilatazione di  $A$  attraverso  $B$  seguita dall'erosione del risultato nuovamente tramite  $B$ , costituisce invece l'operazione di *closing*

$$A \bullet B = (A \oplus B) \ominus B \quad (4.6)$$

L'operazione di chiusura elimina i dettagli più scuri e rende le sezioni di un contorno più omogenee riempiendo i vuoti. La funzione che esegue le operazioni morfologiche utilizzando l'erosione e la dilatazione come operazioni base è la seguente

```
void cvMorphologyEx(const CvArr * src, CvArr * dst, CvArr * temp,
                    IplConvKernel * element, int operation, int iterations = 1);
```

Oltre agli usuali parametri relativi all'immagine di input e output, all'elemento strutturante e al numero di iterazioni, la funzione dispone anche del

parametro *temp*. Questo si riferisce ad un'immagine temporanea, la quale è utilizzata solo nelle operazioni *top-hat*, *black-hat* e del *gradiente morfologico*. Il parametro *operation* specifica invece il tipo di operazione che deve essere eseguita. Assegnando il valore CV\_MOP\_OPEN la funzione esegue l'opening morfologico, viceversa con il valore CV\_MOP\_CLOSE esegue l'operazione di closing. Altra operazione è il gradiente morfologico CV\_MOP\_GRADIENT, il quale consiste nella differenza tra la dilatazione dell'immagine e l'erosione della stessa

$$g = (f \oplus b) - (f \ominus b) \quad (4.7)$$

Il gradiente morfologico esalta le regioni con un'elevata variazione di livelli di grigio. I contorni sono evidenziati e le aree omogenee eliminate, ottenendo un effetto analogo al gradiente derivativo. Ciò dipende dal fatto che la dilatazione tende ad espandere le regioni e l'erosione tende a ridurle, mentre la differenza tra le due esalta i contorni. Con il valore CV\_MOP\_TOPHAT la funzione esegue la trasformazione denominata top-hat, la quale consiste nella differenza tra un'immagine e la sua apertura

$$T_{hat}(f) = f - (f \circ b) \quad (4.8)$$

L'operazione di opening rimuove gli oggetti di interesse attraverso l'elemento strutturante, mentre la differenza tra l'immagine originale e l'apertura produce una nuova un'immagine contenente solo le componenti rimosse. La trasformazione si esegue in genere sulle immagini che presentano oggetti chiari su uno sfondo scuro, e in particolare per correggere l'illuminazione non uniforme. Infine, con il parametro *operation* posto a CV\_MOP\_BLACKHAT la funzione esegue la trasformazione denominata black-hat, la quale consiste nel sottrarre l'immagine alla sua chiusura.

$$B_{hat}(f) = (f \bullet b) - f \quad (4.9)$$

Al contrario della precedente questa trasformazione si applica prevalentemente quando si hanno oggetti scuri su uno sfondo chiaro.

La Figura 4.8 è il risultato dell'operazione morfologica del gradiente eseguita sull'immagine 3.9 tramite un *SE* di forma quadrata  $3 \times 3$  e con origine

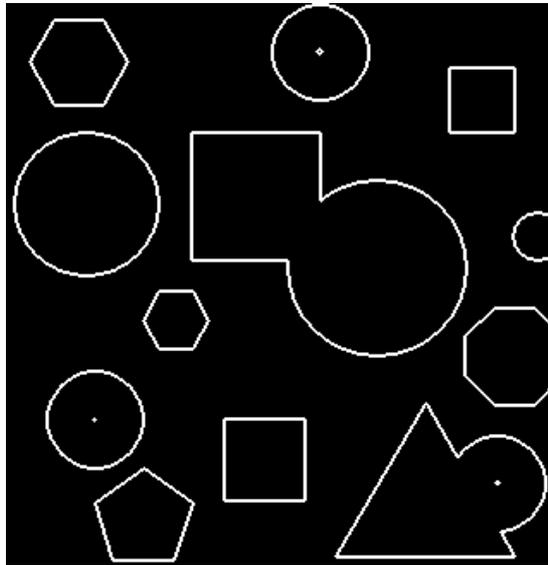


Figura 4.8: Gradiente morfologico

nel centro di simmetria. I contorni delle regioni sono facilmente isolati poichè tra le forme e lo sfondo vi è una netta variazione di intensità.

```
//elemento strutturante  
element=cvCreateStructuringElementEx(3,3,1,1,CV_SHAPE_RECT,NULL);  
  
//operazione morfologica del gradiente  
cvMorphologyEx (src, dst, tmp, element, CV_MOP_GRADIENT, 1);
```

La stessa operazione viene eseguita sull'immagine a toni di grigio di Figura 4.9a. Il risultato dell'operazione mostra i contorni individuati tra le varie regioni con il tipico effetto "gradiente".

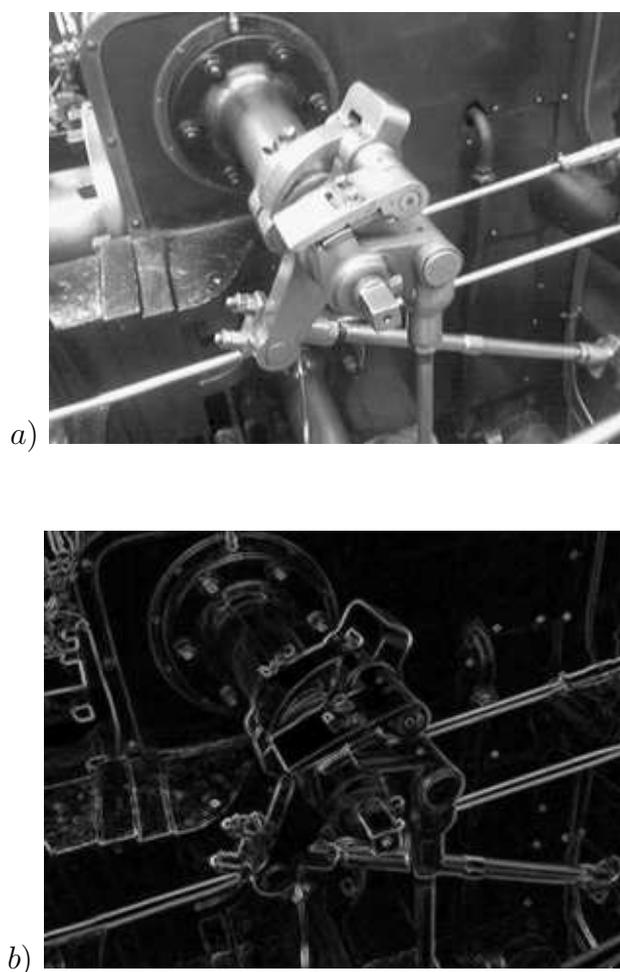


Figura 4.9: Immagine in scala di grigio raffigurante le componenti meccaniche di un motore e relativo gradiente morfologico

Gli operatori di opening e closing si possono utilizzare anche come filtri morfologici per rimuovere il rumore. Consideriamo ad esempio la Figura 3.3, caratterizzata da rumore di tipo salt e pepper. Applicando l'operatore di opening tramite un  $SE$  di dimensione  $3 \times 3$  avente forma quadrata, otteniamo l'immagine 4.10a. L'azione dell'operatore ha rimosso il rumore di tipo salt (punti chiari) lasciando invece il rumore di tipo pepper (punti scuri). Come si può osservare l'apertura ha prodotto anche un incremento delle dimensioni dei punti scuri di foreground. L'operazione di closing eseguita sull'immagine

4.10a con un  $SE$  di dimensione  $3 \times 3$  non è quindi sufficiente ad eliminare il rumore residuo, come si osserva dalla 4.10b.

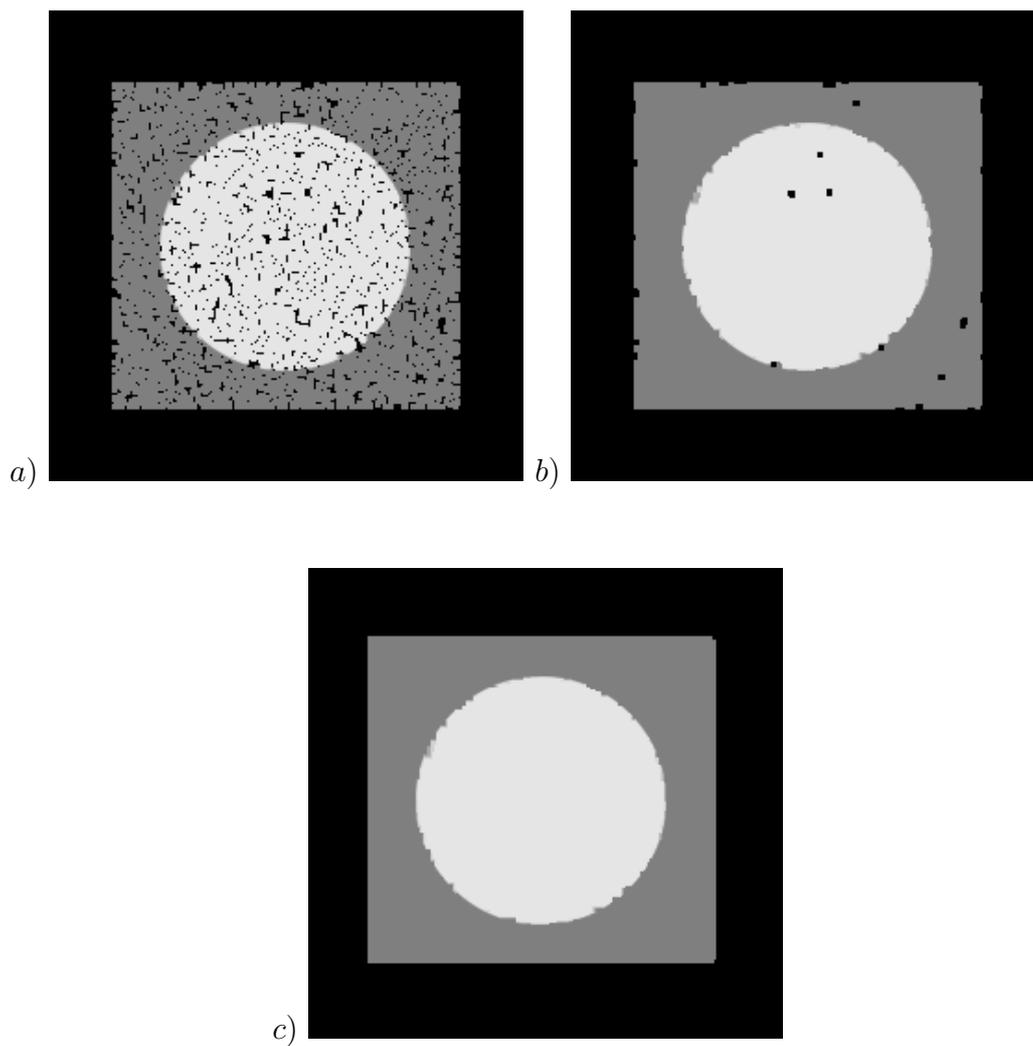


Figura 4.10: Opening e Closing per la rimozione del rumore impulsivo.

È necessario reiterare l'operazione per avere un risultato accettabile, oppure si deve utilizzare un  $SE$  di dimensioni maggiori. Nel nostro caso l'immagine 4.10c è ottenuta iterando per 3 volte l'operazione di closing. Possiamo confrontare questo risultato con quello di Figura 3.3b ottenuto tramite un filtro mediano di smoothing. Notiamo che in entrambi i casi il rumore è stato eliminato. In particolare, nella forma quadrata di foreground (immagine 4.10c) l'operazione ha prodotto un bordo liscio e privo di tratti vuoti. La stessa cosa non vale per il cerchio, il quale presenta un bordo più irregolare rispetto all'immagine 3.3b.

Consideriamo infine l'estrazione dei contorni attraverso l'elaborazione morfologica. Dall'immagine 4.11a eliminiamo i segmenti di linea per poi estrarre i contorni dai restanti oggetti di forma circolare. Il primo passo consiste nell'eseguire l'operazione di opening attraverso un  $SE$  di tipo circolare e di dimensione  $11 \times 11$ . L'elemento strutturante ha le stesse dimensioni dei cerchi in modo da preservarne il più possibile la forma. L'output che si ottiene è dato dall'immagine 4.11b. Come possiamo notare i segmenti di linea sono stati eliminati, mentre i blob ad essi sovrapposti sono quelli che risultano maggiormente deformati. Il passo successivo è l'operazione di estrazione dei contorni, sintetizzata dalla seguente equazione

$$\beta(A) = A - (A \ominus B) \quad (4.10)$$

In tal caso  $A$  corrisponde all'immagine 4.11b,  $B$  è l'elemento strutturante di forma circolare  $5 \times 5$ , mentre  $A \ominus B$  è l'erosione di  $A$  attraverso  $B$ . Il contorno si ottiene quindi dalla differenza tra  $A$  e la sua erosione. Il risultato  $\beta(A)$  è rappresentato dall'immagine 4.12.

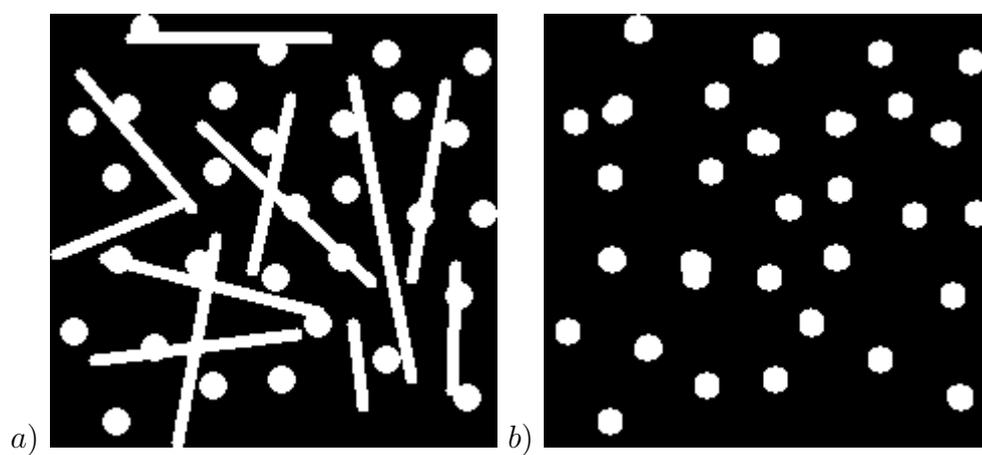


Figura 4.11: Opening attraverso un  $SE$  circolare di dimensione  $11 \times 11$

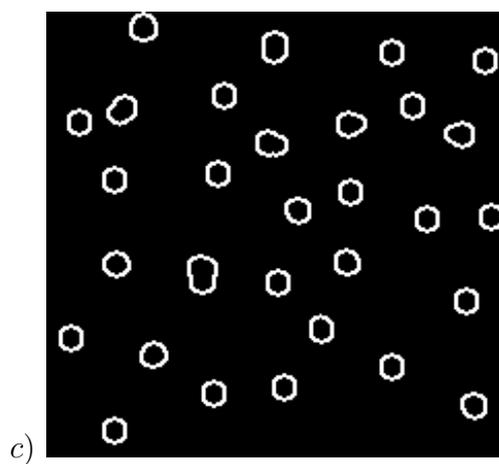


Figura 4.12: Estrazione dei contorni

# Bibliografia

- [1] Rafael C. Gonzales - Richard E. Woods: *Digital Image Processing*, 3<sup>rd</sup> edition, Pearson Prentice Hall, 2008
- [2] Dana H. Ballard - Christopher M. Brown: *Computer Vision*, Prentice Hall, Inc. 1982
- [3] Ian T. Young - Jan J. Gerbrands - Lucas J. van Vliet: *Fundamentals of Image Processing*, Delft University of Technology, 1995.
- [4] Adrian Ford - Alan Roberts: *Colour Space Conversions*, 1998.  
<http://www.poynton.com/PDFs/coloureq.pdf>
- [5] H.K. Yuen, J. Princen, J. Illingworth and J. Kittler. *A Comparative study of Hough Transform methods for circle finding*. Department of Electronics and Electrical Engineering University of Surrey, Guilford, GU2 5XH. U.K.  
<http://www.bwa.org/bmvc/1989/avc-89-029.pdf>
- [6] Bilateral Filters  
<http://scien.stanford.edu/class/psych221/projects/06/imagescaling/bilati/html>
- [7] The Computer Vision Home page  
<http://www.cs.cmu.edu/~cil/vision.html>

- [8] CVonline: The Evolving, Distributed, Non-Proprietary, On-Line Compendium of Computer Vision.  
<http://homepages.inf.ed.ac.uk/rbf/CVonline/>
- [9] Immagini di Test  
[http://www.imageprocessingplace.com/root\\_files\\_V3/image\\_databases.htm](http://www.imageprocessingplace.com/root_files_V3/image_databases.htm)  
<http://homepages.inf.ed.ac.uk/rbf/HIPR2/>